A Framework for Access Control and Resource Allocation in Federations

Soner Sevinc

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

Advisor: Larry Peterson

Abstract

In this thesis we address the access control and resource allocation problems in computational federations, such as testbeds and cloud computing federations. The computational federations of today are growing in their number of participant organizations, where one challenge is to allow organizations participate autonomously by expressing how much of their resources should be used and by whom, through complex policies. In addition, such organizations should be able to exchange resources with any other organizations without necessarily knowing all of them beforehand.

We introduce our federation framework which allows to build federations in varying complexities easily, by synthesizing trust management, policy languages and resource discovery into a single system. Although these three have been studied separately in the past, we show that they are in fact related, and can be viewed as separate layers of a more general system. We argue that complex agreements that involve indirect trust relationships is one key way to enable resource exchange in a federation with numerous organizations, and this can be realized by our synthesis architecture that provides usability as well as expressiveness.

As part of our framework, federation policy language (FPL) is used to express both the security and allocation policies, by providing simple primitives such as contracts that hide the underlying complexity. FPL primitives allow system administrators to express policies such as indirect trust and resource restrictions within the same construct. Underneath, FPL uses our distributed trust management system (CERTDIST) to implement and impose policy primitives. CERTDIST uses digital certificates to allow or deny resource requests and a DHT for complex distributive proofs in an efficient way. The Resource discovery part of our framework (CODAL) is layered on top of FPL, and uses contracts to discover peers, FPL security and allocation policies to authorize for resources that are located possibly in many different organizations.

This thesis also involves a trust model analysis of today's federations and enabler technologies, which shows that simpler trust relationships have been used in these systems, but complex trust relationships is a logical evolutionary step. We evaluate the federation framework with a realistic emulation of a large scale federation using real PlanetLab traces, that shows that complex policies can be expressed with a minimal amount of code, and we can efficiently perform the access control and resource discovery operations in a federation.

Acknowledgments

I am grateful to my advisor Larry Peterson for his continuous support and guidance over the years. He helped greatly to take the right directions in densely studied research fields, improved my work by motivating to crystallize my ideas and be succinct. It has been catalyzing to involve me in the nation-wide GENI project and encourage the two internships that contributed largely to this thesis. Also, thank you for being extremely patient whenever I needed to see the big picture and present my work in better ways.

I cannot thank enough Andy Bavier who has been extremely helpful with both the general discussions and the detailed feedback on each chapter of this thesis. I appreciate his insightful remarks that led to structure this work as a framework to build federations, rather than a particular instance of federation. I thank him very much for sharing and boosting my motivation of clarity and perfection throughout the chapters.

I thank David Walker very much for his feedback and encouragement on the Federation Policy Language, which has become one chapter of this thesis. I appreciate Jennifer Rexford and Michael Freedman's remarks on my presentations that lead to improve the comparative analysis with respect to the previous work.

I am very grateful to Mary Fernandez and Trevor Jim whom I worked with at AT&T Labs research for summer internship and later collaborated on a paper. They have done enormous contributions to me and my work, thank you for being great mentors. The trust management work we have done together has been the foundation of the system described in this thesis, and the large scale federation emulation was a collaborative work with them, which shaped the final chapter of this thesis.

I want to thank Aaron Helsinger whom I worked with in BBN Technologies for summer internship. He has been extremely helpful in leading me towards the right questions which led to the trust model analysis of federations, which is another part of this thesis. I would like to thank Sapan Bhatia for his feedback on federation policy, Tony Mack for his help with PlanetLab database and usage logs that allowed evaluation through federation emulation. I thank Michael Wawrzoniak for his support and advice in presenting my work.

Finally, I am lucky to have had support and inspiration from, or collaborated with numerous friends and colleagues: Fethi M. Ramazanoglu, Gorkem Ozkaya, Serdar Selamet, Minlan Yu, Gungor Polatkan, Emrah Gurpinar, Emre Celebi, among others.

Contents

List of Figures

Chapter 1

Introduction

The word *federation* is defined in politics as "the act of constituting a political unity out of a number of separate states or colonies or provinces so that each member retains the management of its internal affairs." In more general terms, federation means a group within which smaller divisions have some degree of internal autonomy, and operate under common set of rules. In computer science, we study computational federations, where the goal is to share computational resources across organizational boundaries. While a single organization has limited resources, federation increases the amount and diversity of resources available to its participants. Today, an increasing number of federations are arising with requirements that are different from existing federated systems. This thesis explores the requirements for access control and resource allocation in such federations and explains a general federation framework that can be used to build federations.

1.1 Federation Examples

We give some examples that show that computational federations have been around for a while, but are recently gaining more attention in fields of cloud computing and testbeds. The examples are also instrumental in explaining the requirements of our federation framework.

One well-known federation is the **Internet**, which is composed of more than 50,000 autonomous systems (AS), each of which provides connectivity to a smaller geographical region. By federating with each other, ASes, and hence their users, gain access to the overall global Internet. A peering between two ASes is established by setting up physical connectivity between them, following an

out-of-band agreement on its terms. Such a peering agreement generally requires each AS to access each other's portion of the Internet, with upper/lower bounds on certain network metrics such as bandwidth, packet loss, and so on. For the case of users, Internet customers are identified and charged only by their home ISPs and access the Internet according to the some customer agreement. Therefore, for both users and ASes, there is a direct out-of-band knowledge between two entities.

InCommon [31] is a federation of more than 250 academic institutions in the United States. The goal is to share academic materials, where each participating institution grants the others access to its own documents and, in return, gains access to theirs. Authentication is done cryptographically, which, in contrast to the Internet federation, allows an institution to authenticate messages coming from any other one in the federation. This is performed with the assumption that each participant has a direct a priori knowledge of other institutions, hence, their public keys. An InCommon user is authenticated at the home institution with a user name and password and is authenticated and authorized at other institutions with credentials, that is, statements digitally signed by the home institution. REFEDS [55] and IGTF [30] are similar federations that aim to foster academic research, and operate using similar mechanisms.

Another area where federations are emerging is website and applications. **Google Apps** is set of applications such as email, calendar, and so on, provided by Google. There can also be third-party applications, called the marketplace applications, that are developed by independent software providers. Users who log in to use Google Apps may possibly want to use the marketplace applications too, but a separate sign-up is a burden. To achieve seamless authentication of users at marketplace applications, third-party providers federate with Google and authenticate users using their Google IDs. In addition, the marketplace applications might need to use user-specific resources residing at Google servers. In this case, they federate with Google for access to such resources. Similar to InCommon, it is assumed that marketplace applications know Google's public key so that they can authenticate users from Google, and also Google vets the marketplace application with a human-involved process before sharing the resources.

Global Environment for Network Innovations (GENI) [10] is a project aiming to federate numerous research infrastructures and testbeds to form a global testbed for network experimentation. Each testbed in GENI has a set of computational and network resources and possibly a set of users. These resources can be unique to the testbed, perhaps because of its geographical

location. The goal is to allow all users to have access to the overall global resource pool. In this scenario, users should be able to authenticate at different testbeds and, if authorized, access their resources. On the other hand, the individual testbeds should be able to authenticate any users system-wide and control how much of their own resources are being used and by whom. This project is still an ongoing effort, and the devised authentication and authorization mechanisms have to meet these requirements.

Cloud computing federation has similar goals to testbed federation—that is, sharing compute and network resources between cloud organizations. Cloud federation participants are commercial systems with large user bases. The goal of a federation of cloud companies is to increase the diversity and scale of the resources they offer to their customers, while keeping their costs to a minimum. Products like OnApp [46] aim at providing users a federated cloud experience with diverse resources from multiple clouds. In addition, in recent years there has been several studies on federated cloud computing, such as ones focusing on authentication [43], real-time applications [67], platform as a service solutions [48], and so on. Such research is also related to grid computing, where attribute [35], role based [69] and other access control systems across multiple administrative domains have been studied.

Examples suggest that federation is an important option for academic institutions, cloud companies and Internet services companies. Academic institutions can share their uniquely located resources, so that a researcher who is affiliated with a particular university can have access to world-wide resources [52]. Similarly, cloud companies can decrease hardware costs by federating with others, increasing utilization of overall resources, or diversifying geographical presence [34]. Smaller clouds can meet temporal high demand by utilizing idle resource from others. Similarly, a larger cloud company in North America can federate one in Japan in order to meet demand from customer who wants to run CDN-like applications, requiring geographical presence. Also, CPU intensive virtual machines with less strict network delay requirements can be moved to a remote cloud during day time, where the available CPU is much higher because it is at another time zone and overall CPU usage is lower. Finally, as a general vision, federation among Internet services can enable applications that are yet to be developed, provided that the individual companies have means to express and dictate how their services will be shared and with whom.

Three common problems in these example federations are multiplexing resources, access control and resource discovery/allocation. The first is achieved in cloud computing by virtual machines,

providing an abstraction to users independent of the underlying resource pool. Security threats caused by others sharing an organization's resources is another issue caused by multiplexing of resources. In today's cloud computing systems, this is solved by isolation of client's environment from the host's. Therefore, we can say that the resource multiplexing problem is solved to a large extent. As for the access control problem, one subproblem is the human related process of deciding policies about how much resource to be granted to whom. While cloud organizations apply pricing methods, testbeds base their policies on bartering so that others can benefit from their resources as well. Although humans can come up with a variety of allocation policies, current mechanisms to express and dictate policies are programmatically limited. Such mechanisms depend on direct knowledge between organizations, which is limiting when there are numerous organizations in a federation. In general, we need mechanisms to express policies, apply those policies in an automated fashion, discover resources allowed by those policies, and allocate them. This thesis will argue that a new system that handles resource authorization, discovery and allocation among indirectly known organizations is a gating function to expand federations. Once we have such systems in place, we can construct larger federations that will make the previous problems harder, such as more resource multiplexing, or lead to new ones not known before.

1.2 Requirements

In this thesis we focus on the access control and resource allocation problems in large scale federations. The model of federation that we envision in this thesis is as follows. A federation is made of numerous autonomous organizations that have their own resources and users, but want to provide their users a larger resource pool. Organizations follow some federation-wide policies to interoperate, but also specify their own policies about how their resources are used by their users and also other organizations. Policies can be very complex, such as roles, resource amount, capacity, time based dynamic policies, and so on. For example, some cloud organization can issue policy P1 = "CloudX gives discount to Testbed1's preferred users.", where the preferred users are defined by other policies issued by Testbed1. Another example policy can be that CloudX can grant "10% discount in peak hours and 20% discount other times, with maximum 10 virtual machines per user". Policies like this are represented as credentials, and can be passed to other principals in a federation. When a user issues a resource request to an organization, resources are

granted if the available credentials allow it. This set of credentials can also include authentication credentials, proving user's identity. The access control decision follows from a process that tries to prove that a chain of credentials imply eligibility for the resources.

Some specific questions raised from such a federation are, how can complex policies be expressed by organizations, how can a proof process be built that uses credentials to arrive at access control decisions, how are the available organizations and resources discovered in a federation, how can a resource request be mapped to multiple organizations' resources and acquired automatically, how can a federation scale to millions of organizations, how fault tolerant is a federation in the face of organizational failures and misbehaving entities, and so on.

We can categorize these questions under three main technical requirements: trust management, expressing policies and resource discovery. These are fields studied in the past, but lack features to fulfill federation requirements which we explain next in more detail.

1.2.1 Trust Management

Authentication and authorization of principals in a distributed system is performed by a trust mechanism that discovers/collects and uses credentials to verify conformance to the policies. For example, when a Google Apps user tries to log in to a marketplace application, the user's authentication tokens are transferred from Google to this application using OpenID [47]. Similarly, security tokens are transferred between two InCommon participating organizations using SAML, Security Assertion Markup Language [57], when one's student wants to use resources at the other. Such mechanisms are commonly referred to as identity management systems.

Federations that we study in this thesis, on the other hand, require trust management, which is a more complex approach to access control than the identity management systems. Trust management systems were studied in the past [14, 16, 32, 37], which can construct complex proofs of access control decisions using formal languages and digital certificates. The reason for this complexity is scale, because as the number of organizations increase, not every organization will necessarily know all others directly. Therefore, proof of access control in such federations will require many credentials to be discovered and used together to logically infer whether an operation should be allowed or not.

One requirement for trust management system for federations is efficient credential discovery and proof construction for arbitrary trust relationships. For example, a hypothetical cloud organi-

zation can issue the following policy: P1 = "CloudX gives discount to Testbed1's preferred users.". Then, the testbed can declare: P2 = "StateU is a preferred organization". Finally, the university identifies its student with P3 = "Alice is student at StateU". Each credential originates from a different entity, yet all need to be present at CloudX so that Alice can use the discount. Credential chains like this, and possibly longer, constitute proof for access control, which is not always easy to construct. One challenge is that the credential may not always be present at its subject. For example, Alice may not have P3 when she visits CloudX. In that case, it may be impractical for CloudX to traverse all preferred organizations of Testbed1 and check if Alice is associated with any of them. Subject may not have the credential maybe because it was not stored, or it was issued for a group of principals, thereby individual principals lacking knowledge of it, or it was issued for a short time interval within which it did not reach the subject. Therefore, we need a reliable and efficient mechanism to search and retrieve relevant certificates for a proof. Current trust management solutions assume the certificates are stored at the signer [32] or the subject [37]. We are not aware of trust management solutions that operate with centralized repositories. Although some Grid computing federations use centralized credential stores, such as CAS [50] and Permis [20], they do not apply trust management, hence have simpler trust structures, which will be discussed in more detail in Section 2.6. Even if all federation participants agreed on a central location to store certificates, it is hard to ensure efficient and reliable operation of a centralized repository in an autonomous and distributed system.

A second requirement for a trust management system in a federation is programmable credentials. Policies in cloud computing and testbeds may be more complex than string attributes such as "discount" in the example policy P1. CloudX can issue instead "10% discount in peak hours and 20% discount other times, with maximum 10 virtual machines per user". Such policies can get arbitrarily complex, and therefore can be represented only with a program getting specific parameters, such as time. Furthermore, assume that CloudX grants this credential for its resources located at some other organization Cloud0. Therefore, Alice should present all three credentials, in terms of digital certificates, to Cloud0 to be able to use the discount. To our knowledge, none of the trust management systems in literature support both distributed credential discovery and fully programmable credentials.

1.2.2 Expressing Policies

Policy languages and their requirements were explored in the past in the context of testbed federations such as sfatables policy engine [11], and Grid computing [33,66]. The goal of these systems is to allow an organization to express the amount of its resources that can be allocated by others in a federated system. The federations that we study in this thesis require expression of not only the local resources but also third party resources, which is possible by using programmable credentials supported by the trust management system. We can generalize the relation of the policy language to the trust management system such that the language should provide an easy way to program the credentials. There are three overall requirements of the federation policy language, which we explain by using the example policy in Section 1.2.1.

First, the policy language should be able to express complex resource amounts and limits that can be seen in cloud computing and testbed federations. Attributes like "discount" of example policy P1 can get more complex and be expressed by programs, as mentioned before. Some other examples are maximum 100 machines, 10% of free capacity, 2GB/sec bandwidth during night time and 1 GB/sec day time, machines except in set X, and so on. Existing systems like sfatables can express such complex resource allocation policies, by providing simple primitives that exposes metrics relevant for computational resources, the implementation of which are with complex programs that compile and process policies in the backend.

Second, the language should be able to express the credentials required for the policy. For example, one should be able to precisely specify what it means to be a "preferred" user. Such policies are generally defined by a set of credentials, and called security policies. While the trust management system is responsible for construction of the proof of the policy, the goal of the policy language is to provide simple primitives and libraries to allow one to easily express the required credentials. Existing trust management systems provide the underlying mechanism to express security policies, but they lack higher level abstractions for cloud and testbed federation.

Finally, a policy language should be able to express third party resources. We mentioned that at P1, CloudX could grant its resources located at Cloud0 instead of the local resources. For example, Cloud0 may have issued P0 = "Cloud0 gives maximum 100 machines to CloudX excluding any of its 10 quad core machines". So, when CloudX grants some of its resources located at Cloud0 to Testbed1, it might mean only the ones granted through P0, or possibly also other credentials issued before. Therefore, such policies refer to other policies in a nested way in order to delegate

third party resources. Since this delegation can get arbitrarily long, the proof of access control needs a chain of programmable credentials presented at the resource origin. Existing systems does not support such complex third party resource delegation, which depends on trust management integration and programmable credentials.

1.2.3 Resource Discovery

Resource discovery systems in literature investigate how to map a user's resource requirements to the available resources in a distributed system. These involve specification of the request, gathering of resource availability information, and mapping the request to resources by possibly maximizing some utility metrics. Federations need such a mechanism since it is not practical for users to know where any requested resources may reside, especially in a large scale federation.

One distinct requirement of resource discovery in federations is that the system should be able to process policy as well as resource availability in mapping request to resources. For example, in SWORD [2] which runs on PlanetLab, a user can ask for "10 machines at North America with 10GB free disk, 1GB free memory"'. SWORD focuses on fulfilling the request in a way to optimize resource utilization, by reducing some penalty metric, however, no complex policy is involved, since all requests are administrated by PlanetLab. In a larger scale federation, on the other hand, the policies of organizations in which the machines reside, and also delegator organizations' policies, if any, would opt into the decision process. Therefore, the discovery system should be able to execute the complex programs written by the policy language. Moreover, since discovery may be possible over indirect paths, system should be able to use multiple credentials to arrive at a resource mapping.

Second, a single user request should be able to be fulfilled from many organizations, since not all pieces of the request might be found at only one organization. This should be done in a seamless way, requiring minimal participation from the user, and by automatically detecting, authorizing, and acquiring resources.

1.3 Contributions

This thesis shows that one practical way to scale federations of today is through complex indirect trust relationships, which can be realized with a system that synthesizes trust management, policy

languages and resource discovery systems. These three areas were studied separately in the past, but fail to fulfill both of the expressiveness and usability requirements of today's federations, where one key way to accommodate large number of participants is establishing complex trust relations easily. We argue that the three types of systems are interrelated, and should be seen as layers of a more general system, that we call the federation framework, which is crucial to realize such federations. Having reduced the barriers to expressiveness and ease of use, we believe this framework will enable resource sharing agreements that were not possible before, and allow federations with more number of organizations that can access to more diverse resources.

While synthesis of the three domains is the primary contribution of this thesis, we make individual contributions with respect to the three domains as well. First we analyze trust models of the federated systems and make the observation that indirect trust relations are inevitable for political and technical reasons in a federation. Access control in such an environment can only be achieved by a complex trust management system. To satisfy the requirements mentioned in Section 1.2.1, we develop a distributed trust management system called CERTDIST that has distinguishing features of distributed hash table (DHT) based credential store and retrieval, and programmable credentials, that is, certificates carrying programs. DHT allows key-based, in addition to IP-based certificate retrievals, making it possible to construct a wider range of policy proofs distributedly. DHT also increases fault tolerance and improves retrieval efficiency, especially in case of temporary glitches or overloads at federation participants. Programmable credentials allow integration of policy language with trust management, effectively allowing expressing and proving the complex policies required by the federations.

Second, in order to allow administrators to easily write policies with requirements mentioned in Section 1.2.2, we build a declarative policy language, FPL, on top of CERTDIST. FPL allows policy writers to easily express policies in a federation. Though it provides high level abstractions and primitives especially relevant for cloud and testbed federations, it can be used in other contexts as well. FPL exposes CERTDIST's certificate retrieval capability with its language primitives to able to write complex security policies. High-level FPL constructs such as "contracts" combine both security and allocation policy elements to express complex policies in federations concisely and easily. This includes ones expressing third party resources, as well, which use CERTDIST's programmable credentials underneath in its implementation.

Finally, in order to satisfy the requirements of Section 1.2.3, we introduce Contract-based

Resource Discovery and Allocation system, CODAL, that uses the policy language, FPL as its lower layer. It is different from previous systems [2, 68], which operate either in a single administrative domain or a federation with small number of organizations and simple inter-organizational policies. CODAL's "collaborative matching algorithm" can fulfill a user request from many organizations in the federation by utilizing FPL contracts to dynamically discover peer organizations and the security policies to authorize for resources. It performs constraint logic programming (CLP) to map a user resource request to available resources, which utilizes FPL elements to impose complex allocation policies in addition to user constraints.

We evaluate this framework in Chapter 8 by emulating a realistic large-scale federation using real PlanetLab usage logs and making use of the distributed nature of PlanetLab to create a federation model from scratch. Our evaluation shows that FPL can express complex policies, and CODAL can discover federation peers and allocate resources efficiently. We show that CERTDIST increases fault tolerance and efficiency in certificate retrievals, especially under high user-request loads at an organization.

Chapter 2

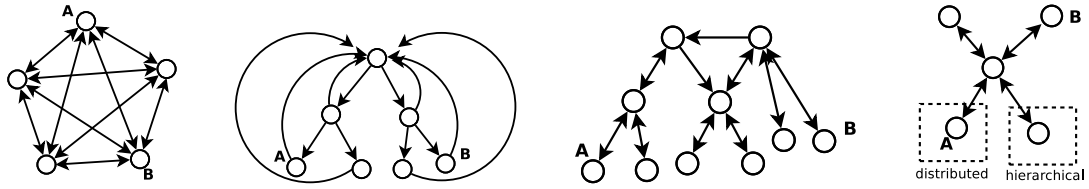
Background

This chapter gives more detailed information on the federations of today, and the technologies used, and contrasts them to the federation model we envision in this thesis. We first give a brief overview of trust models in literature, which we use to analyze federations. Next, each federation is explained in detail, including technologies used, the actors and their interactions, and their trust model analysis. We describe the technologies that do not particularly appear in a federation but studied separately, in Section 2.7. Finally, this chapter argues that federations of today have simpler trust structures than that of our federation model, and/or the technologies used are limited, lacking the expressiveness and usability properties required by a complex federation model.

2.1 Trust Models

To better characterize different federations, we use trust models that depict the trust relationships in the system. Trust models have been studied [39, 45] to analyze security of distributed systems. Depending on the complexity level of the trust model in a distributed system, we can characterize the required authentication and authorization mechanisms or argue if the currently used mechanisms are a limiting factor for the system.

Today, mostly simple trust relationships exist in distributed systems, in which simple authentication and authorization mechanisms are sufficient. For example, PlanetLab trusts Stanford University in attesting to valid students, so that when Stanford says that a particular individual is a student, PlanetLab accepts him or her as a valid user and later can give the right for ex-



(a) Circle of trust (b) Hierarchical (c) Distributed (d) Bridge

Figure 2.1: Trust models

perimentation in the testbed. This delegation is done by giving a managing role to a responsible person in Stanford who can input new Stanford users into the PLC database. On the other hand, trust relationships may get more complex in future systems. For example, similar to PlanetLab, ProtoGENI can choose to trust Taiwan University for its students. Then, PlanetLab may want to trust every university that ProtoGENI trusts in attesting new students, and in doing so, PlanetLab verifies students indirectly through ProtoGENI. In this scenario, a role-based delegation of the creation of user records will not work since we cannot expect PG to create all its user records at PL's database. Such trust relationship would require using a chain of digital certificates.

Figure 2.1 shows some well-known trust models: circle of trust, hierarchical, distributed, and bridge. A solid directed edge from a node A to node B shows A *trusts* B, which is established out of band between the two entities. More specifically, a directed edge represents an *assertion*, or a policy statement, made by A about B that could be stating a vouching relationship or delegation, depending on the policy. An assertion can be implemented as a single digital certificate signed by A. A *trust path* is a sequence of directed edges that begins with some node A and ends at a node B, showing an indirect trust from A to B.

In the *circle of trust* model (2.1(a)), there is a direct trust between every entity in the system. The length of any trust path is 1. The downside of this model is that it is difficult for a new node to join the system, since there need to be $O(N)$ trust relationships established out of band for N nodes.

In the *hierarchical* trust model (2.1(b)), every node trusts its children, if any, and the root. One example is a PKI system. In the process of authenticating a domain like princeton.edu, there are three assertions required. First, everybody trusts a root certificate authority, locally keeping its public key. This trust is depicted with the first edge going from node A to the root node. Next, the root has to sign a certificate for the edu domain, shown by the edge from the root to

the second-level node. Finally the third edge shows edu trusts princeton. It is easier for a new node to join in the hierarchical trust model, which is done by the new node downloading the root public key. The disadvantage is that the root becomes the single point of compromise that can affect the whole system.

In the *distributed* trust model (2.1(c)), there are multiple trust anchors and trust between them. This model fits environments with more autonomy, where there is not a single root of trust but multiple rootlike entities. One feature of this model is that trust paths can be longer; as seen in the figure, the maximum path length is 5, compared to lengths 1 and 3 in the other two models. The advantage of this model is that the compromise of one or more nodes does not affect the whole system. Also, similar to hierarchical trust model, it is easy to join the system, by establishing trust relationship to one or more participants. Distributed trust model is the most general and complex one, where all other models are a special case of it. It is also interesting because it reflects the trust relationships among humans for resource exchange. Because, generally entities autonomously decide the peers with whom they will share their resources, and in turn, those peers can decide their own peers. This results in an almost arbitrary set of paths for resources to be delegated and accessed.

Finally, the *bridge* trust model (2.1(d)) involves a bridge node connecting different parts of the graph, which can reduce the trust path length compared to distributed model. Bridge model can also be used in a situation where two or more groups of nodes with different trust models need to be connected, and nodes can talk to only the ones in their own group. In that case, when a node wants to exercise the trust path to a node in another group, its request has to go through the bridge node. This implies that the node logs in to the bridge, and the bridge talks to the target node on its behalf. This usage of bridge model is basically for providing a compatibility layer for the end points of the overall graph, and can be scaled by having multiple bridge nodes form a group that has, for example, a distributed trust model.

In the following sections, we use these trust models to analyze the trust structure of some existing federations.

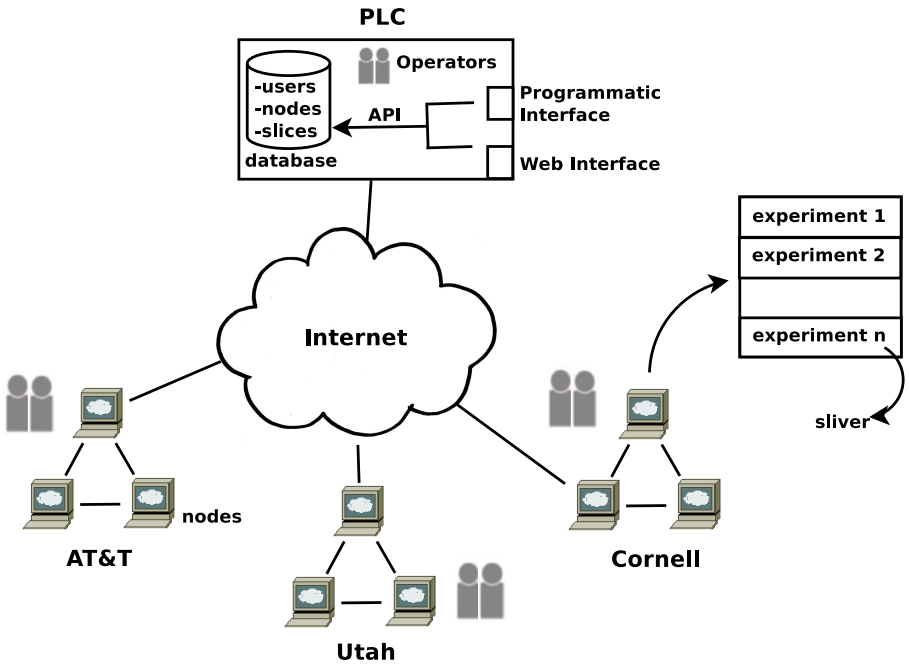


Figure 2.2: PlanetLab's centralized structure, where PLC manages all users and nodes from all sites, and the resource allocation using machine virtualization

2.2 PlanetLab Federation: SFA, sfatables

PlanetLab (PL) [52] is a research facility for realistic network experimentation that has machines that span the world. Figure 2.2 depicts the architecture of PL. Universities and research institutions, called *sites*, join PL by donating a set of machines and, in turn, gain access to the overall resource pool. PlanetLab Central (PLC) manages resource sharing, acting as a central trusted entity. It uses machine virtualization technology to allow machines be simultaneously shared by many *users*, researchers using the facility. A *node* is a single machine contributed by one of the sites. Users are allocated a *slice* of overall resources, which consists of a collection of individual virtualized nodes, called *slivers*. A slice can be used to perform a single *experiment* conducted by a group of users in the testbed.

The main property of this architecture is that PLC is the central trusted authority that manages the whole testbed. The user, node, and slice records are kept in PLC database and can be accessed and updated by the users through web or programmatic interface. New slices are requested by users, which have to be approved by the operators.

A single centralized testbed is limited in terms of both quantity and diversity of resources.

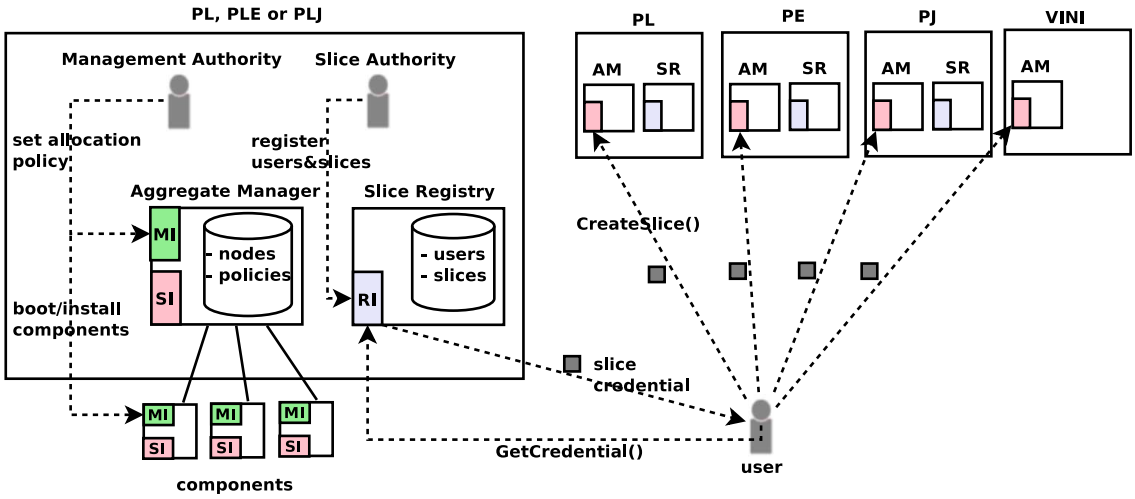


Figure 2.3: System actors and interfaces at SFA participant testbeds: (1) Authorities set allocation policies and create slices; (2) users access resources by collecting and passing credentials from/to authority-managed interfaces, RI and SI

Therefore, PL has chosen to federate with similar testbeds in order to increase the available resources to its users. Slice-based Federation Architecture (SFA) [51] is an ongoing effort for a federation architecture that aims to provide a set of standards and mechanisms that enable federation. Today, PL federates with other PL-like testbeds, such as PlanetLab Europe, Japan, and VINI [9]. A user of PL can use resources at those testbeds, and vice versa. The difference from the central structure is that each testbed is autonomous, managing their own resources and users. As a result, the resource allocation decisions are not made solely by PLC, but by the individual facilities and as a result of agreements between them.

SFA defines the system interfaces, principals, objects, and a trust mechanism, as depicted in Figure 2.3. A testbed has a set of *components*, each of which is an object representing a testbed resource, such as a single machine or a programmable router, and corresponds to a node in PL. Components export two interfaces, a *management interface* (MI) and a *slice interface* (SI). MI comprises the management operations, such as installing and booting machines. SI involves slice-related operations at a component, such as allocating, starting, and stopping a sliver. The *aggregate manager* (AM) of the testbed represents and manages all components within the testbed and exports the interfaces MI and SI. A user can invoke such operations at the AM or one by one at each component in a user's slice. A *management authority* (MA) is a principal who is responsible for the proper operation of the nodes and for placing the resource allocation policies at the AM or

components using the MI. These policies are set to fulfill the allocation preferences of the testbed or the resource owners. If the testbed has its own users, their records and the slices assigned to them are kept in *slice registry* (SR). A *slice authority* (SA) is a principal who is responsible for the creation of slices and approval of new users into the system. *Registry interface* (RI) exports record modification and retrieval operations at the slice registry. Users can call functions in RI in order to update their individual records, as well as to obtain credentials. Users can call "GetCredential" to get a slice credential in the form of a digital certificate, which denotes the user as a user of a particular slice. Using the slice credential, a user can allocate resources at any testbed in the federation by a call at the *slice interface* (SI) of their AMs or components. Currently VINI has only resources; therefore, it is depicted by an AM but not an SR. SI performs resource allocation, adhering to the policy established by the MA at the AM or component. All users in the system have a public/private key pair kept in the SR of their home testbed, which they use during remote calls and which SR uses while signing the credential for the user. SFA introduces a policy tool called sfatables [11], used by an MA to set resource allocation policies at an AM. This can place complex policies on resources, imposed on users who have a slice and want to allocate resources at the AM.

ProtoGENI (PG) [53] (not shown in the figure) is a federation effort led by the Emulab [63] research testbed, developed at the University of Utah. It brings together more than 10 Emulab-like testbeds in the United States, all of which run the Emulab code base for specifying and creating experiments. The difference between Emulab and PlanetLab is such that Emulab experiments focus on control and repeatability, whereas PlanetLab provides a realistic network environment in which an experiment runs in the real Internet. Emulab allows specification of network metrics, such as delay and bandwidth, to create an emulation of desired network conditions. Emulab allows an experiment to run in a real machine and allows choice and full control of the OS. Since the machine is not shared by others, this allows more repeatable experiments. Therefore, PL and Emulab provide two different points in the spectrum of network experimentation, where researchers use them back and forth for first controlled and then realistic testing of new services.

ProtoGENI contributes to SFA and adopts it as its federation architecture, allowing compatibility with the PL federation. Therefore, its actors and interfaces are similar to Figure 2.3; however, since these federations are developed by independent teams, they arrive at different implementations of the same architecture. The SI and RI are the most important pieces for compatibility,

since the MI is mostly called internally by the testbed management authority. The consolidation of such interfaces is crucial for federation between those systems, which is an ongoing effort.

2.2.1 Trust Model Analysis

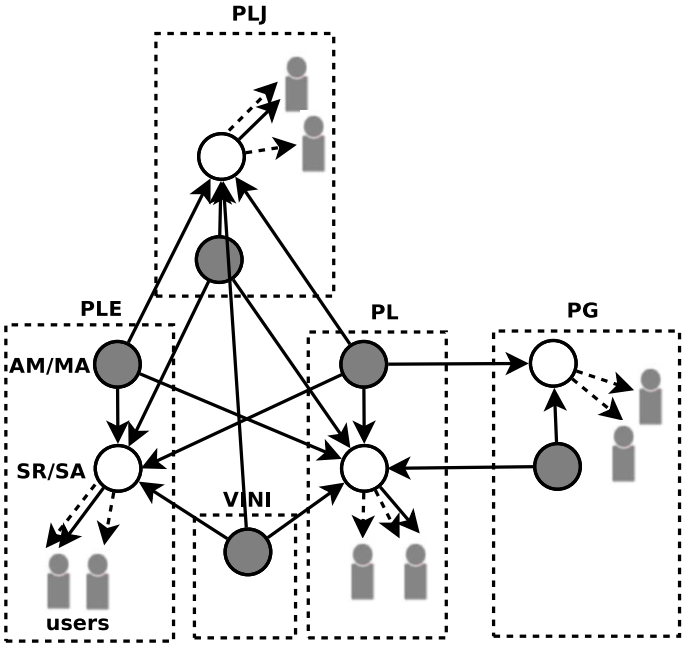


Figure 2.4: SFA Federation Trust Model: Circles represent the autonomous entities, or actors in a trust model, such that a gray circle is the MA or AM administered by MA; likewise, a light circle is SA or SR. Dotted and solid directed edges show identity trust and trust for resource allocation, respectively.

The PlanetLab and SFA trust model is shown in Figure 2.4. Each of PL, PG, PLE, and PLJ has both resources and users, whereas VINI has only resources. Circles represent the autonomous entities, or actors in a trust model, such that a gray circle is the MA or the AM administered by the MA; likewise, a light circle is SA or SR. For PL* and PG, there are both light and gray circles, whereas VINI is denoted by a gray circle only. Two types of directed edges distinguish between identity trust and trust for allocation of resources, where the former is represented by dotted edges and the latter, by solid edges. We use this convention as a convenience in later sections as well.

Each testbed has a set of trusted users, and it can decide to give a slice to some of those users. A slice credential is shown by the solid directed edges from an authority to its users. Testbeds give slices only to their own users. A directed edge from an AM to an authority means that the AM

trusts the slices from that authority; that is, it can create slivers for those slices. Therefore, a set of edges beginning at an AM and ending at a user shows that the user can create a sliver at that aggregate, constructing a trust path, as described in Section 2.1. In the current SFA federation, a slice from any PL* authority can be used to create slivers at the other PL* aggregates, as shown by edges from AMs to authorities for all PL*. On the other hand, VINI allows only PL and PLE slices; therefore, we have directed edges from VINI to PL and PLE authorities. This might be because a need for PLJ did not occur yet, because this is not negotiated yet, or because of other such human-related reasons.

The interoperability effort between PlanetLab and ProtoGENI is also seen in the figure. While the main focus of this effort is to make the credential format and API operations compatible, its implication in terms of trust is that both AMs trust other authorities for allocation of resources. We expect such trust relations will be constructed over time between ProtoGENI and other PLs as well.

In the current SFA federation, slices can be created only for the already registered federation users. This means a user who wants to create an experiment has to sign up at one of these testbeds. Users of such testbeds are generally already associated with their home institutions such as universities, and have accounts with user names and passwords. Therefore, a separate sign-up for the federation is generally a burden. Another limitation of the current SFA trust model is that indirect trust relations do not exist; for example, the fact that PL trusts PG does not imply that PLJ and PLE also trust PG.

2.3 InCommon: SAML

In contrast to SFA, InCommon does not require users to have public/private keys but, rather, user name and passwords. Authentication and authorization of users are done by a federated identity management system (FIM). The particular FIM used by InCommon is the single sign on (SSO) protocol specified by the SAML [57] standard. SAML SSO protocol involves two types of entities, service providers (SP) and identity providers (IdP). All the institutions in InCommon that want to share their academic documents must have the SP software, and the institutions that have users to use those resources have to have the IdP software. These software pieces are provided by an implementation of SAML called Shibboleth [60], which is the most widely used software by

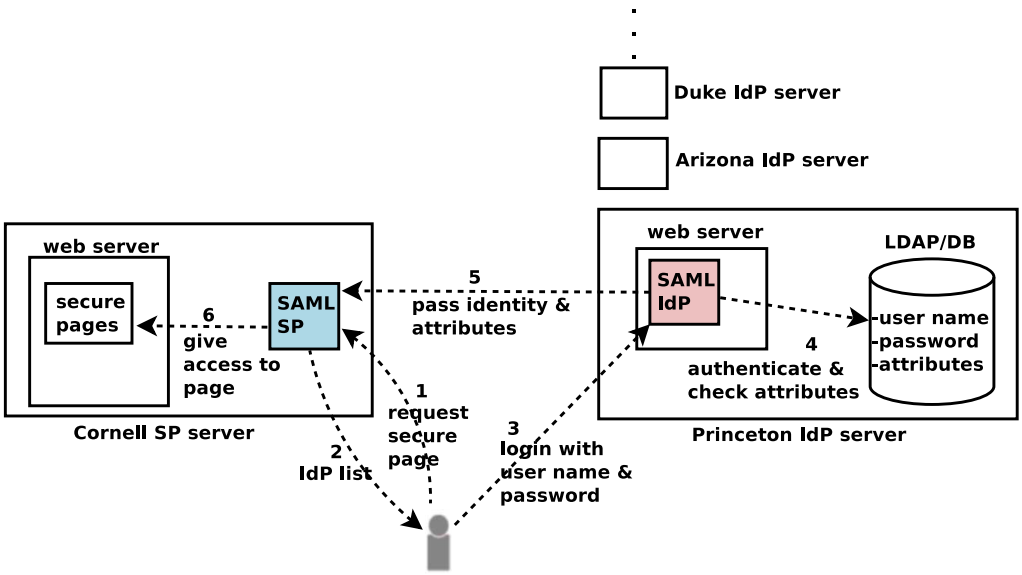


Figure 2.5: Use of SAML SSO Protocol in InCommon: A Princeton user requests documents at Cornell at step 1, which is interrupted by the SAML SP software at Cornell, which initiates the federated authentication and authorization (steps 2–6)

InCommon participants.

SAML SSO is exemplified in Figure 2.5. In the first step, a Princeton University student tries to view documents residing at Cornell University web server with the browser. The SP software intercepts the request to the web server and starts the SSO protocol to authenticate and authorize the user. In the second step, the user is returned a list of known and trusted identity providers (IdP) by the SP from which user selects his or her home institution, if it exists. Third, the user selects Princeton University from the list, is redirected to it, and is authenticated with the Princeton user name and password. The IdP software runs as a web application within the web server, which in the fourth step retrieves user identity and attributes from the institution's LDAP and databases. Attributes can be user properties such as affiliation, email, and so on, that can help the SP decide to give user access to resources or not. Fifth, if the user is authenticated, it is redirected back to SP together with the authentication and authorization tokens. Finally, at step 6, the user is granted access to the secure page at the SP server, where user's passed attributes are used to determine level of access for the user.

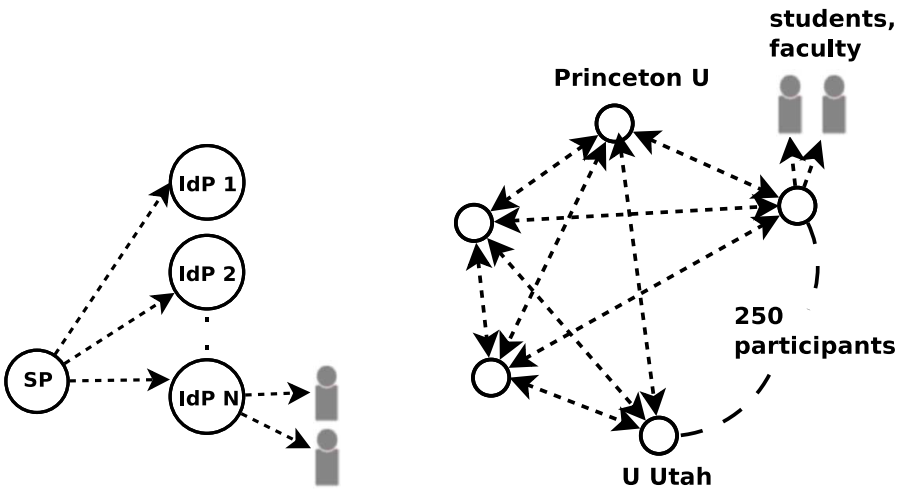


Figure 2.6: Trust models of (a) SAML SSO and (b) InCommon, which uses SAML SSO. Each participant institution has SP and IdP, which are not explicitly shown.

2.3.1 Trust Model Analysis

SAML SSO has a simple trust model where the service provider (SP) trusts the set of known identity providers (IdP) and the IdPs trust their own users, as shown in the Figure 2.6(a). A user can be authenticated by an SP only if there is a trust path starting at the SP and ending at the user.

InCommon uses SAML SSO and has a trust model, as shown in Figure 2.6(b). For the purpose of our discussion, assume that every member institution in InCommon, such as Princeton University or University of Utah, runs both a SAML SP and IdP. Actually, some might choose to just run an IdP if they do not have any resources to share or just run an SP if they do not have any users. So the figure shows directed edges from every SP to every IdP, where, for simplicity, we do not explicitly depict the SP and IdP pieces at each node. InCommon forms a trust model much like the circle of trust model between the institutions.

One limitation of the circle of trust model is that it can be hard for new members to fully join the system, since they need to establish direct trust relations with participants one by one. In this model, the existing member institutions generally get to know new members by periodically updating the list of public keys for all members. In the case of InCommon, this update operation might not be too inefficient, since participants are generally treated uniformly; that is, an institution shares its resources with all members or none. In contrast, if a participating institution

were picky about which members to trust or to what level, then it would have to review every new member separately. Therefore, in a federation where peering policies are more specific to institutions, the circle of trust model is less preferable.

On the other hand, unlike the PlanetLab federation, users do not need to sign up, but they can readily use the federation resources with their existing accounts in their home institutions. Another important feature of InCommon is that its user base is also interesting for other federations such as testbeds and cloud computing. The already-existing institution-to-user trust relations in InCommon can be made use of as part of other federations. For example, if PL were to trust InCommon for its users, then many users from academic backgrounds could use PlanetLab without extra sign-up overhead.

2.4 Google Apps: SAML and OpenID

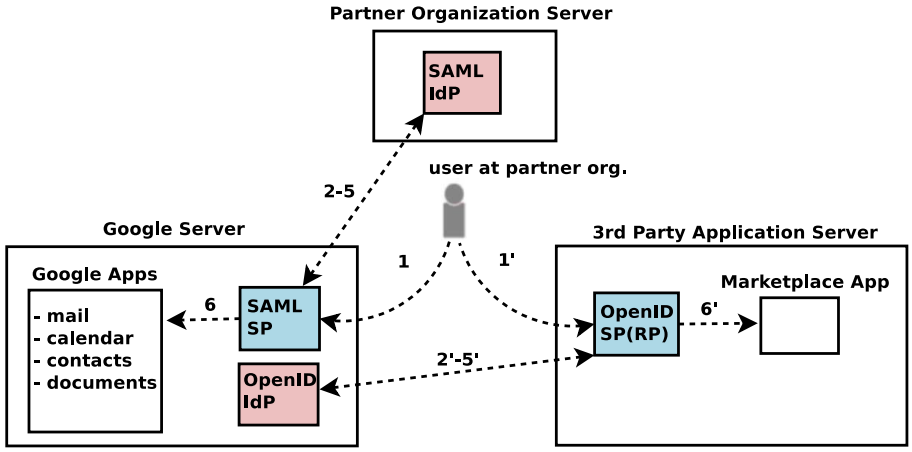


Figure 2.7: User's authentication at Google Apps and marketplace applications using the FIM technologies SAML SSO and OpenID

As mentioned before, Google Apps is an environment where both Google and independent organizations provide applications for users. Google Apps makes use of FIM technologies like SAML and OpenID. OpenID [47] is another standard like SAML for SSO and is mostly used for authentication. A user can login into an OpenID-supported web site using an ID from well known OpenID providers, like Google and Yahoo. Similar to SAML SSO, OpenID runs a message exchange protocol between domains to authenticate the user.

Figure 2.7 shows how Google Apps makes use of these technologies. There are two ways to log

into Google Apps. First, a user can be directly associated with Google and has a user name and password from Google (not shown in the figure); second, users of partner organizations can log in with their organizational user name and passwords using a SAML SSO (shown by the steps 1–6 in the figure).

There are again two ways to log into marketplace applications. First, a user already logged in at Google Apps can seamlessly log in at the marketplace application using OpenID. Second, the user can directly visit the marketplace application, and again OpenID authenticates the user after a series of steps very similar to SAML SSO, as shown by the steps $1'$–$6'$ in the figure. As a user logs in at the marketplace application, Google acts as the identity provider and has an OpenID IdP installation. On the other hand, the marketplace application is the service provider, also called the relying party (RP). Each such application has to implement OpenID login by installing the software for the OpenID RP.

There are special cases of the OpenID protocol that are different from SAML SSO and are not shown in the figure. For example, if the user is visiting the marketplace application by first logging on to Google Apps, then log-in step 3 is bypassed. This allows the user to seamlessly land in the marketplace application, for example, by following a link to the application within the Google Apps website. Second, OpenID can be used together with resource sharing federation technologies such as OAuth and can induce an additional step to ask the user whether or not he or she agrees to send identity and other information to the marketplace application.

2.4.1 Trust Model Analysis

The Google Apps trust model is shown in Figure 2.8. Similar to InCommon, users can use their home domain log-in information for authentication, using SAML SSO. In addition, they can also choose to sign up Google Apps and be directly trusted by Google. Marketplace applications trust Google for valid IDs, and they can indirectly authenticate both Google users or users from other domains using OpenID. As an improvement over the trust models of PlanetLab federation and InCommon, there can be indirect trust between two organizations in Google Apps, where Google is a trusted intermediary, acting as an identity broker. In this way, a third-party application provider can authenticate users from an independent domain even if it does not know them directly.

One drawback of this trust model is that there is a single trusted entity over all the federation. This means a user from an independent domain that wants to authenticate at a marketplace

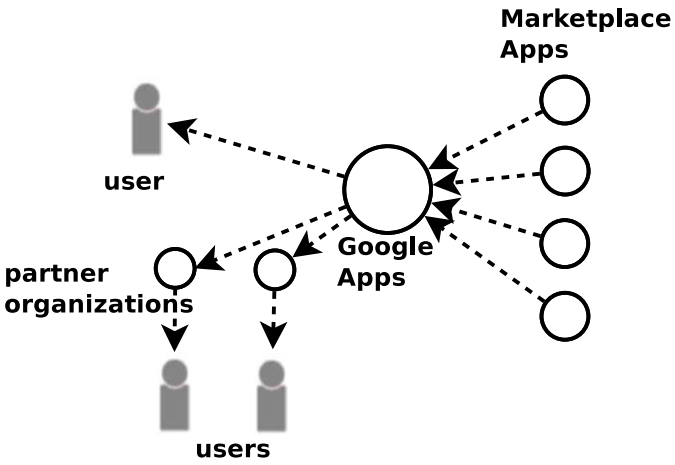


Figure 2.8: Google Apps Trust Model: Users from partner organizations can be authenticated at Google directly or indirectly and also at marketplace applications indirectly over Google.

application has to log into Google Apps first. In contrast, if a federation had many Google-like entities, a trust path from an application to a user could go through any one or more of the many intermediaries. The current authentication method of Google Apps is not suitable for this case because the intermediary to log into is not obvious to the user. Instead, such an environment would need an automated mechanism that discovers a trust path from user to app, constructs a proof of it, and provides that proof to the app that the user wants to log into.

2.5 GENI: SFA/ABAC

The GENI [10] project is led by the GENI Project Office (GPO), which promotes and coordinates different federation efforts in order to arrive at a ubiquitous federated testbed. The GPO supports SFA as a candidate architecture for GENI and contributes to the compatibility efforts such as, for API, credential and resource specification formats among implementations like PL and PG. Also, the GPO develops tools and prototypes testing the interoperability of such federations, informing the ongoing implementations the design.

Attribute Based Access Control (ABAC) is a federation enabler technology that is also proposed as a candidate for implementing GENI. ABAC formally expresses the assertions in a distributed system and constructs logic proofs of access control decisions by using these assertions. The assertions are in the form of attributes assigned to the system principals or logic rules about how

new attributes can be derived from existing attributes. ABAC [37] represents assertions with digital certificates and, distributively collects and uses them to construct proofs for access control decisions. We talk more about ABAC in Section 2.5.1.

2.5.1 Trust Model Analysis

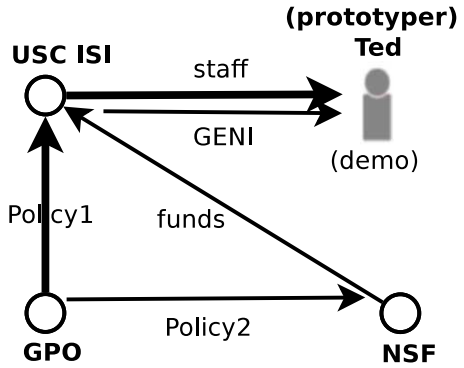


Figure 2.9: Example ABAC policy trust models: Bold directed edges show the trust path that proves Ted is a GENI prototyper, and the light directed edge shows Ted has demo attribute.

Figure 2.9 shows a trust model that illustrates trust paths of two example ABAC access control decisions. This federation example is from the ABAC wiki web page [6] and involves three organizations: the GENI Project Office (GPO), the National Science Foundation (NSF), and the Information Science Institute (ISI) at USC. There are two policies made by the GPO, which are as follows: Policy 1,"All ISI staff are GPO prototypers," and Policy 2, "All principals having GENI attribute from NSF-funded organizations have demo attribute." Therefore, for a user called Ted to prove that he is a GPO prototyper, the user's ABAC software has to collect a certificate saying he is ISI staff. When combined with the Policy 1 statement, the proof is complete. This is depicted by the trust path drawn by bold edges, beginning with GPO and ending with Ted. Similarly, to prove that he has the demo attribute, the user first gathers a certificate stating that NSF funds ISI and also a certificate stating that ISI says that Ted has the GENI attribute. Combining these two and Policy 2, proof of Ted having the demo attribute is complete, which is shown by the trust path with lighter edges on the figure.

Unlike Google Apps, there is a more complex trust path discovery in ABAC. Normally there will be many organizations funded by NSF in addition to ISI; therefore, ABAC's logic engine has the task of inferring that the organization has to be among one of those organizations that gives

Ted the GENI attribute. Then, the possible organizations will be contacted, and the assertion containing the attribute will be retrieved as a certificate to construct the proof.

The downside of the ABAC system is that it is more theoretical and the credential discovery does not have a full implementation at the current state. For this reason, the efficiency of the discovery and proof process at a large scale is unknown. In addition, ABAC credential discovery assumes certificates are located either at the issuer or the subject [38], which is not necessarily true in large-scale federations. ABAC cannot express policies like delegation depth, which can be done by more formal logic language based system like SD3 or DL. Finally, ABAC attributes do not include the resource allocation policies, which are not always as simple as roles. The federation framework described in this thesis aims to handle complex trust models similar to the ABAC model, but also to handle allocation policies.

While ABAC is one candidate proposed for GENI that can handle some complex trust relations, the actual deployments of GENI has been simple structures so far. More recently GENI design is aligned towards a bridge trust model, where a clearinghouse servers as the trusted intermediary [17]. This work is published after our work on trust structures of federations [58], and explains different types of trust and their role in GENI. We see this work as a natural evolution of GENI from circle of trust model to more complex trust models, in this case the bridge model. Since GENI is only in academic context, this architecture can be sufficient for medium term or it can serve as a stepping stone for the future.

2.6 Grid Computing

Grid computing aims to make computing available whenever and wherever needed, analogous to the power grid, where the computational power will be supplied by many administrative domains, which are hidden from the users of the Grid. This idea is very similar to the computational federations that we study in this thesis, such as federated cloud or testbeds. However, the primary deployments of Grid computing has been to support a few well known large projects that have large computational requirements, and therefore has to span multiple domains, such as SETI@home [4] and LHC [13]. This kind of usage requires simpler trust relationships among different organizations and also organizations and projects. Therefore, the technologies that run on deployed Grid computing platforms generally use identity or attribute certificates issued by known authorities,

with simple allocation policies. This is similar to the testbed federation examples we have given before, therefore we will not depict the trust models of such systems. Research on Grid computing introduces trust management and negotiation systems that can delegate attributes. However these systems do not address complex resource delegations among organizations through arbitrary paths.

Community Authorization Service (CAS) [50] was used in Earth System Grid [23]. CAS is used a trusted third party service, that keeps policies issued by all resource providers (RP) for all the users. A RP can grant some portion of its resources to the "community" by issuing a policy at the CAS server, and specify actions a user can perform such as read, write, execute on files. A user who authenticates at the CAS server can obtain credential to use a particular resource and present them at the RP, which allows the request if the credential is valid, or can also temporarily deny the request according to local policies. This is a bridge model of trust, where CAS acts as a clearinghouse trusted by every principal in the system to keep policies intact, hence is a single point of compromise. A similar system is PERMIS [20], where role-based policies are stored in a centralized LDAP server.

Virtual Organization Membership Service (VOMS) [3] is an authorization system that is used for European Data Grid (EDG) [27] deployment. It involves virtual organizations (VO) that consists of a group of users, who want to access resources located at resource providers (RP). Users can have different groups and roles within a VO, which can be obtained from a VO's VOMS server, and presented to RP, which grants resources to the user based on the particular VO(s) the user is part of, and the attributes user has. VOMS leaves it to RP to construct local policy mechanisms that should consume these attributes. The trust relationships are between VOs and RPs and also a VO and its users, that result in simple trust paths rather than multi-path.

GB-RBAC [69] is a role based access control system that simplifies assigning roles to users by introducing groups, and allows group-level administrators as well as system-level administrators within a single domain. However, it assumes direct trust relationships between any two domains are already established.

Attribute based multi access control, ABMAC [35] provides a common representation of local policies that use attributes. Local policies are represented by the logical conjunction of a set of functions, where the internal definition of functions are specific to different domains. Although the representation of local policies are logic based, it does not involve trust management.

Akenti [61] is a system developed for authorization to web resources, but also used in Grid context, which involves a more distributed authorization approach and also conditional credentials. When a user wants to access a resource, one or more policy files for a resource are consulted, which give the locations of a set of stakeholders. Then, the policies of stakeholders (use-condition certificates) are remotely obtained, which contain conditions, such as attributes needed to perform certain actions on the resource, and also which authorities should sign the attributes. Finally, attribute certificates are retrieved from the authorities. As a result, a longer trust path is walked compared to CAS or VOMS, when a user tries access a resource. However, there are no arbitrary delegations, such as delegation of stakeholder, authority or attribute in Akenti, which makes the trust model limited. Also, the policy language for Akenti is XML, which makes it hard to express a delegation logic with multiple credentials. On the other hand, Akenti use-condition certificates are expressive, and can specify conditions in terms of a boolean expression for an action to be performed. These conditions can be statements that can only be evaluated at the policy enforcement point (PEP) of the resource, such as about current machine load, disk availability, time or the state of some related system variable. This makes those credentials similar to the programmable credentials that we envision in our federation framework.

PRIMA [40] system uses privileges that can be delegated by users to others, allowing distributed collaboration. A privilege is defined by a subject, object (resource) and allowed actions on the object, such as allowed operations on a file. Although this represents a fine grained policy at certain contexts, it cannot express conditions, hence its expressiveness is limited. The goal of PRIMA is to enable smaller Grid collaborations, where users do not necessarily contact administrators to manage resource delegations. This goal is set as a real world user requirement, as a result of a Grid community survey. Therefore, PRIMA allows direct delegations between two users, forming indirect trust relations that improves scalability. We also adopt this as a requirement in our framework, and allow delegations among users. A set of privilege certificates will be analyzed at a PRIMA policy decision point to check validity and authoritativeness of each certificate, and specific actions will be allowed. However, the mechanism to discover/retrieve delegation certificates is not clear in PRIMA, and the privilege structure lacks programmability. This precludes any such delegations to occur between organizations as well.

Negotiating trust on the Grid [8] applies distributed trust management on the Grid resource allocation problem, and therefore is very closely related to our federation framework. It uses the

PeerTrust [42] distributed trust management system to dynamically negotiate and establish trust between entities previously unknown to each other. When a user tries to access a resource, an incremental disclosure of credentials takes place between parties, to arrive at a chain of certificates that reveals that user has the attributes that allow him to use the resources. PeerTrust uses a logic based language to express its policies, where the credentials can contain conditionals that express requirements for the policy to be valid, such as date time, or additional credentials. While these features allow PeerTrust to be able to express arbitrary attribute delegations, hence complex trust models, the programmability of the credentials are limited to security policies such as assigning attributes to principals with simple conditionals. Therefore, PeerTrust cannot express complex resource allocation policies, such as those that require the ability to operate on compound data structures, such as local accounting state of an organization at which a credential is being evaluated, a list of requested resources, a blacklist of resource units to be avoided, etc, to perform arithmetic operations such as quotas, and also to access to system related metrics such as CPU utilization, and so on. Another feature of PeerTrust that limits its applicability to distributed trust models is that its credential discovery cannot retrieve certificates where the signer is unknown. This limits the proof construction to top-down fashion (from root certificate to end of a chain), which can be inefficient in multi-path delegations where there are many intermediate delegees.

MyProxy [44] is a credential repository service that allows storing and retrieving user credentials. This is utilized by other Grid systems, such as the work on the Grid trust negotiation with PeerTrust, as an additional credential store. In our framework, we do not assume such a repository, and store credentials in a DHT composed of participant organizations. But in the future, MyProxy can be an efficient alternative to store and access certificates, provided that this repository is maintained properly, and the security considerations related to host compromise, as mentioned by its authors, are handled, which otherwise makes the repository a single point of failure. In light of such considerations, we choose the federation credential storage to be fully distributed, but allow for the possibly to leverage centralized solutions in the future.

Grid Security Infrastructure (GSI) [24] allows proxy certificates, where a user's identity can be delegated to one or more of user's jobs that act on user's behalf, forming a trust path, however, these delegations are not actually between different principals and do not contain complex policies. GSI has been the default authentication and identity delegation standard for most of the authorization systems we mentioned in this section, and also systems like Condor-G [25] that

combine it with intra-domain resource management protocols such as file transfer.

Policy service for the Grid [62] studies resource allocation policy to allow individual institutions to manage access to their resources. This includes limiting the number of machines that can be used by a VO, specifying the allowed connections within the network, placing bandwidth limits and so on. Similarly, sfatables [11] allows expressing detailed allocation policies in the testbed federations context, and manages conflicting ones through prioritization. These resource allocation policy languages lack the ability to merge security policies and allocation policies, therefore resort to inflexible user attributes to classify users. They also grant local resources only, rather than third party resources.

Resource Management Architecture [22] involves a resource specification language (RSL) that allows allocation of Grid resources from multiple domains. This is a common objective with our federation framework's resource discovery and allocation, where a resource request should be fulfilled from possibly multiple organizations. RSL requests are also similar to our framework, which involves resource units and conditions on their properties, such as minimum memory requirements. A resource broker divides an RSL request to multiple requests to various resource managers, all of which are known by the broker through an information service. This system is limited in assumptions that all resource managers are known by the broker through an information service, and user authorization depends only on local policy at individual domains and domain's direct knowledge of the user. In our view, resource discovery is not independent of allocation policies, and it is not desirable and/or feasible for every user to discover any resource in a large scale federation.

Another Grid system uses Sub-contracts [18] to utilize peering relationships between organizations to discover and allocate resources. While this is very much like contract-based resource discovery and allocation (CODAL) of our federation framework, it does not define any language or authorization mechanisms to specify and enforce contracts. The described contract structure is basically a description of resource requests, or the final scheduled resources. It assumes that resource requests are submitted directly to a peer, which is responsible for allocating it or submitting to another peer. This does not require any indirect trust relations, hence trust management mechanisms, but suffers from every user request possibly having to go through peers. Also, there are no allocation policies defined, therefore resources are assumed to be granted to peers independent of any resource allocation agreements.

OurGrid [5] system improves the inter-organizational peering relationship with a reputation

mechanism, where peers who contribute more are favored more by their peers. However, OurGrid does not involve allocation policies, therefore an organization cannot differentiate among its peers through resource agreements. Also, resource requests are through peers and there is no trust management in OurGrid system, similar to Sub-contracts approach. We believe reputation is an important mechanism that can be an input in in-person resource sharing agreements, therefore can shape allocation policies, but is not enough to replace them.

A different branch of research that explores Grid resource allocation problem is the Grid economy, which applies market based mechanisms to arbitrate Grid resources among its participants [1, 7, 19, 65]. These mechanisms involve auctions that maximize the utilization of resources, and use electronic transactions. Market mechanisms are generally seen as too complex to be practical in large scale federations [5].

2.7 Other Technologies

Various systems in the literature do not appear as part of any existing federations, but are related to our federation framework. These involve trust management systems (TM), contracting languages, and resource discovery systems.

PolicyMaker [16] is the first system that introduced the trust management concept, describing how a set of remote and local policies, called assertions, can be regarded as programs that interact and compute together into a policy decision. PolicyMaker is very expressive since it supports fully programmable credentials; however, the computational complexity of its compliance checker—that is, the trust path constructing algorithm—is too high to be practical in its original form [15]. Also, the cost of high programmability is that it is hard to write policies. Every application that uses PolicyMaker must develop its own way of creating the policy program.

KeyNote [14] is another TM system that is based on PolicyMaker, which introduces a specific language to encode policies. This makes it easier to write policies, but it cannot express policies such as delegation depth, or conjunction of attributes. Therefore, it trades-off expressiveness for ease of use.

Delegation Logic (DL) [36] is the first TM system to our knowledge that used a logic language to express policies. Declarative semantics makes writing policy programs easier and the program serves as a logical proof of its expected behavior. DL can express some common security primitives

such as delegation depth, threshold structures, delegation of attribute authority, etc. DL shares a common limitation with PolicyMaker and KeyNote, in that it does not involve credential discovery and assumes all credentials are present at the time of policy execution. DL is more expressive than KeyNote, but does not support programmability as in PolicyMaker.

Secure Dynamically Distributed Datalog (SD3) [32] is a distributed TM system, involving self-certification by credential discovery, which is also based on a declarative logic language. SD3 certificate retrieval depends on name resolution to an IP address and authoritative queries at the signer of a certificate. This can be limited in cases where a signer is not yet known during policy execution. Also, SD3 does not involve conditions in the certificates, or arbitrary programs required for supporting complex policies, such as allocation constraints.

Another area that is related to our federation framework is the contracting languages. Similar to contracts in our federation policy language (FPL), there are courteous logic programs (CLP) [28, 29, 54] in semantic web to express business rules in contracts. CLP involves logic based expression of business rules, which is in declarative nature similar to FPL contracts. These contracts involve policies such as how much refund for a product can be given depending on time and defect, lead time allowed before an order can be changed that depends on type of customer, product type, backlog at seller, etc. The policy allows specifying prioritization of conflicting rules, so that the appropriate rule will be executed if a caller satisfies both rules. Therefore, it is similar to sfatables that was mentioned in Section 2.6. We can also utilize this feature in our contracts in the future. On the other hand, CLP policies do not involve constructs that make it possible to express resource allocation policies, such as operation on compound data structures like lists, recursion capability, etc.

Similar to CLPs, SLAs in the context of Internet service providers (ISP) and web services [12, 56], can describe the QoS of services such as overall 99% availability of an email service and 99.9% from 8 AM to 5 PM. These are simple contracts that do not involve security policies, complex allocation policies or third party resources.

Resource discovery systems in literature, such as SWORD [2] and Rhizoma [68], operate within a single administrative domain and therefore lack interorganizational allocation policies, as well as any per user allocation policies. While Rhizoma allows specifying complex rules to constrain the required resource sets, SWORD allows only range queries. Therefore, the former has more expressive policies in specification of resources.

Another distributed allocation system is SHARP [26], which allocates resources from many administrative domains in a federation. It mediates resource exchange between domains with simple bartering policies such as number of machines exchanged. Similar to SHARP, our discovery system CODAL can perform multi domain resource discovery and allocation but support much more complex allocation policies than SHARP by using FPL elements in its algorithms.

2.8 Federation Framework

Our federation framework aims at realizing distributed trust model. All of the other trust models discussed in Section 2.1 are special cases of this model; they can be implemented as abstraction layers on top of a system supporting the distributed trust model. Moreover, the distributed trust model is inevitable in real life scenarios such as users delegating privileges without contacting administrators [40], or organizations establishing in-person contractual agreements with small number of well connected but trusted organizations. Therefore, we should support the distributed trust model, and not only security policies, but also resource allocation policies. None of the existing federations or technologies involve multi-path delegations with complex policies. Therefore, we need a different approach to realize the federation model we envision in this thesis.

Figure 2.10 shows a rough comparison of some of the trust mechanisms, and our approach. Points on the graph do not represent exact numeric points, but rather try depict the usability/expressiveness trade off that we encounter in various systems.

One family of access control technologies is the federated identity management systems (FIM), as discussed before, such as SAML, OpenID and OAuth. Such systems perform authentication and authorization with CoT models, and policies are simple attribute-based policies. Another family of technologies is the trust management systems, that can realize more complex trust models, such as PolicyMaker, SD3, ABAC, etc as discussed before. Trust management systems can express more complex policies; on the other hand, they are harder to use since they require writing policy programs, and can have high computational complexity.

PolicyMaker is very expressive since it supports fully programmable credentials; however, it is hard to express policies and computational complexity of its compliance checker is too high to be practical. Other TMs sacrificed expressiveness for ease of use, for example, DL introduced using logic primitives as policies, which makes it easier to write policies in comparison, but policies are

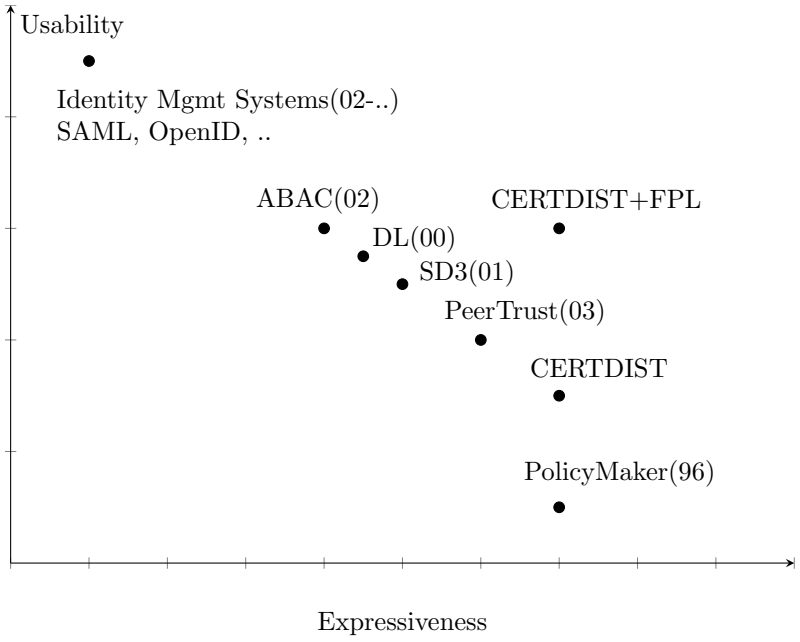


Figure 2.10: Comparison of access control systems

restricted in certain ways. A later work by same authors, ABAC, was constrained to attribute based policies, which cannot express policies like delegation depth, but it can be easier to write attribute policies. SD3, on the other hand, can express delegation depth, is based on a logic language and can discover credentials unlike DL and PolicyMaker. PeerTrust is similarly a logic based language, can also support conditional credentials, and performs trust negotiation.

Compared to those systems, CERTDIST has two distinguishing features. First, it has a more powerful certificate retrieval, finding certificates from a DHT even if the signer is not known. This allows distributed proofs of complex trust models, that is, even when certificates are not known locally or by subject.

Second, CERTDIST allows fully programmable certificates, making it as expressive as PolicyMaker. On the other hand, CERTDIST is not very easy to use, since writing a checker and programming the credentials can require expertise in logic language. Our federation policy language(FPL) defines the federation-specific abstractions that are implemented by CERTDIST and presented to system administrators to use as high-level constructs. As a result, we achieve both expressiveness and usability, first by CERTDIST, which is used by security architects to create new FPL elements, and second by FPL, which is used by system administrators and resource

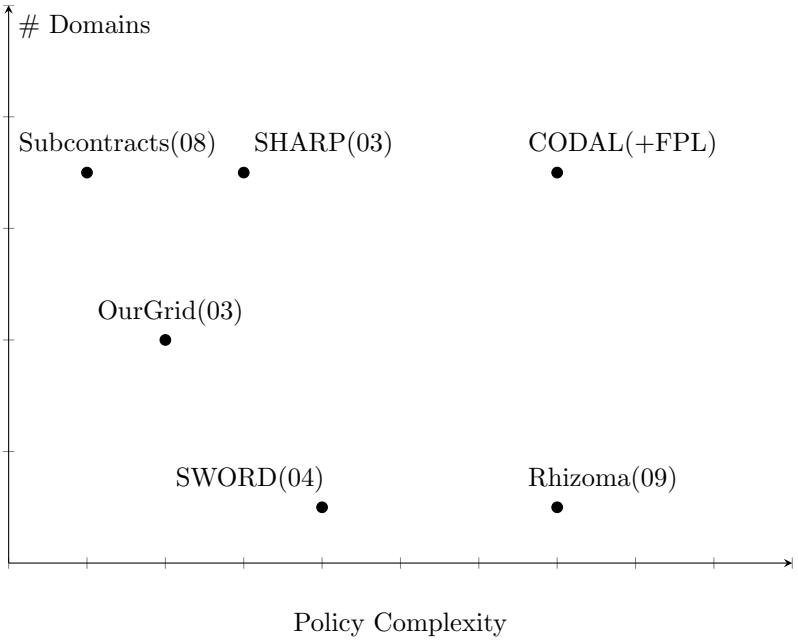


Figure 2.11: Comparison of distributed resource discovery and allocation systems

owners to easily express policies.

Figure 2.11 shows a comparison of previous work in resource discovery and allocation systems. Again, the points in the graph are not exact data points, but relative comparisons.

Since SWORD and Rhizoma are built for resource discovery within a single administrative domain, they reside lower in the graph. Rhizoma can support complex policies while choosing the best fit for a request, such as utility values, applying group constraints, maximum network path length, etc.

The sub-contracts work can support allocation among organizations indirectly knowing each other, therefore scales to many administrative domains; however, there is no allocation policy in this system, therefore it resides in the upper left corner of the graph. OurGrid can allocate resources through only one-hop among indirectly known domains, but regulates allocation amount depending on reputation based classification of its peers.

SHARP is able to allocate from many domains though arbitrarily long trust paths. However, the bartering type allocation policies do not satisfy the complex allocation policy needs of organizations.

Our contract based resource discovery and allocation system, CODAL, uses FPL (hence CERT-

DIST) in its lower layer to be able to discover many organizations where resources could reside, process complex allocation policies issued by those organizations in a delegation path, and authorize allocations.

Chapter 3

Federation Framework

This chapter describes the primary contribution of our thesis: the synthesis of trust management, policy language, and resource discovery mechanisms into a general and flexible federation framework. The individual pieces of the federation framework, CERTDIST, FPL, and CODAL mechanisms, are explained in more detail in later chapters.

The general requirements that the federation architecture must meet are expressiveness, usability and extensibility. It should be possible to express security policies with complex trust relationships, such as ones seen in a distributed trust model. The system should be able to efficiently construct proofs of access control decisions using distributed credential storage and collection. Despite the complexity, it has to be easy for system administrators to write policies using simple constructs, the implementation of which is hidden underneath. Similarly, resource allocation policies should be easily expressed, where policies can involve various computational resource types and constraints. These should easily be extended to resources and constraints in other domains as well. Moreover, allocation policy must allow expressing third-party resource delegations. Security and allocation policies should be able to be expressed in the same framework, where they can be used within one policy statement. Finally, the federation architecture must require minimal effort from the user, automatically discovering federation wide resources, authorizing and allocating them in a seamless way.

Federation Framework is designed to meet those requirements. Figure 3.1 shows the general layout of the federation framework. CERTDIST's distinguishing features, such as enhanced certificate retrieval functionality and conditional credentials, are utilized by the Federation Policy

Language (FPL) to develop federation specific abstractions. These abstractions can be categorized as *security policies*, *allocation policies*, and, finally, the *contracts*, which can use both types of policies. The Contract-Based Discovery and Allocation (CODAL) system utilizes FPL elements to perform federation-specific tasks, such as discovering new organizations in federation, collecting necessary credentials for authorization, and applying allocation policies in its constraint logic programming, which maps user requests to available resources.

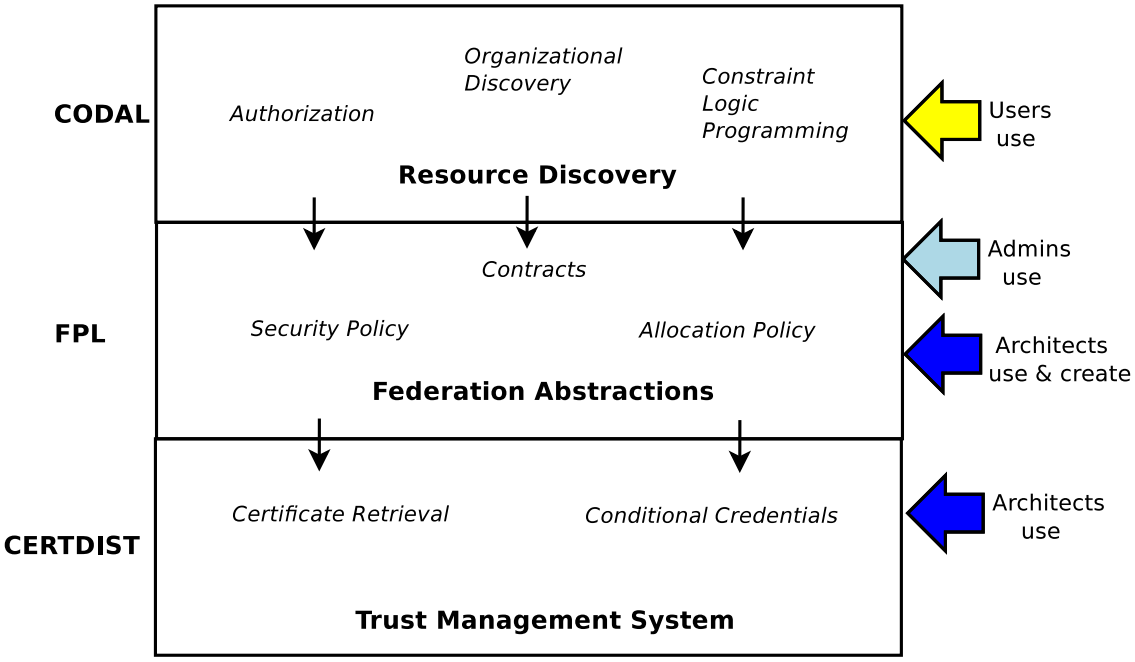


Figure 3.1: High-level view of federation framework that combines a trust management system, policy language, and resource discovery system.

3.1 System Actors

Different layers of our federation framework are used by different groups of people in a federation. Here we list these actors and the tasks they perform in the system, which generally differ in their level of complexity and ease of use.

3.1.1 Security Architects

Security architects extend FPL with primitives that serve as building blocks of higher level policies. Such primitives are generally policies that are general to the federation. For example, in order to join a federation, each participant organization should adopt a common chosen authentication policy. An example to such a policy would be one that is based on a hierarchical trust model, which would require multiple certificates in its proof. This policy can be represented with a primitive called *authenticate()*, that would be implemented by using CERTDIST. Creating such primitives in an efficient and secure way requires expertise in logic programming, and understanding of distributed trust management that involves collecting and using credentials, therefore digital certificates. Therefore, architects deal with most of the complexity in our federation framework.

As federation evolves, architects continue to figure out common policies required by federation participants, and extend FPL with language elements that will allow expressing and proving such policies. Similar to authentication, access control for the common API supported by most organizations in a federation can be represented by authorization primitives written for specific operations. In addition to security policies, architects also extend FPL with primitives that represent allocation policies. This requires not only determining resource types, but also available constraints that can be placed on them. The implementation of such constraints will again be much more involved, using functions like tapping into system resources, figuring out dynamic metrics such as requester's capacity, and gathering and using system state information. Primitives can be used by policy makers in individual organizations in a federation, avoiding most of their underlying complexity. FPL contains elements in the context of testbed and cloud federations, as will be discussed in Chapter 5, but is flexible and can be extended for other contexts as well.

3.1.2 System Administrators

System admins determine the organization-specific policies at each organization, possibly reflecting the allocation wishes of the resource owners. Admins use FPL to impose these policies at their organization. The primary FPL construct admins use is a *contract*, which arbitrates an organization's resources among the other organizations in a federation. It is up to the admins to decide with whom to share resources, and then to negotiate in various out of band ways about the conditions of a contract. An admin then writes the contract that will fulfill his/her end of the agreement, expressing the conditions under which the local resources should be granted. Note

that a contract does not always have to be reciprocal to another: it can be a one-way donation of resources to others. Contracts combine both security and allocation policy primitives, therefore admins can express not only what type and how much of resources to be granted, but also to which principals, roles, or attribute owners over arbitrary trust models. Once a contract is decided, it is placed in the organization's active contracts repository to allow or deny incoming resource requests. Admins can use security policies directly as well, to express special access control restrictions on any chosen API operations or custom services.

3.1.3 Users

Users of a federation are consumers of the federation resources, such as a researcher in GENI who wants computational resources to run an experiment, a student who wants to view academic documents in the InCommon federation, or Internet users using an e-commerce application. Users interact with the topmost layer of the federation framework, CODAL, that discovers, authorizes and acquires resources for the user. Each user's request is represented with a resource specification (RSPEC) that has to be fulfilled from one or more organizations in federation. Users are hidden from any inter-organizational allocation agreements, and they are not concerned with the organizations from which the resources will be obtained, as long as the RSPEC is fulfilled. Users do not have to write any security or allocation policies; however, they need to construct the RSPEC, which generally involves resource descriptions involving type and number of resources, unit and group constraints on them. Such constraints can be specified by the user programmatically or using various graphical user interface (GUI) methods. The resulting resource and constraints are translated into RSPEC in Prolog format. While the format of RPSEC stays the same, the resource types and user constraint primitives can be extended by security architects, as needed. The RSPEC will be processed by CODAL software that runs the user constraints and also contract policies to arrive at resource allocation decisions. This software runs the primitives in a generic way, therefore remains static as FPL is extended.

3.2 Trust Management

The lowest layer of the framework is our trust management system, CERTDIST. This is the piece that performs distributed credential storage, discovery and proof construction for access control.

As part of the requirements of our architecture, it is able to implement complex trust relationships in a federation, by discovering credentials in an efficient way and even when the signers are not known. CERTDIST allows expressing policies that require large numbers of certificates through recursive logic rules. In addition, it allows programmability of the credentials with its expressive logic-based language. Therefore, two main abstractions that it provides are about credential discovery and the programmability of these credentials.

Figure 3.2 shows the two distinguishing features of CERTDIST, certificate retrieval (credential discovery) and conditional credentials. Certificate retrieval uses a distributed has table (DHT), particularly Kademlia [41], as part of its processing, different from other distributed trust management systems such as SD3 and ABAC, or any other trust management systems that do not support certificate retrievals, such as PolicyMaker and Keynote. This allows expressing more complex policies and more robust proving capability, which will be discussed in greater detail in Chapter 4. In addition, conditional credentials contain Prolog programs, specifically SWI Prolog [64], in addition to the local policies. This is a powerful language used to express various policies using string processing, datetime operations, arithmetic operators and compound data structures. This is unlike other trust management systems that allow only attributes or a limited form of assertions in their credentials, but more like the PolicyMaker system, which allows programmable credentials. Some other CERTDIST components are certificate storage, event subsystem, and cryptography extensions for Prolog.

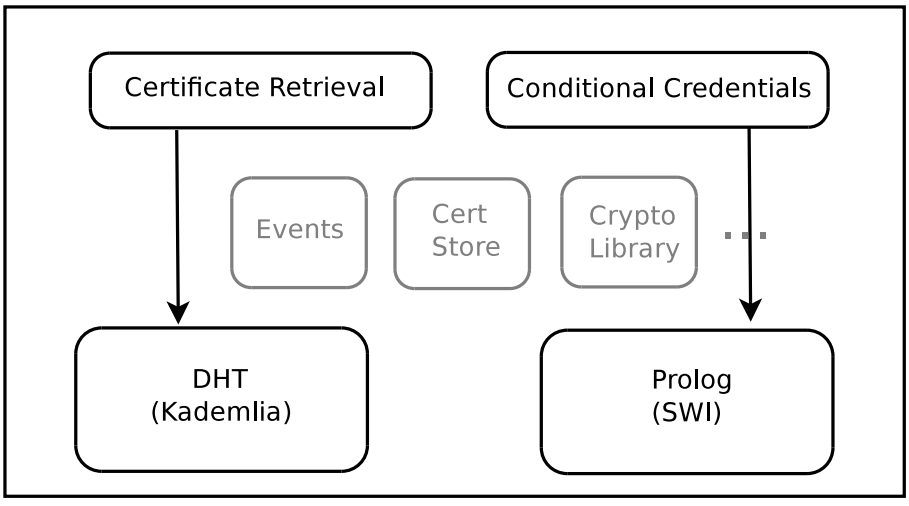


Figure 3.2: CERTDIST trust management system's features, certificate retrieval and conditional credentials, are realized using DHT and the Prolog language, respectively.

3.3 Federation Abstractions

Federation abstractions are defined in FPL, where the security and allocation policy elements are created by architects and used by organizational admins in creating contracts. As mentioned before, FPL should provide organizational admins with simple abstractions which possess all the expressive power of a complex trust management system, and combine it with resource allocation policies. The primary categories of abstractions that need to be supported are authentication and authorization policies, resource types and constraints, third party resources, and constructs that combine security and allocation policies.

Figure 3.3 shows a high-level view of the FPL components. This language is extensible by new elements for other federation contexts or as new security and allocation policies are required while the federation evolves. Here we explain the connection of FPL to CERTDIST and how language elements are implemented using CERTDIST, also with some policy examples. We also show how our contracts differ from previous systems that use contracts.

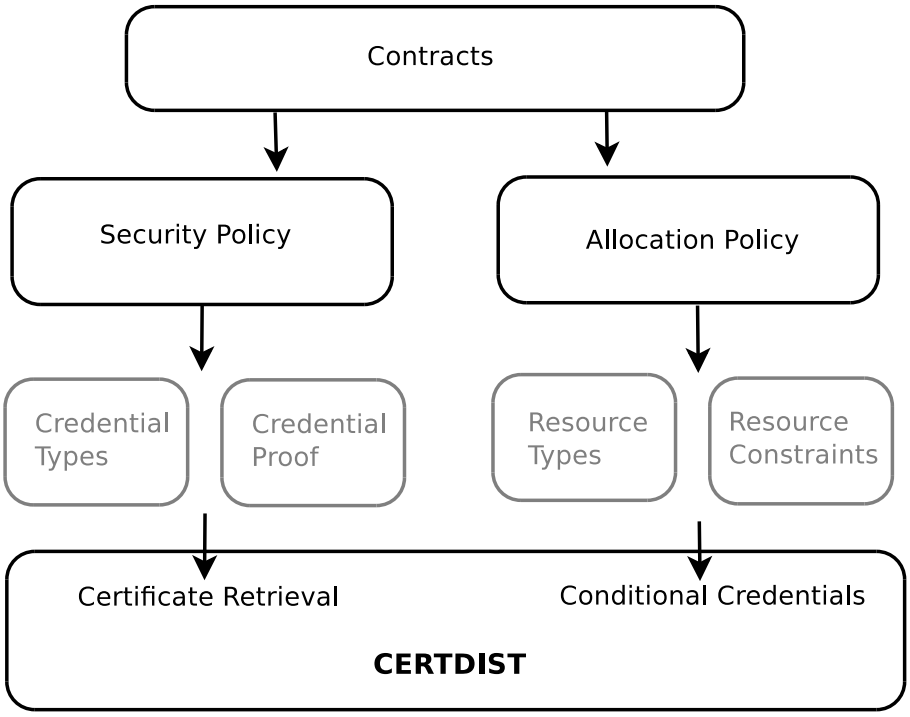


Figure 3.3: FPL federation abstractions that express security and allocation policies, created using CERTDIST as its lower layer. Contracts can combine both types of policies to express complex policies in federations.

3.3.1 Security Policy

Security policy in general determines which principals are allowed to perform various operations in a system. In systems with simple trust models, mechanisms such as access control lists can be used to map user identities to operations, such as in a web portal. Similarly, roles can be used to determine the set of principals who can perform certain operations, as in InCommon, which maps attributes like "student" or "faculty" to permission to see different documents. On the other hand, complex trust model systems have security policies requiring many digital certificates signed by different authorities and having complex dependencies. DNSSEC is one example policy where a set of certificates is required for proving that a name maps to specific IP addresses. In federations, such policies can be placed for individual API operations, naming of principals, creation of system wide roles, and so on.

FPL security policies are a library of known roles, authentication, and authorization primitives, each defined using the underlying CERTDIST, which collects and interprets digital certificates to construct a security proof. Security architects implement these primitives, by defining the certificates required to construct a proof of a particular policy. Such primitives are provided to the policy writers, such as organizational admins, as the high-level security abstractions.

An example of complex federation policy is one proposed security policy for GENI. It consists of hierarchical human-readable names (HRN), such as plc.eu.inria, to name objects—that is, organizations, users, machines, and slices. HRNs reflect the hierarchy of organizations within geographical locations and the users and resources under them. A slice basically stands for an organization-specific role, such as plc.eu.inria.overlaysec, such that whoever belongs to this slice can use it to allocate a set of resources across the federation. HRNs can be created under a namespace only by objects that have a "register" privilege on that namespace; plc implicitly has this privilege, being the root, and hence can create plc.eu, but plc.eu has to obtain the credentials from plc first to create plc.eu.inria. An object can delegate its privilege if it has a delegate bit for this privilege. Objects that have register privilege on a namespace can create slices.

Consider the slice membership policy, with an example such as when a user named plc.jp.keio.matt wants to use slice plc.eu.inria.overlaysec. We represent a certificate with $\{S\}_{PrX}$, where the statement S is signed using private key PrX, the public key of which is PbX. The following set of digital certificates proves Matt's public key PbM is allowed to perform this operation. PbP is plc's public key, known by everybody as the root.

(1) {PbE *has hrn* plc.eu}$_{PrP}$

(2) {plc.eu *can register at* plc.eu *with delegation* 1}$_{PrP}$

(3) {PbI *has hrn* plc.eu.inria}$_{PrE}$

(4) {plc.eu.inria *can register at* plc.eu.inria *with delegation* 1}$_{PrE}$

(5) {PbT *has hrn* plc.eu.inria.thierry}$_{PrI}$

(6) {plc.eu.inria.thierry *has slice* plc.eu.inria.overlaysec *with delegation* 1}$_{PrI}$

(7) {plc.jp.keio.matt *has slice* plc.eu.inria.overlaysec *with delegation* 0}$_{PrT}$

(8) {PbJ *has hrn* plc.jp}$_{PrP}$

(9) {PbK *has hrn* plc.jp.keio}$_{PrJ}$

(10) {PbM *has hrn* plc.jp.keio.matt}$_{PrK}$

The certificate (1) binds a public key to HRN plc.eu, and (2) states that it can create new objects under this namespace. (3) and (4) are required to prove the same thing for plc.eu.inria. (4) and (5) designate the user Thierry as a part of the slice overlaysec. Since this credential has the delegate bit, Thierry can join another user, Matt, to this slice using certificate (7). Finally, certificates (8), (9) and (10) authenticate *PbM* as plc.jp.keio.matt. The set of certificates that can prove a particular policy is not unique. For example, Inria may have directly given the slice to Matt, or it may be given through more than one delegations. Therefore, the implementation of the policy is not bound to a fixed number or set of certificates, but is able to define and execute a logical proof given a set of input certificates, and produces an outcome. In our example, the implementation would involve checking certificate fields such as whether the right delegate bit is set or not, and also whether the privileges are given at required namespaces or not, etc.

Security architects can implement such a policy using CERTDIST to come up with FPL abstractions such as, *authenticate(User, PublicKey)* or *has_privilege(User, Cred, NameSpace)*. These abstractions are implemented as functions that evaluate to true or false by collecting and verifying certificates. The definition of the abstractions can use each other, such as authentication of an object requiring first, the creator object to prove register privilege. New abstractions can be created from existing ones, such as *has_slice(User, SliceName)* or other types of security policies, such as access to API operations.

In order to define the abstractions, security architects first determine the type of certificates

in the system, such as identity, privilege and slice. Second, depending on the semantics and fields of credentials, federation-specific callback functions can be written by architects to ease DHT certificate retrievals. These two are depicted by the credential types and proof boxes in Figure 3.3, which are described in more detail in Chapters 4 and 5. Systems like ABAC [37] also provide higher-level abstractions to specify security policies, but FPL is more extensible and expressive since it directly interfaces to a powerful trust management system that allows new complex policy primitives to be developed.

3.3.2 Allocation Policy

Allocation policy determines how much of a set of resources is allocated to principals. In previous work for both the Grid computing and testbed arenas, allocation policies regulate access to network and computational resources. For example, sfatables [11] is a tool used in PlanetLab federation, which can express a variety of allocation policies. The allocation policy for Grid computing is studied at [62], leveraging existing policy frameworks such as IETF and DMTF. Resource specification language (RSL) [22], again in Grid context, can gather resource information from multiple domains, and allow users to specify resource units and conditions on their properties, such as minimum memory requirements. A different way of arbitrating resources in Grid is through market based mechanisms, as studied in various systems mentioned in Section 2.6. Our framework is unlike market mechanisms, but relies on humans writing the allocation policy, similar to other systems we mentioned.

Allocation policy involves defining resource types and the constraints placed on them. In previous work, the resource types consist of local computational resources such as CPU, memory and storage, which can be grouped into different virtual machine types. Constraints can be static, such as "maximum of 10 virtual machines" or "except the machines in an exclude-list". Constraints can also be dynamic, such as "10% of my current resources during day time, 20% at nights". To realize dynamic policies, an organization would collect the current system information and execute the policy with such information as input. FPL also provides primitives to define third-party resources that can be used as a new meta-type in policies, and also new primitives for dynamic resource constraints, that are unique to our federation setting with indirect trust relationships. These are represented by resource types and constraints boxes in Figure 3.3.

FPL allows delegating third-party resources to others, without them having to negotiate or

agree on any resource exchange with the third parties. Third-party resources are implemented by using conditional credentials underneath. A set of such credentials proves the existence of a third-party resource, and therefore can be used in a policy statement. An FPL primitive that defines such a set of credentials is used by an organizational admin to easily express third-party resources in a contract, such as "resources I have at plc.jp".

Here are some sample FPL allocation policies in testbed federation context. Policy P1 says plc.eu gives to plc.jp a maximum of 100 machines during the day and a maximum of 200 machines during nights. P1 will be obtained by plc.jp in a signed certificate. The mechanism (discovery) through which this credential is obtained is not crucial for this example, and will be explained later. plc.jp can later use P1 to allocate resources, where overall successive allocations have to conform to P1's policy. A later policy, P2, by plc.jp can say that plc.jp gives 20% of the capacity it obtained from European organizations (plc.eu.*) to the plc.jp.keio organization. As a result of P2, plc.jp will not only grant to plc.jp.keio a credential containing P2's allocation constraints, but also pass the credential for P1, and any other credentials it obtained from plc.eu.*, which allow plc.jp.keio to prove to them that it has the right to some portion of plc.jp's resources. Different primitives can express different granularity of third-party resources, such as single organization, or continental resource as in the example. Also note that third-party resource delegation can happen over arbitrarily long paths. While the admins just use these primitives, implementation details such as the credential collection and proving of a valid certificate chain are handled by CERTDIST.

Security architects also provide primitives for expressing constraints on the resources (local or third-party). As mentioned before, some examples are percent capacity limit, machine number limit and resource exemption in testbeds context. Some dynamic constraints are more challenging to implement in the face of third-party resources. For example, the capacity of local resources in an organization is simply a function of the available hardware at the organization. But when we talk about the capacity of the organization at a remote site, it depends both on remote organization's hardware, and also the policy that applies to the first organization. Since the allocation policy is a binary function that accepts or denies a particular resource request, it is not trivial to come up with a capacity value associated with the policy. In order to calculate capacity, that is, the maximum resource that could possibly allocated with it, a heuristic search algorithm is applied. A primitive that represents such an algorithm is provided by FPL, again hiding its implementation.

Another example to such a primitive is one that allows to tap into accounting data structures keeping track of current resource usage amount per credential. Such primitives can be building blocks for higher level allocation policies implemented by security architects, or directly used by admins who want to create their custom policy constructs.

3.3.3 Contracts

Contracts are a key abstraction in Federation Framework. Contracts allow organizations to easily make their resources available to others, which is a gating function to expand a federation. Contracts enable this by combining allocation and security policies in the same policy statement, to first express the resources to be granted, and second, the delegees of the resource. This is important since it can be impractical to write a contract per grantee, especially when there are many organizations in a federation. In that case, contracts can grant resources to groups of principals that are defined by security policies. For example, in testbed federation context, plc.jp can write a contract saying it delegates all resources it obtained from European organizations (plc.eu.*) to organizations in Japan (plc.jp.*), with a limitation of 40 virtual machines maximum at each organization. As a result, in order to be eligible for the contract, certain credentials need to be provided. Also, as new organizations in Japan emerge, they can have access to these resources automatically. Similarly, any other security policy provided in FPL can be used to restrict the recipients of the resources to certain organizations.

Contracts that are seen in previous work do not have this ability to express both types of policies. For example, CLPs discussed in Section 2.7 do not involve security policies, but only define the conditions of certain purchases or interactions in semantic web. These conditions are logic based rules, which is similar to our contracts, but do not express complex allocation policies, or third-party resources either. Similarly, SLAs in the context of Internet service providers (ISP) and web services [12,56], describe the QoS of services negotiated only between certain organization or services, and lack security policies.

3.4 Resource Discovery

The last layer of our architecture is contract-based resource discovery and allocation (CODAL). The resource discovery problem in a federation is different from previous systems that operate

within a single administrative domain. Here we explain the individual pieces of CODAL, which interface to FPL elements such as contracts, security, and allocation policies, as shown in Figure 3.4.

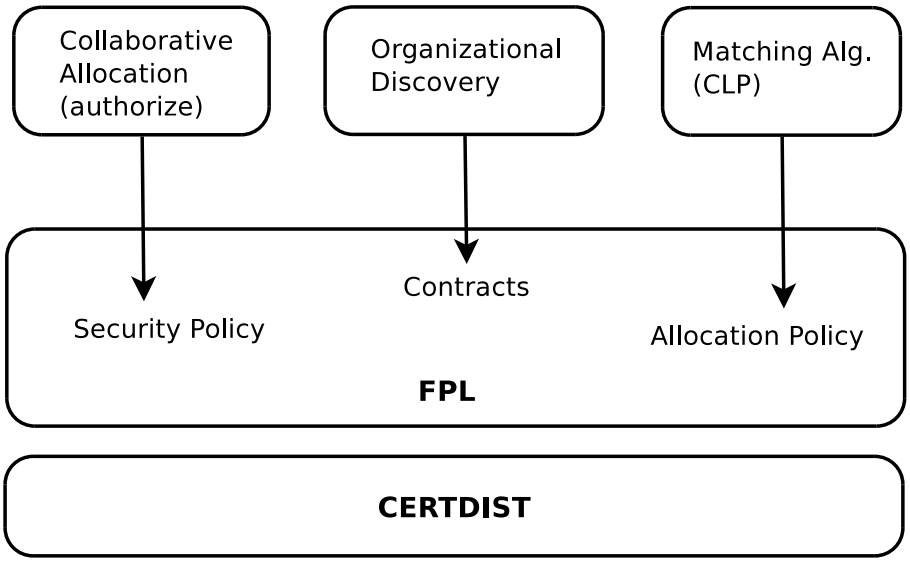


Figure 3.4: Resource discovery layer, CODAL, uses FPL contracts, security, and allocation policies to perform federation-wide resource discovery and allocation.

3.4.1 Organizational Discovery

A key problem in a federation is discovering new federation peers. This is different from systems like RSL, where organizations publish their resource information into a common information service. In a federation, organizations can have private agreements with others, and do not have to publish these globally. Therefore, similar to the acquisition of resources, visibility of them can be dictated by organizational policies.

CODAL discovers new federation peers, that is, organizations from which resources can be allocated. As mentioned before, contracts can express delegation of a multi-origin set of resources to a set of principals. CODAL uses FPL contracts already granted to the organization and periodically queries their granters to learn about possible third-party resource sources, together with the allocation policy restrictions applied on them. In the contract example of Section 3.3.3, plc.jp.jaist can query plc.jp and discover resources residing at plc.eu.inria, plc.eu.cambridge, so on, where a chain of conditional credentials are passed for each path so that plc.jp.jaist can allocate

from those sources later. Obtaining of digital credentials can be coupled with the discovery phase or be separate for efficiency reasons, which are discussed in more detail in Chapter 6.

3.4.2 Collaborative Allocation

CODAL performs collaborative allocation, meaning it fulfills user requests from one or more organizations. Even assuming the organizations are known, that is, after the organizational discovery of Section 3.4.1, it is challenging to fulfill a resource request from multiple administrative domains. The group constraints of the resource request need to be satisfied even though the individual pieces might reside in different peers. Also, necessary certificates need to be collected efficiently in order to allocate from many peers quickly.

Collaborative allocation performs a sequential querying of peers until the resource request is fulfilled. It passes the whole resource request and the constraints to each peer in sequence, with the restriction that the group constraints are "subset-aware", which is explained in more detail in Chapter 6. This restriction is instrumental in a peer's matching algorithm to arrive at a partial match of the resource request. Collaborative allocation uses FPL security policies in order to collect necessary certificates before each remote query. Such certificates can be needed for authentication at peer organizations or authorization of resource ownership using a chain of conditional credentials.

Collaborative allocation assumes there are enough peers in a federation that, when queried sequentially, will eventually satisfy all the units and constraints in a resource request. This may not always be true, or the requester may have been granted only a small amount of overall resources. In those cases, a more complex algorithm that optimizes the mapping of resource request to a global view of the resources can be more effective. Also, there could be group constraints that can only be satisfied in a transaction-like allocation scheme from multiple organizations. Parallel allocation or transactions can be a future work for the collaborative allocation.

3.4.3 Matching Algorithm

A resource request arriving at an organization is processed by the matching (or mediator) algorithm, which performs constraint logic programming (CLP) to map an RSPEC to available local resources of the organization. CODAL performs best-effort *partial matching* of an RSPEC, meaning it find only a subset of overall resource units requested, but still not violating any unit or group

constraints. CLP applies both the user-specified constraints in RSPEC, as well as allocation policy restrictions placed in contracts, in order to determine the properties of the resources that can be allocated. As a result, CODAL's matching algorithm executes the FPL abstractions written inside contracts in order to determine which resources can be allocated and in what amount.

Chapter 4

Trust Mechanism: CERTDIST

We introduce our trust mechanism CERTDIST, which is in the family of *trust management* (TM) systems such as PolicyMaker and Keynote. CERTDIST has a logic-based language for specification of policies, similar to TM systems like SD3 [32] and PeerTrust [42] but can support more expressive policies. CERTDIST has a distributed hash table (DHT) retrieval of certificates, that allows distributed compliance checking of a wider range of policies. With these features, CERTDIST is powerful enough to perform access control in complex trust models and serves as an underlying mechanism to implement the FPL (Federation Policy Language). In this chapter we first explain some trust management terminology and then details of CERTDIST design components. The processing of our system is explained with an example using the Prolog language; therefore, we give a short introduction to Prolog beforehand.

4.1 What is Trust Management?

Before explaining our system in more detail, we describe how trust management differs from other security mechanism approaches. A trust management (TM) system handles complex authorization decisions by possibly using policies from many remote sources in the form of certificates and also local policies. When a remote request arrives at an application, the application passes the request string and any relevant certificates to the TM system. The TM decides on authorization depending on the specific principal requesting the action and the obtained certificates. During the decision process, the TM may retrieve more certificates if needed, running a complex program to check

if they together meet the conditions specified in the local policy. This is different from identity management systems, in which the decision process is as simple as checking for the existence of some string attributes in user certificates.

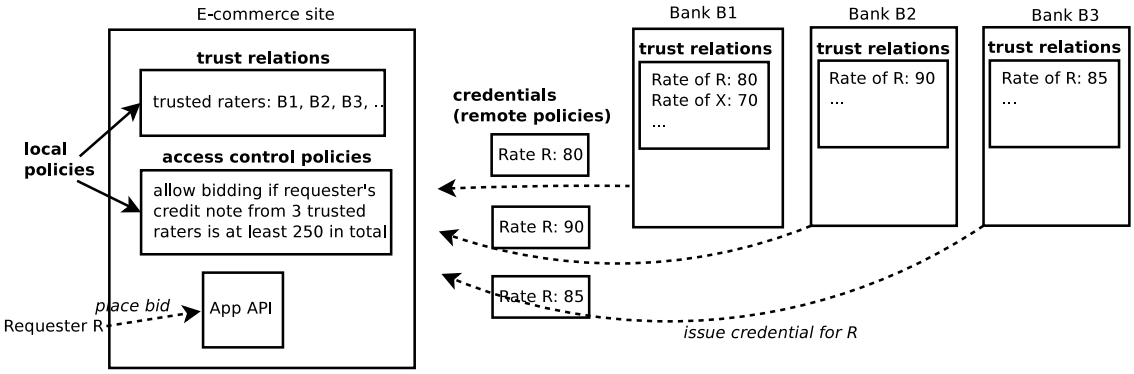


Figure 4.1: An e-commerce application, where a requester wants to place a bid on an item. The TM approach to categorizing policies: Local policy is made of trust relations and application policy. Remote policies are credentials carrying trust relations.

Figure 4.1 exemplifies an application that can use TM and describes terminology for TM. There is an e-commerce site which can accept bids for items. The *access control policy* for the specific type of bid is such that users who request to place bids need to have a total credit rate of 250 from three of the trusted rater organizations. The *local policy* of a site is composed of the access control policies and the *trust relations* of the site. Examples of the latter are the organizations trusted to rate the users, the site's local key, users, and so on. A subset of the trust relations at an organization can be signed in digital certificates, called *credentials*, which becomes a *remote policy* at another site. The trust relations that can go into a credential are *public trust relations*, such as ones in the banks, and others are *private trust relations*, such as one at the e-commerce site. Access control policies do not go in a credential, because by definition they are policies for a local operation specific to the organization. In the figure, credentials from three banks, B1, B2, and B3, are passed to the e-commerce site. We do not depict here how the credentials are collected, how the TM is invoked, or its processing of the local and remote policy, and so on, which will be discussed more later in this section.

There are three main pieces in a trust management system: policy expression language, compliance checker and certificate retrieval (credential discovery). First, a policy language can support policies with varying levels of complexity, such as attributes, attribute-based delegation, param-

eterized attributes, and fully programmable policies. The RT family of languages [37] captures most of these policy types with a sequence of languages of increasing complexity. The drawback of expressive policy languages is that they can be hard to use. For example, the first TM system, PolicyMaker allows fully programmable policies, but it is hard to write policies and the computational complexity of its compliance checker is NP-hard in its original form, and polynomial when restricted in certain ways. TM systems like SD3 and PeerTrust utilize declarative logic programming as their policy language, based on Datalog and Minerva Prolog, respectively, which increases readability and correctness of policies. RT is also a declarative family of languages, but diverge from logic programming for the sake of simplicity, although it is based on Datalog in its core.

The second main piece of a TM is compliance checking, which is defined as the process of reaching a yes or no result to a request, given a set of certificates and the local policies [15]. From the trust models' perspective, which was explained in Section 2.1, it is equivalent to discovering a trust path between two entities. Therefore, compliance checking translates into finding one or more credential chains that show a delegation of rights, starting from trusted owners to the subject of the request. If the required certificates are not locally present, a decision cannot be reached until they are remotely discovered; therefore, compliance checking is closely related to the credential discovery problem. Previous work on credential chain discovery [38] introduces a compliance checker for the RT_0 language that can work both with local or remote credentials.

One issue in compliance checking is the direction of the search for certificate chains in the certificate graph. A TM can perform top-down, bottom-up (backward, forward in [38], respectively) or bi-directional search of the certificates, meaning, starting from the root of the certificate chain to the subject, or the other way, or both. Depending on the set of certificates and the properties of the policy such as delegation fan-out, it can be more efficient to look for subjects with particular certificates, or signers that have signed particular certificates, or both, while trying to construct a chain. Another consideration is the graph traversal method, such as depth-first and/or breadth-first traversal of a credential graph, where either can be more efficient depending on the policy parameters, such as the length of delegations. Finally, application-specific constraints and heuristics as part of the policies can be utilized by a compliance checker, searching some paths before the others or eliminating some, while searching for credential chains. This relates to the expressiveness of the policies, such that, expressive policies can enable more efficient heuristics. In other words, although compliance checker is application independent, the policies within individual credentials

can pro-actively help graph search to skip some paths, or favor others.

The third main feature of a TM system is credential discovery, that is, retrieving certificates remotely to be able to prove policies. Some early TM systems do not support this feature, and assume credentials are stored locally, such as PolicyMaker and Keynote. More recent TM systems solve this problem by storing/retrieving credentials at/from their signers or subjects. Storing certificates and responding to many requests can be costly and require high availability for these parties. Therefore, RT introduces a type system that dictates that a type of credential will always be stored either at signer or subjects, to ensure certificates are always available. Grid systems utilize central repositories such as MyProxy [44] to increase availability of certificates, but there is no TM system that is based on centralized certificate storage.

4.2 CERTDIST Design

In this section we explain CERTDIST design in terms of three main TM design considerations: policy language, compliance checker and credential discovery. CERTDIST policy language is based on SWI-Prolog [64], similar to SD3 and PeerTrust that are based on Datalog and Minerva Prolog, respectively, but allows pure Prolog policies, unlike those two that impose higher level syntax. CERTDIST extends Prolog with cryptography and certificate discovery primitives, and utilizes built-in function and data structures, and as a result, can express policies ranging from parameterized attributes to programmable credentials, and resource allocation policies. CERTDIST compliance checker is based on the Prolog interpreter, has the ability to apply any of the compliance checking methods mentioned in Section 4.1, and can verify policies that other TM systems cannot. CERTDIST has a DHT credential discovery feature, that allows certificate retrieval even when signers are not known, allowing credential chain discovery in any direction, and even when signer/subject servers are unavailable to answer requests.

We first explain compliance checking in CERTDIST in detail with comparison to other TM systems, and also show the general flow of how access control requests are sent and received, and the components involved during this process. Next, we specifically describe the certificate retrieval steps, which is a crucial part of the overall flow, with a comparison to previous TM systems, emphasizing CERTDIST's distinguishing features. Finally we explain the policy language in CERTDIST, and also give a brief introduction to Prolog.

4.2.1 Compliance Checking

CERTDIST combines the capabilities of compliance checkers in RT_0 and SD3 TM systems, hence can search a credential graph with any direction or traversal method, and can use application specific policies in this search. To better explain certificate graphs, credential chain discovery, and considerations associated with it, consider the following DNSSEC example, similar to one used in the SD3 paper. We change and simplify the policy for the sake of our example, into "name N is a valid DNS name", rather than "name N is bound to IP address A". So, to prove such a policy, the compliance checker has to find a chain of certificates with *ns* attribute, and a final certificate with *addr* attribute. More specifically, the chain starts with a certificate signed by the root key. There can be zero or more certificates that bind the next key to *ns* attribute, and finally one certificate that binds the name N to *addr* attribute. Note that the set of certificates in this credential graph would be pretty large, if we happen to mimic all branches in today's DNS system. Despite its size, let's assume for now that this graph is stored locally and available to the TM system. When an access control query asks if N is a valid name or not, the compliance checker should ideally be able to traverse the graph in bottom-up direction (forward in RT terminology) in a small number of steps, to see if there is a path from N to the root, or not. SD3 relies on an existing key to start a search, therefore, is limited to top-down direction, which is inefficient for this policy. RT_0's compliance checker can perform search in both directions; therefore in this graph, it can initiate search in both directions, realize there is a large fan-out in top-down direction, and complete chain discovery using bottom-up direction instead.

Although RT_0 is able to search the graph in both directions, it has a limited ability to incorporate application specific constraints in the search. Note that, the credential graph can contain not only DNSSEC credentials, but ones related to other policies. In that case, name N can have a large fan-out in bottom-up direction, as well. Although DNSSEC policy dictates that the chain must end with an *addr* attribute credential, there is no way to express this in the RT_0 compliance checker. The checker will try to handle cases such as, a name server delegates *addr* attribute to anyone having *tmp* attribute. This is completely legitimate for RT_0's generic compliance checker, but does not reflect the application policy, DNSSEC in this case. RT_0 checker has a limited application specific customization of the search process, such that, some attributes can be configured to invoke search in specific directions. For example, since only bottom-up direction makes sense for this policy, *addr* attribute could be configured to never check top-down direction. Similarly, there

can be customizations as to the traversal method, either depth-first or breadth-first traversal can be applied per role. However, such customizations do not address the general problem, since one attribute can be used in different applications and policies.

The complexity level of the individual policies can be effective in determining compliance checking efficiency. For example, if the credentials expressed parameterized attributes, such as *ns(princeton.edu)*, it would be possible to efficiently perform top-bottom search of credential graph. Because, the *ns* attribute would tell explicitly which part of the namespace the name server is responsible for, therefore, we could eliminate unrelated paths in the search. SD3's DNSSEC algorithm does this, by supporting parameterized attributes, and placing a constraint such that the parameter seen in a credential must be the suffix of the one following it in a chain of credentials. Note that, to achieve such an efficiency in compliance checking, a TM has to support complex policies, such as ones based on string processing, in order to do checks like if one parameter is a suffix of another. We can easily extend this example to lists and subset/superset relationships, and deduce that TM system should be able to support arithmetic, string, date-time based operators and compound data structures. RT_0 language does not support parameterized attributes, or string based constraints.

CERTDIST is similar to SD3 in that it can support complex constraints and heuristics. In addition, the expressiveness of Prolog gives it the ability to apply graph search methods used in RT_0. It can perform top-down/bottom-up/bi-directional search in a credential graph, both locally, or distributively thanks to DHT certificate retrieval. Although depth-first is the default method in answering queries in logic languages, CERTDIST can perform breadth-first traversal of a certificate graph as well, by utilizing Prolog data structures such as queues, or programming techniques such as iterative deepening. Therefore, we can say that CERTDIST combines merits of both systems, and can prove policies neither system can. For example, using CERTDIST we can decide to eliminate certain branches seen in top-down direction depending on some paths discovered during bottom-up direction. Let's assume we want to check if a user U is granted resources by organization O1. In bottom-up direction, we can check which institutions U is associated with. Then we can learn that U is part of a testbed T in Japan. Then, in top-down direction, we see that O1 grants its resources to some cloud organizations and some testbeds, some of which are in Asia. So, we first try to construct a delegation path from "testbeds in Asia" to T, assuming this is the most likely path we can find a credential chain. It is most likely because we can derive that

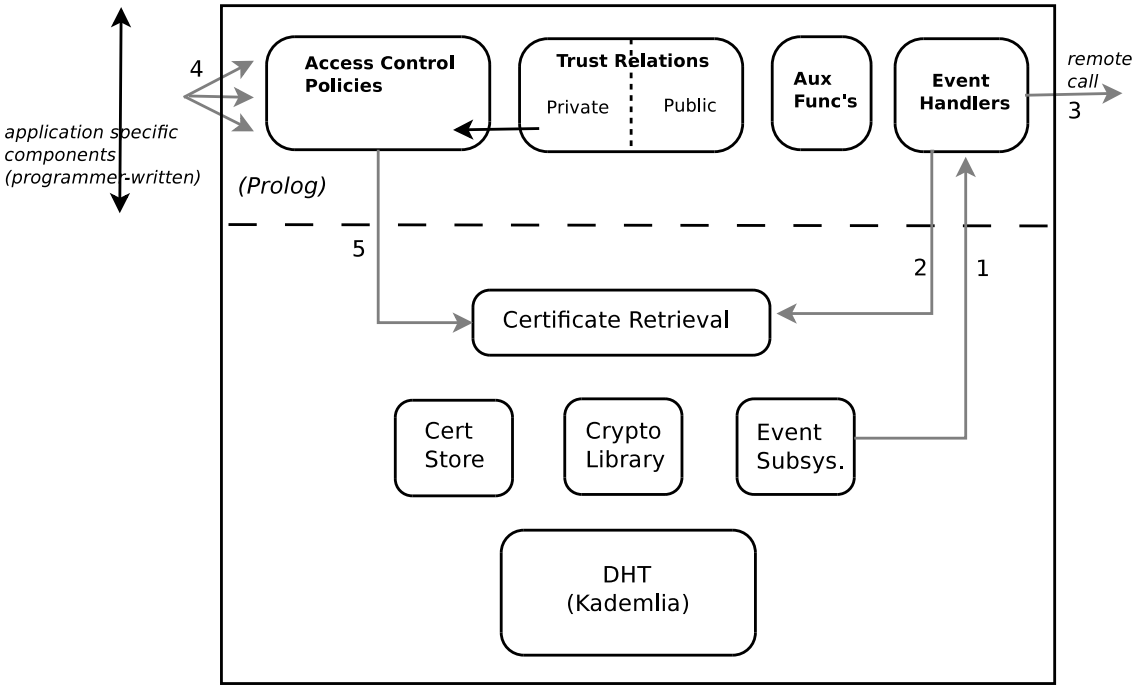


Figure 4.2: Application specific components: access control policies, trust relations, events handlers, and auxiliary functions. Certificate retrieval is used during compliance checking.

Japan is a country in Asia probably by looking up in a list or calculating proximity in other ways, and also resource exchange between testbeds is more likely rather than between commercial clouds and testbeds. As a result, we are able to find a chain from O1 to O2, and to T and finally to U, granting some resources to the user. In other TM systems, this kind of selective bi-directional compliance checking is not possible.

In CERTDIST and similar TMs like SD3, such heuristics and constraints are expressed by a complex local policy, which we often call the local access control policy that exists per application or per query type. In the DNS example, this would be the policy that dictates suffix relationship between two parameters, or expresses the requirement that the last certificate should be an *addr* attribute. Similarly, in the TM example of Figure 4.1, it is the policy that expresses total rating must be greater than 250. In other words, we can say that the local access control policy of a query describes the properties of one or more chains of certificates required to allow the query. In theory, a local access control policy does not have to be more complex than the credentials collected remotely, but generally it is, and as a convention we also call them application specific checkers.

Figure 4.2 shows the overall components of CERTDIST. The programmer writes access control policies, trust relationships, event handlers, and some auxiliary functions in Prolog. We note that programmers in CERTDIST context correspond to the architects in our federation framework. Since CERTDIST can be utilized for other purposes, we use programmers as a general class of people who design and/or know and implement an application's security architecture. Any remote operation in an application that requires complex policy decisions will have a separate checker function, located at the access control policies. This checker function is dispatched upon a query made to the CERTDIST daemon. For simplicity, we can name the checker functions with the convention request_*name*. For instance, in the TM example of Section 4.1, the bid() function will have a checker function such as request_bid() that determines whether the bidding should be allowed or not. As depicted in the figure, access control policies can utilize private trust relations and also retrieve certificates during its execution.

The general processing of CERTDIST is as follows. At step 1, the CERTDIST event subsystem triggers an event that requires authenticating and authorizing at a remote CERTDIST daemon. Such an event could be a user trying to create an experiment at a tested/cloud that requires his or her student credentials and identity information. An event handler specific to the type of request is invoked, which first collects necessary certificates for the request at step 2. We explain the specifics of the certificate retrieval in the next section in detail. The event handler performs the remote request at step 3 by passing the certificates as a parameter. The event handler should be written by the application specific policy writers, just like the access control policies. Events are registered at the event subsystem so that upon a click at UI or form input, and so on, the appropriate handler is invoked.

The request arrives at a remote CERTDIST daemon at step 4, which is dispatched to an access control policy. The certificates that are passed by the query are stored at the certificate store, which is not shown in the figure. The request may need additional certificates in case they are not already passed by the callee. Such certificates can be retrieved at step 5 by invoking the certificate retrieval module. The access control policy can be programmed to remotely retrieve a small number of certificates or use only the locally stored certificates, and so on, since remote certificate retrievals can be overhead for the callee. If the result of the access control policy is to allow the request, then the semantics of it are performed by passing the request and parameters to the application-level modules responsible for the request.

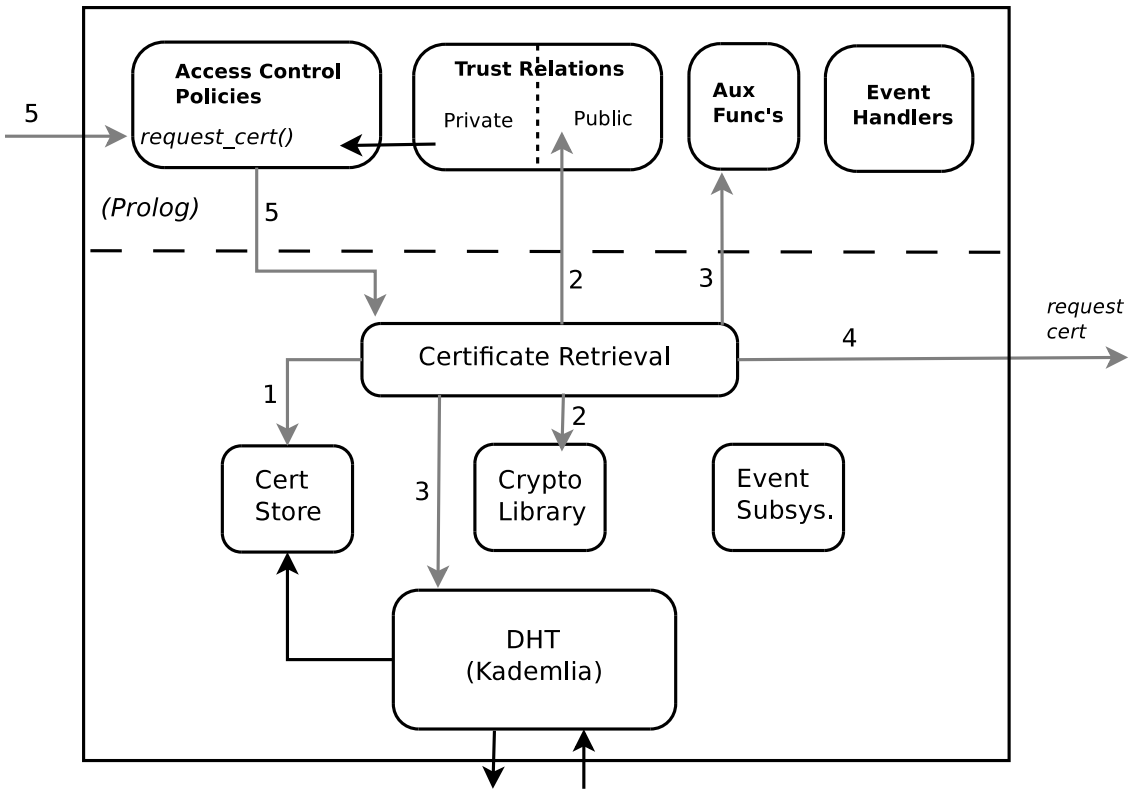


Figure 4.3: Certificate retrieval can use a local certificate store, digitally sign certificates, or perform DHT or authoritative queries to locate them.

4.2.2 Certificate Retrieval

As mentioned before, the certificate retrieval module can be invoked both by the event handler at the caller and the access control policy at the callee. One distinguishing feature of CERTDIST is that it can retrieve certificates not only directly from their signers but also from a DHT. This allows credential chain discovery in forward direction even when subject does not have the certificate, and similarly, in backward direction even when signer does not have the certificate, or the server of any party is down. As a result, DHT can increase efficiency, fault tolerance and feasibility of distributed compliance checking. We believe DHT certificate retrieval can be incorporated and useful in other TM systems as well, in decreasing individual certificate retrieval latency. However, it would not be very useful in their distributed compliance checking ability, since their checkers are limited even with local certificates, as discussed in Section 4.2.1. Next, we explain the processing of certificate retrieval logic below using an example certificate.

$$\{\text{plc.eu.inria.thierry } \textit{has slice} \text{ plc.eu.inria.overlaysec } \textit{with delegation } 1\}_{PrI}$$

A certificate can be found even though there is partial knowledge about it. A certificate is composed of signer and content fields and a type, which can be used as indices in the search. In the example above, the signer field is PbI, the content fields are (plc.eu.inria.thierry, plc.eu.inria.overlaysec, 1), and type is "has slice." Before a certificate retrieval, only some of these fields may be known. For example, a certificate query can be: retrieve all "has slice" certificates signed for plc.eu.inria.thierry. In such a query, neither the signer fields nor the delegation bit field of the certificate is known. Our system can retrieve certificates even if some fields are missing, but it requires that the type of certificate is always known.

The certificate retrieval logic is depicted in Figure 4.3. There are four ways a certificate can be retrieved, which are explained in a specific order. However, we do not restrict this order; it is possible to pass a parameter to certificate retrieval logic to specify which one of these ways should be used and in what order.

Locally from the Certificate Store

The locally saved certificates will be checked at step 1 in the figure to see if any of them matches the certificate description. This is the fastest way of retrieving a certificate since it does not require any remote queries or cryptographic signing operations. Any certificate in the store is verified before to have a valid signature and not have expired yet. The specific format in which the certificates are stored and the way they are searched are explained in the implementation in Chapter 7.

Locally from Local Policy

If step 1 fails, the local public trust relations are checked at step 2 in order to find a policy that can be used to sign the certificate locally. This entails the certificate type and content matching the certificate query and the signer being the local CERTDIST daemon. If the policy is found, the cryptography library is used to sign the certificate. The implementation of the cryptographic functionality will further be explained in the implementation in Chapter 7.

Once a new certificate is signed, it will be stored in local certificate store to avoid signing a new certificate if requested again later. In addition, the certificate will be stored at the Kademlia DHT. A certificate has to be public in order to be stored in the DHT. Also, a certificate should be

usable by other principals in the system, such as a DNSSEC IP address certificate. Such properties depend on the semantics of the certificate and hence, on the specific application. Therefore, the decision to store—and under what keys—should be specified by the programmer as an application-level primitive. Such a primitive can be implemented in CERTDISTs auxiliary functions module, as shown in the figure. Hence, at step 3, this function will be invoked with the certificate as input, which will return the list of DHT keys. Next, the certificate will be stored in the DHT with such keys.

When storing values in a DHT, a design decision is choosing how to specify keys for later lookup. The DHT keys for a certificate can be hash of a combination of its type and any of its signer and content fields. The certificate can be stored under several keys to allow it to be retrieved under different partial knowledges of fields. Each key will be stored at several peers in a DHT as part of regular processing of DHT systems.

For example, this certificate can be stored under the key "hash(*has slice*, plc.eu.inria.thierry, plc.eu.inria.overlaysec)", so that it can be retrieved without necessarily knowing the signer. DHT lookup would especially be useful to retrieve further certificates signed by Thierry, since authoritative queries to Inria will not result in any certificates.

Storing under different keys can make sense when there are multiple delegation paths to the subject, and different certificate fields are important for different policies. For example, an attribute certificate can give a user "Register" privilege at namespace "plc.eu.inria". This can be stored under the hash of username and privilege, and also username and namespace. One policy may need the user to exercise the register right, independent of namespace, in which case the first key would be used to retrieve the certificate. Note that, since credentials are delegable, it might have been signed by anyone having the credential with delegation rights, and there can be more than one such certificate. Similarly, another policy could require one of several rights at a namespace, in which case the second key would be relevant. In general, the benefits of multiple keys depend on the specifics of the policy that CERTDIST is implementing, and multiple keys can increase efficiency of DHT lookups if properly chosen.

Remotely from DHT

If the certificate is not found in a certificate store or local policy, then it will be looked up in the DHT. Similar to step 3 of certificate store operation, the DHT keys will be first looked up by

calling the auxiliary function; next, one or more DHT lookup operations will be issued using such keys.

Since CERTDIST's DHT peer daemon is connected to other CERTDIST instances through the Kademlia protocol, a DHT lookup will arrive at several nodes in the DHT, which are responsible for the key searched. A DHT node which gets a lookup request will check its certificate store, and if certificates matching the description given in lookup are found, the result will be returned. As soon as one of the nodes returns the answer, the DHT lookup will return the answer to the querier peer daemon.

In our example, when a user wants to use the overlaysec slice, the lookup key will be generated from the username and slice name, and the certificate signed by Inria or Thierry would be found in DHT. Similarly, when a user wants to perform an operation, the required privilege or the namespace can be used to generate the hash, together with the username, and one or more certificates can be retrieved.

Authoritatively from the Signer

A certificate can be retrieved with an authoritative query, which is sent directly to the signer of the certificate at step 4, if the signer is known. In our certificate example, a query will be sent to plc.eu.inria's IP address. An authoritative query can require, first, the resolution of the signer's IP address, which itself may require retrieval of several certificates. The address resolution policy is application specific and can be written by the programmer at the auxiliary functions part of CERTDIST. After a certificate is found remotely, it can be stored at the certificate store, which is not shown in the figure.

In the access control policies section of CERTDIST, there is one system-defined checker function, called request_cert, for certificate requests made to the CERTDIST daemon. This is required because revealing digital certificates to others is also subject to security policy, meaning that not everybody can retrieve any certificate from a CERTDIST daemon. Any application has request_cert, but different applications can implement different semantics for the function, such as different preconditions for various types of certificates. When a certificate request arrives at step 5, it is looked up at the local certificate store and local policies only and returned if found.

4.2.3 Policy Language

CERTDIST can express not only policies seen in RT and SD3, including attribute-based delegation and parameterized attributes, but also programmable credentials, using Prolog policies. Logic programming is declarative, readable and unambiguous, therefore is a natural choice for specifying policies, hence for our TM system; it is also used in other TM systems like SD3 and PeerTrust, as well as other contexts like representing business policies in CLP contracts [28, 29, 54]. Prolog also improves compliance checking of our system, as discussed in Section 4.3.4.

We mentioned that the local access control policies are complex rules that have arithmetic, string operations and compound data structures. Such policies can be part of credentials as well, in addition to the local policies, which we specifically call *conditional credentials*, meaning, executable certificates rather than with simple attribute bindings. Also, we extend Prolog with our advanced certificate retrieval primitive, that handles the DHT certificate retrieval that was discussed in Section 4.2.2, and the cryptographic operations underneath.

Among the previous work, the PolicyMaker system supports fully programmable credentials; hence it is very expressive but has high computational and programming complexity, as mentioned in Section 2.8. Later TM systems are generally built for purposes where complex policies are not needed. As mentioned before in Section 4.3.4, RT family of languages are attribute based, and do not contain primitives like string processing for parameterized attributes. SD3 can express parameterized attributes, but lack programmable credentials, and is restricted to DNSSEC policy.

We categorize the complexity level of CERTDIST policies and explain them with comparison to previous TM systems. Both the credentials and the local policies in CERTDIST can be constructed by four policy types or constructs: attributes, conditionals, general-purpose operations, and repetitions. We will give examples of these category of policies in CERTDIST later in Section 4.3.

Attributes

Assigning attributes to principals is the simplest type of policy. An attribute is one or more values in basic data types, such as string, integer, and so on, which denote roles or properties of a principal. All SD3 credentials are attribute policies. An example SD3 credential is given below; it is in the form of a tuple, signed by key K3, associating a DNS name "a.gtld-servers.net" with an IP address.

K3$DNS(a.gtld-servers.net.,198.41.3.38)

Conditionals

Some policies express conditions for principals to be able to have certain attributes. SD3 expresses its access control policies with Datalog rules, which can contain conjunction and disjunction of conditions but cannot have conditionals in its credentials. The RT family of languages can express conditionals in credentials as well as the local policy. These credentials can give attributes to a principal with the condition that the principal has some other attributes, and their conjunction and disjunction. Being able to express conditionals in credentials is important for expressing policies such as attribute-based delegation. As an example, consider the following RT credential.

$$A.R \leftarrow A.R_1.R_2$$

This means that if an entity which is given attribute R_1 by A gives attribute R_2 to any entity B, then A believes that entity B also has attribute R. If R_2 and R are equal, this means A delegates the right to issue the R attribute to anyone having the $A.R_1$ attribute. Such statements allow policy writers to state general rules about the ownership of attributes, avoiding having to write policies for each principal.

General-Purpose Operations

Sometimes we need to express complex operators and compound data structures in policy, such as arithmetic, string, date/time operations, and lists. In the application example given in Section 4.1, the sum of the credit ratings should be at least 250, which the policy language should be able to express arithmetically. The same holds for strings, which was discussed in DNS example of Section 4.3.4. List arguments are also needed for policy statements.

The RT family of languages does not support general arithmetic/string operations or lists, but one can place simple constraints on the variables with enumeration, integer, float, and date-time values. Similarly, SD3 does not have most of the general purpose operations.

Repetition

A policy statement may express repetitive tasks, such as conditions to be held for each element of a given list, which can be realized by loops or recursion. SD3 access control policies have recursion,

by which they can operate on structured strings. For example, in order to prove a DNSSEC name to key binding, SD3 policy runs the proof recursively starting at root to a variable depth in the tree. RT language does not have loops or recursion in policies. For example, as intersection of k attributes is expressed by statically writing each of the attributes in the policy, rather than looping until conjunction of k attributes are obtained, which can be useful when k is large, and especially using parameterized attributes.

4.2.4 Prolog: Programming in Logic

CERTDIST uses Prolog to represent the four policy types—attributes, conditionals, general-purpose operations, and repetition—listed in Section 4.2.3. Next, we give an introduction to SWI-Prolog [64], and show how these policy types are represented using Prolog.

Facts and Rules

A Prolog program consists of definition of a set of predicates—that is, functions that return either true or false. Predicates are defined with *facts* and *rules*. Consider the following Prolog program:

```
child(alice, bob).
child(alice, jack).
child(john, bob).
child(john, jack).
sibling(bob, jane).
sibling(X, Y) :- child(F, X), child(F, Y), not(X=Y).
```

The program defines two predicates, `child` and `sibling`. Here, `child` is defined only with facts, whereas `sibling` is defined both by a fact and a rule. A rule defines the conditions for a predicate with the Horn clause :-, such that all predicates on its right-hand side have to evaluate to true for the predicate to be true. The predicate `child(P, C)` means `C` is a child of `P`. The comma indicates logical AND and the semicolon indicates logical OR operators when placed between two predicates. Any term that begins with an uppercase letter is a *variable*. Therefore, the rule says that `X` is a sibling of `Y` if they share a common parent and they are not the same.

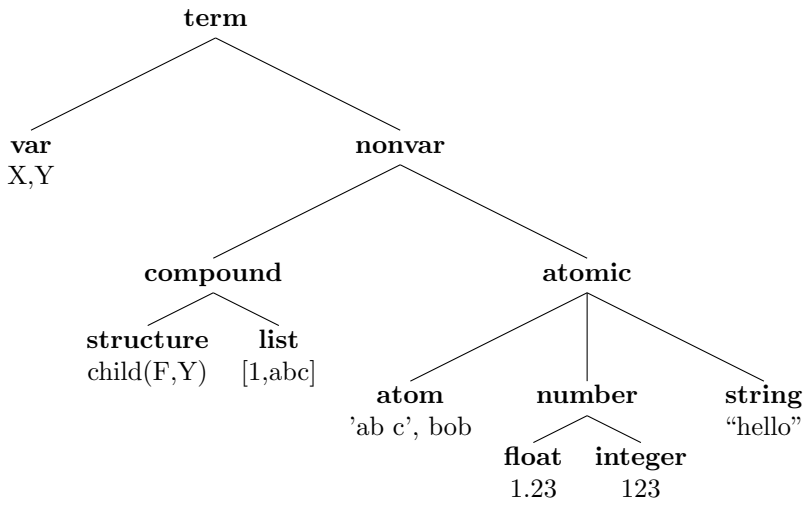


Figure 4.4: Prolog's type hierarchy

Data Types

Figure 4.4 shows the Prolog data types and their examples. Atoms are non-compound Prolog values that are not numbers or strings, such as `john` or `'alice bob'`. A structure is a term in the form `child(X,Y)` where here `child` is called the *functor* and the rest are arguments. Arguments themselves can be structures as well.

Lists are represented by comma-separated elements within square brackets. Elements of a list are not accessed in random fashion, but rather one by one, starting with the first element. If L is a list, then the expression `L = [Head|Tail]` allows us to get the first element at `Head` and the remaining list at `Tail`.

Execution

A Prolog program is run by querying the program with a *goal*. The Prolog engine tries to evaluate the goal to true, and if it can, it also finds all possible answers. For example, if we provide the goal `child(alice, X)`, then the program will return a result set containing `child(alice, bob)` and `child(alice, jack)`. On the other hand, the goal child(john, jane) will return false. The goal `sibling(X, Y)` will return true with the following answers:

```
sibling(bob, jane)
sibling(bob, jack)
sibling(jack, bob)
```

```
sibling(bob, jack)
sibling(jack, bob)
```

The reason of this ordering is because the Prolog engine processes facts and rules top-down. The goal `sibling(X, Y)` will match to the sibling fact first; therefore, `sibling(bob, jane)` is output first. Next, the goal will try to be satisfied by the rule. Prolog engine tries all possibilities of matching in a process called *backtracking*. For example, since there are 4 child facts and the sibling rule requires a pair of child predicates, 16 combinations of pairs are possible. Eight of the pairs fail because they do not have same parent, and 4 of them fail because they do not satisfy `not(X=Y)`. The final 4 pairs produce the remaining results.

Cut Operator

Programmers can control backtracking using the *cut symbol* (!). The Prolog engine does not backtrack for predicates before a cut operator, which means it forgets all the alternative matching possibilities up to that point. For example, suppose we write the above rule as

```
sibling(X, Y) :- child(F, X), !, child(F, Y), not(X=Y).
```

This time the goal `sibling(X, Y)` will produce only the answers `sibling(bob, jane)` and `sibling(bob, jack)`. This is because once the first `child` predicate is matched to `child(alice, bob)`, Prolog will not try to match it to another `child` fact. The second child predicate will try to match all the four `child` facts, of which only one satisfies the rule.

Special Symbols and Built-in Predicates

Terms that begin with an underscore are anonymous variables whose values are not remembered and can match any value. Special predicate `not(P)` gets the negation of the predicate `P`.

Implementing Policy Types

Using structures, we can bind atoms to strings or other values, effectively representing attributes or parameterized attributes. A rule is a conditional, where the right hind side of a Horn clause is the conditions for a predicate to hold true. We can represent attribute-based delegation or more complex conditions using a Prolog rule. SWI-Prolog [64] has a large library that can be used to realize general-purpose operations and has lists, among other data structures like queues. Finally,

Prolog has a similar usage of recursion as SD3 which is based on Datalog, which handles the repetitive policies. SD3 also regards recursion to be a required piece for implementing realistic policies [32].

4.3 CERTDIST Example

We give an example application that uses CERTDIST and specifically show access control policies, public and private trust relations, certificate retrieval, and conditional credentials. We use an application example independent from federations to better describe CERTDIST's processing as a stand-alone TM system.

The example is from the PolicyMaker [16] paper, which first introduced the trust management concept. Think of a work-flow system in a company, where there is a purchase department, personnel department, directors, and the company manager. The manager has the right to place orders at the purchase department without any limitations. The manager can delegate this right to others by placing limitations such as restricting the maximum amount in a purchase. This delegation can be made by others, as well, resulting in a chain of delegations. The manager can also delegate taking care of company personnels' positions to the personnel department, which therefore can attest who is a director, a regular worker, and so on. One specific purchase policy set by the manager is that purchase orders larger than $1,000,000 can be issued only if at least three people authorize them, and all these people are attested to be company directors by a specific personnel key. When the three directors Jack, Joan, and Matt want to make a purchase, they send a request to the purchase department's servers, which basically can be represented with the tuple: (Amount, ["jack_key", "joan_key", "matt_key"]). For such a request to go through, six different policy statements need to be present at the purchase department:

1. Manager's key, trusted by the purchase department
2. Purchase policy set by the ManagerKey
3. PersonnelKey attests JackKey as director
4. PersonnelKey attests JoanKey as director
5. PersonnelKey attests MattKey as director
6. Jack, Joan, and Matt authorize purchase with specified amount

Among these policies, possibly all except (1) are certificates, that is, remote policies. (1) is in the local policy of the purchase department, as a part of its trust relationships, because one can assume that everybody knows and trusts the manager's key. On the other hand, the manager's purchase policy can possibly change, as do the personnels' positions. Therefore, such certificates have to be retrieved remotely or submitted together with the request.

Figure 4.5 shows the local policies of (a) the purchase department, (b) the manager, and (c) the personnel department. Line 1 in (a) and all lines at (b) and (c) are *trust relations*. Lines 3–20 at (a) are *access control policies*, where lines 11-20 are part of a common schema. We use the shorthand notation "...key" to refer to keys that stand for very long strings.

4.3.1 Access Control Policy

There is one access control policy called `request_purchase`, which is located at the purchase department. This policy is also defined in terms of some other predicates, which are shown from lines 6 to 20. The policy will be invoked when the event handler at manager's CERTDIST daemon makes a remote call in order to purchase at the purchase department. The event handler at manager daemon simply performs the remote query without complex certificate collection, and, therefore is omitted here. The action is allowed only if the result of `request_purchase` is true.

4.3.2 Trust Relations

Trust relations can be specified both with Prolog facts and rules. In our example, `trustedKey`, `can_purchase`, and `director` are predicates that express trust relationship policies, where the first two are specified by facts and the third is specified by two facts and a rule. The `director` predicates at lines 1 and 2 are Prolog facts, which specify the people who are directors probably for a long time, less likely to change, so they are hardcoded. In addition, the new assigned directors can be read from a more dynamically updated file, "directors.txt", using the Prolog rule at line 3. The rule looks up a given key in the file and returns true if it is found in the file. To avoid queries pulling all file information, the `Key` must be specified as a value. The built-in Prolog predicate `var(X)` returns true if X is not set to a value, and the built-in predicate `not(P)` gets the negation of P. In addition, `member` is a built-in predicate that checks membership in a list. Lastly, `file_to_list` is a predicate that can be implemented using built-in Prolog I/O operations such as `see` and `read`.

(a) Local Policy of Purchase Department:

```
1.  trustedKey("manager_key").
2.
3.  request_purchase(Amount, KList, CallerKey) :- KList = [CallerKey|_],
4.                                                credential(amount(Amount), KList),
5.                                                preconditions(Amount, KList), !.
6.  preconditions(Amount, Key) :- trustedKey(Key).
7.  preconditions(Amount, Key) :- credential(can_purchase(Key, Conds), XKey),
8.                                Conds = [Rule],
9.                                apply_conditions(Rule, [Amount, Key]),
10.                               preconditions(Amount, XKey).
11. apply_conditions(Rule, Param) :- Rule = (rule(Param) :- Prog), Prog, !.
12.
13. subset_signed(N, KList, KSign) :- subset_signed_rec(0, N, KList, LSign).
14. subset_signed_rec(N, N, _, _).
15. subset_signed_rec(X, N, [KNext|KList], KSign) :-
16.                               credential(director(KNext), KSign), !,
17.                               NewX is X+1,
18.                               subset_signed_rec(NewX, N, KList, KSign).
19. subset_signed_rec(X, N, [KNext|KList], KSign) :-
20.                               subset_signed_rec(X, N, KList, KSign).
```

(b) Local Policy of Manager:

```
1. can_purchase(_, [(rule([Amount, KList]) :-
2.                              subset_signed(3, KList, "personnel_key"),
3.                              Amount <= 1000000)]).
```

(c) Local Policy of Personnel Department:

```
1. director("jack_key").
2. director("joan_key").
3. director(Key) :- not(var(Key)),
4.                  file_to_list("director.txt", DList), member(Key, DList).
```

Figure 4.5: CERTDIST implementation of PolicyMaker example

As mentioned before, some trust relations are *public*, meaning they can go in credentials, and the others are *private*, used only locally. In our example, `trustedKey` is private, whereas `can_purchase` and `director` are public trust relationship policies.

Public Trust Relations

Public trust relationship policies are the set of policies that can be placed inside certificates and shipped remotely. We choose the format of such policies, hence the credentials, always to be a Prolog structure for uniformity.

Some of the public trust relationship policies can be simple facts such as the `director` predicate of line 1 at (c). Such policies have the format, as seen below, where the *functor* is the type of the policy statement, and the *Fields* is an ordered sequence of Prolog values.

$$functor(Fields)$$

On the other hand, some public trust relationship policies express conditionals by including conditions as the last field of the Prolog structure. When placed inside certificates, such policies produce *conditional credentials*; therefore, we call such policies *conditional public trust relations.*

We explain the format of these credentials for the clarity of our example. Although this does not represent an extension to the Prolog language, hence not a contribution by itself, the fact that it can be placed in certificates and retrieved remotely is a distinguishing part of CERTDIST. The `credential` primitive that does that will be described in Section 4.3.3.

A conditional public trust relationship policy is a modified form of Prolog rule, where the right-hand side of a Horn clause is placed as the last field of a Prolog fact. The specific format is as follows:

$$functor(Fields,\ Conditions)$$

Fields	=	zero or more comma-separated Prolog values and anonymous variables
Conditions	=	list of *Rule*'s
Rule	=	(rule(*ParamList*) :- *Prog*)
ParamList	=	a Prolog list containing variables
Prog	=	a Prolog program that uses *ParamList*

The specific prototype of *functor* is assumed to be known by whoever uses it as part of a common schema. This includes the fields *Fields*, parameters *ParamList* and their semantic. *Fields* is used to express the policy using non-variable terms; however, it can contain anonymous variables, too, in order to make general statements about policy. *Conditions* defines a list of rules, each rule being a predicate that must hold true for the policy to be satisfied. A rule has parameters *ParamList*, and the conditions on those parameters, *Prog*, which is basically a Prolog program. This means the credentials can carry programs, but not arbitrary code, which will be discussed later.

In the example, the `can_purchase` predicate is a Prolog fact which expresses a conditional public trust relationship policy. *Fields* contains one element that denotes the set of keys to which purchase right is being delegated. *Conditions* contain a single *Rule*, where *ParamList* contains two elements: the delegees' keys, `KList`, and the requested purchase amount, `Amount`. So, this predicate can delegate the purchase right either directly to a set of keys, using the *Fields*, or it can place conditions based on delegee keys and the purchase amount, using *Conditions*. More than one *Rule* could be needed for cases where *Conditions* contains multiple *Rules*, where, each is a logically distinct predicate with different parameters.

The `can_purchase` predicate appears in the local policy of the manager, (b), where *Fields* is to an anonymous variable, _, meaning the credential is not issued for a specific principal. Instead, its *Conditions* specify conditions on the `KList` about who can use this credential and with what amount, `Amount`.

4.3.3 Certificate Retrieval

CERTDIST can search and retrieve certificates using the special predicate `credential`, as seen in Figure 4.5. We explain the implementation of this predicate in Chapter 7 in more detail, including its optional parameters. Here, we explain only the high-level processing of the predicate, with its basic format as follows:

```
credential(Statement, SignerKey)
```

The predicate certifies that `Statement` is uttered by `SignerKey`. `Statement` is a Prolog structure that specifies the certificate content being searched. This structure can contain fields that are Prolog variables, which allow certificate retrieval with partial knowledge, as explained in Section 4.2.2. The `credential` predicate can find a fact that matches `Statement` in any of four ways:

from the local certificate store, by signing, from DHT, or authoritatively. If there is more than one match, `credential` will backtrack on these. If the predicate found a list of certificates remotely, it will store these locally and backtrack on the locally stored certificates. Note that, it is possible to write inefficient policy, such as one that backtracks on a previous predicate and performs many remote certificate queries, each retrieving only one certificate, although retrieving all certificates at once may be possible. These are considerations that the architects keep in mind while writing policies. If `SignerKey` is a list of keys, then `credential` will verify it is uttered by all these keys. This means every key indexed i in the list utters `Statement` and is also aware of previous keys in the list who also utter it, that is, the ones indexed $i = 0, .., i - 1$. This can be implemented by i signing a certificate that contains a certificate signed by $i - 1$, and so on, which finally contains `Statement` itself.

The auxiliary functions part of CERTDIST, mentioned in Section 4.2.2, will contain functions such as IP address resolution and determining the DHT keys from the certificates. Such functions can be written by the programmer; for example address resolution, a directory can map known public keys to IP addresses. Second, the DHT keys can be hash of non-compound fields of `amount`, `director`, and `can_purchase` structures. Such functions will automatically be utilized by the `credential` primitive while performing authoritative and DHT queries, respectively. Federation architects/CERTDIST programmers decide on the best ways to utilize the DHT; what certificates to store/lookup under what keys, since this is a very policy dependent process. However, it can be a future work for CERTDIST to not allow certain ways where DHT definitely does not make sense.

4.3.4 Compliance Checking

Here we explain the overall flow of the compliance checking in our example, where a set of directors want to make a purchase. As mentioned before, our example application has one access control policy that performs compliance checking, called `request_purchase(Amount, KList)`, as seen in Figure 4.5. We assume Matt's CERTDIST daemon performs a call `request_purchase(250000, ["matt_key", "joan_key", "jack_key"])` and also passes a certificate that contains `amount(250000)`, signed first by Matt and then by Joan and Jack. We can assume there is a unique identifier per certificate so that one `amount` certificate cannot be used to perform multiple purchases. After the certificate is stored at the receiver, `request_purchase` is invoked with `CallerKey` set to

"matt_key".

The processing of request_purchase in Figure 4.5 is as follows. In line 3, it checks whether the call was made by the first element of KList, the list of requester keys. In line 4, the credential predicate ensures that the specified Amount is approved by all keys at KList. It performs this by finding a certificate containing amount(Amount), signed by the keys in KList in specified order. The credential predicate will find it in the certificate store, and if it satisfies the ordering, it will go on with further preconditions at line 6.

The predicate preconditions checks if Key and Amount comply with the purchase policy. Recall that the manager has the right to order a purchase with any amount, and it can delegate this right to others. Thus, it can be the case that a chain of delegation certificates proves that Key can order with size Amount. These semantics are reflected by the recursive property of preconditions predicate starting at line 7, which says that Key can order with amount Amount, provided that first there is a principal with key XKey who delegates to Key with a set of conditions, Conds (line 7); second, the single rule Rule in these conditions can be satisfied with parameters Key and Amount (line 9), and, finally, XKey itself has the right to order a purchase of size Amount (line 9). Line 6 is the base of this recursion, which indicates that recursion can end with a trusted key, such as the manager.

In our example query, the credential predicate at line 7 will try to find a certificate that matches:

```
can_purchase(["matt_key","joan_key","jack_key"], Conds) signed by XKey
```

We can see that the Key field of can_purchase is set to a value, whereas Conds and XKey are variables. This means the signer of the certificate is not known; therefore, credential cannot make an authoritative query but can possibly find it either from store or DHT. If Matt had already passed this certificate together with the request, credential will find it locally in the certificate store. However, credential cannot find it in local policy, because the purchase department trust relationship policies do not involve a can_purchase fact or rule with this list of keys. But, credential can also query DHT with keys derived from two possible sets of fields: {can_purchase} and {can_purchase, ["matt_key", "joan_key", "jack_key"]}. The former will search for all can_purchase certificates in the system, and the latter will search for ones that are signed specifically for this key list. In our example, only the former can succeed, because the

manager's delegation certificate would be stored only with the former key, since its `KList` field is a variable.

Either from certificate store or DHT, `credential` will find a certificate with content as follows, which matches to the previous certificate definition. This is the certificate created by `manager_key`, which contains the manager's policy at (b).

```
can_purchase(_, [(rule([Amount, KList]) :-
                    subset_signed(3, KList, "personnel_key"),
                    Amount <= 1000000)]
            ) signed by "manager_key"
```

After `credential` executes successfully at line 7, `XKey` will be set to `"manager_key"`, and the `Rule` will be set to manager's 3-out-of-k keys delegation policy, at line 8, where `Rule` is as shown below.

```
rule([Amount, KList]) :-
    subset_signed(3, KList, "personnel_key"),
    Amount <= 1000000
```

At line 9, `apply_conditions` will be invoked with all its parameters set to values, that is, `Rule`, `Amount` and `Key`. The definition of `apply_conditions` in line 11 simply runs the program `Prog` located inside the rule, which is shown below.

```
subset_signed(3, ["matt_key","joan_key","jack_key"], "personnel_key"),
250000 <= 1000000
```

This program will evaluate to true only if `subset_signed` succeeds. This predicate is part of the common schema between the caller and callee, whose definition is given in line 13. It returns true if at least `N` number keys from the list `KList` are verified to be directors by the key `KSign`. The predicate is defined recursively, checking elements of `KList` one by one at each call. Line 16 tries to find a certificate signed by `KSign` and states that the first element of the list is a director. If it succeeds, the current number of found directors is increased by one, and the recursive call is made to find the others. If it fails, line 20 will be executed, where the recursive call is made without increasing the current found number. The recursive predicate will succeed when `N` number

of directors are found at line 14, or it will fail when all elements of `KList` are traversed and no more calls to recursive predicate are possible.

Note that the call to `credential` predicate at line 16 can perform authoritative queries as well, in contrast to the `credential` at line 7, because this time the signer key field is set to a value, `personnel_key`, which can be used to map to its IP address and query the personnel department directly for certificates.

After `apply_conditions` succeeds at line 9, `preconditions` will be called recursively, this time trying to satisfy `preconditions(250000, "manager_key")`. This will succeed at line 6, since `manager_key` is one of trusted keys in trust relation of line 1. As a result, the `request_purchase` predicate evaluates to true, and the application calling the TM knows that this call is authorized and can place the order.

Our example shows that a complex policy can be implemented by CERTDIST with a minimal amount of code. We note that, the format of the credentials and other common policy elements need to be uniformity known by the different principals. In fact, in every distributed system there has to be a common understanding of data structures and operations so that distinct entities can interoperate. All the credentials and access control policies in a CERTDIST application should be part of the common schema. In our example, the following predicates' names and parameters are schema elements: `can_purchase`, `request_purchase`, and `director`. In addition, the complete definition of `subset_signed` is part of the schema. The schema should be periodically exchanged among different entities in the system in order to interoperate.

4.3.5 Policy Complexity

The example of Figure 4.5 contains all four types of policy constructs: assertions, conditionals, general-purpose operations, and repetitions. Trust relations like `director` predicate express attributes about principals. Both local policies and credentials can contain conditionals, expressed by rules and facts. Built-in Prolog list operations are used in recursion for repetitive policy processing.

Using these constructs we can build more diverse and complex policies. For example, an access control policy can look for certificates in different directions, such as starting at the root of a chain, or the subject, as in the case of the `request_purchase` example. Another example is that one can place a limit on the delegation chain length using an additional parameter to the

recursive checker function, incrementing and checking this value at each call, and so on. Also, credentials can contain more complex programs so that `apply_conditions` produce results for one credential, which can be a parameter to the conditions of another credential. In this way, policies that require sharing states between different credential conditions can be implemented. As a final example, it is possible to use a Prolog list to keep the set of credentials used in the policy and place group constraints on them. These examples of more complex access control policies can be easily implemented with CERTDIST.

One drawback of expressive credentials is that they can contain possibly dangerous operations, such as writing to disk, sending packets over a network, inserting new rules or facts in the policy, and so on. A malicious credential should not be permitted to execute such operations. In the next chapter we provide a schema of high-level elements in the FPL language, which also overcomes such problems in a federation by introducing a safe set of primitives to construct credentials. One approach in future work can be to allow only the credentials constructed with such primitives or to allow arbitrary credentials only from trusted entities.

In the context of our federation framework, security architects are responsible for the efficiency and correctness of the policies. Also, every certificate signed in the federation framework can be traced back to a HRN, human readable hierarchal name, which also is a basis for accountability. An organization in a federation, that is, an academic institution, testbed, cloud computing organizations, etc., participates in contractual agreements with other organizations, hence is directly accountable to others in the credentials issued. We expect such accountability to ensure efficiency and proper operation of policies to a large extent.

Chapter 5

Federation Policy Language

In this chapter we introduce the federation policy language (FPL) piece of the federation framework, which is used to express policies in a federation. FPL is built on top of CERTDIST and is a schema that is designed to express federation policies. FPL has the same syntax as Prolog and can use Prolog built-in and CERTDIST predicates. FPL constitutes federation specific functions and structures that make it possible to express policies, and generate new schema elements. It is easy for federation security architects to extend FPL policies, using CERTDIST and existing FPL elements.

FPL has a distinct feature: it can express both security and allocation policies, as opposed to languages like ABAC [37] that was proposed for GENI or sfatables [11] that is used in PlanetLab. While it allows very complex policies, it is also easy to express such policies. FPL achieves this by layering the policy language on top of a distributed trust management system, which represents a unique place in the design space of other policy languages and trust management systems. We argue that the policy languages and trust management systems are mutually dependent in most real deployment scenarios, such as computational federations.

First, a trust management system cannot be fully useful without a domain-specific policy language. To illustrate this, consider the ABAC trust management system that is designed for applications where access control is attribute based. ABAC has high-level constructs to express such policies easily, but they are not specific to a particular domain, hence are limited. For example, ABAC is one candidate TM to implement GENI access control, but could also be used to implement a web publishing service authorization as mentioned in their paper, or internal policy

in a company, and so on. Although ABAC is designed specifically for attribute based policies, it does not support simple features such as delegation depth for attributes. This is as a result of how ABAC addresses the trade-off between expressiveness and ease of use: limiting expressiveness to make it easier to write and understand policies. If GENI needed delegation depth based policies in the future, ABAC would not be able to implement such a policy, since it is designed from the start for ease of use and simpler policies. Although a TM system is a static mechanism with a fixed code base, policy is a human dependent, dynamic set of rules, that evolve by time. Therefore, "one size fits all" approach does not work if we want to apply a TM system directly to implement polices.

Instead, we take a systems approach based on layering, where the trust management system is distinct from a domain-specific policy language that is layered on top of it. Our trust management system, CERTDIST, is Turing-complete, hence fully expressive as discussed in Chapter 4. FPL does not have to expose all of this expressiveness, maintaining the simplicity in policies. For example, FPL does not include delegation depth in its security policies but it is easy to include such a policy construct to the language if needed. As a result, we address the trade-off between ease of use and expressiveness by layering policy language on top of a very expressive TM system, while maintaining the possibility for this language to be extended by security architects. A rough analogy would be: CERTDIST is similar to C or assembly language, with which higher-level languages can be written; Python and Java are analogous to FPL, where each language is useful in a different context.

Similarly, a policy language is not fully useful without a trust management system. Policy languages such as sfatables and CLP express the amount of, or conditions for, local resources to be granted to other principals. Such a policy is written by admins at an organization, and stored locally to later facilitate the decision of whether to accept or deny incoming requests, and to keep track of accounting of allocations. However, more complex polices often need to refer to certain roles rather than individual principals. For example, in PlanetLab or GENI, one can grant certain resources to all participants of a particular slice. This requires a security policy definition of what slice membership is, which can be complex, and so requires a trust management system. The general observation is that, a chain of certificates is required that starts at the resource owner and ends at the requester, for resource allocation to happen. In the simplest allocation policy, this chain has length 1, which is the locally stored policy at the organization. In the complex case, a

chain of certificates can define not only to whom the resources are granted, but also the resource itself. The resource that is granted can be defined by a chain of certificates, which happens when an organization is delegating third-party resources. In that case, this allocation policy would appear somewhere in the middle of the chain of certificates that grant resources to a principal.

FPL achieves both ease of use and expressiveness through layering on top of a trust management system, but requires a distinct group of people—security architects—to evolve and extend the language. This can be both an advantage and a disadvantage, since first, architects must have both an understanding of the federation requirements and the technical knowledge of logic programming to express this formally in CERTDIST. On the other hand, it is an advantage to understand the pros and cons of candidate policies and also to reduce ambiguity in discussions on policies.

One drawback that comes with the power of a fully programmable trust management system is the potential for mistakes or malicious code in the implementation of FPL elements. Accountability features of FPL allow one to trace every signed certificate to a vouched principal in the system, where a human-readable hierarchical name (HRN) represents this vouching relationship; this is discussed more in depth in this chapter. HRN and the security architects can help reduce the effects of such drawbacks, and we see the additional mechanisms to address these issues as future work.

Table 5.1 shows the FPL elements and their simple typing. FPL types are derived from the Prolog types and denoted after a colon following FPL fields or elements. For example, *hrn* is a Prolog atom that must be in the format of an HRN, as discussed in Section 3.3.1. Similarly, some fields must contain valid public keys, denoted by the *pkey* type, which is basically a Prolog string. Also, *cert* is a Prolog string that contains a valid x509 digital certificate. The overall FPL elements are in three larger categories; security policies, allocation policies, and utility elements. Also, all three contain subcategories determining the context where the elements are used, such as credential types, authentication elements, resource types, and so on. In this chapter we will explain each of the subcategories in detail; however, we explain their implementation using CERTDIST in Chapter 7.

Figure 5.1 shows how different subcategories relate to each other, which is similar to the high-level Figure 3.3. Contracts are a key abstraction of FPL, and will be discussed in Section 5.4, and are shown as the highest level element in the figure. Admins at each organization in a federation can create contracts to manage allocation of their resources, place complex allocation policies

Category	**Security Policy**
Federation Statements (Cred. Types)	<*fedSt*>= <*idSt*>| <*ipSt*>| <*attrSt*>| <*resSt*> **ip**(Signer:*hrn*, Subject:*hrn*, IP:*ip*):*ipSt* **id**(Signer:*hrn*, Subject:*hrn*, Key:*pkey*, Authority:*bool*):*idSt* **attr**(Signer:*hrn*, Subject:*hrn*, Location:*hrn*, Attribute:*atom*, Delegate:*bool*):*attrSt* **resource**(From:*hrn*, To:*hrn*, ID:*int*, (- | Resource:*resSt*), Constraints):*resSt*
Statement Proofs	**certify**(Statement:*fedSt*, From:*hrn*, Method:*atom*, CertList:*cert list*)
Authentication Policy	**authenticate**(HRN:*hrn*, Pubkey:*pkey*, Authority:*bool*)
Role/Attribute Authorization	**has_attr**(HRN:*hrn*, Target:*hrn*, Attribute:*atom*, Delegate:*int*)
Resource Authorization	**has_resource**(Permit:*resSt*)
Category	**Allocation Policy and Contracts**
Resource Specification	**nodespec**(Name:*atom*, Loc:*int*, Core:*int*, Disk:*int*, Cpu:*float*, Mem:*int*, Bw:*int*):*node*
Resource Constraints	**mn_limit**(Permit:*resSt*, NodeList:*node list*, Limit:*int*) **fr_limit**(Permit:*resSt*, NodeList:*node list*, Limit:*float*) **exclude_machines**(NodeList:*atom list*, EList:*atom list*)
Accounting & Capacity	**scheduled**(Permit:*resSt*, Nodes:*unit list*, Total:*unit list*) **capacity**(Permit:*resSt*, Capacity:*float list*) **account_nodes**(Permit:*resSt*, Nodes:*unit list*)
Third party Resources	**permit_from**(From:*hrn*, Permits:*resSt list*)
Contracts	**contract**(ID:*int*, Start:*date*, End:*date*, To:*hrn*, Resource:*resSt*, Constraints) :- Conds
Category	**Other**
Utility & Built-in Predicates	**get_authority**(HRN:*hrn*, AuthHRN:*hrn*) **begins_with**(HRN:*hrn*, Prefix:*hrn*) **permit_origin**(Permit:*resSt*, Origin:*hrn*) **is_onpath**(HRN:*hrn*, Permit:*resSt*) **can_alloc**(Permit:*resSt*, NodeList:*node list*) **arg**/3, **not**/1, **var**/1, **member**/2
Basic Policy	**myIP**(IP:*ip*) **rootIP**(IP:*ip*) **myHRN**(HRN:*hrn*) **rootHRN**(HRN:*hrn*) **myPubkey**(PKey:*pkey*) **rootPubkey**(PKey:*pkey*) **myPrivkey**(PKey:*pkey*)

Table 5.1: FPL elements and types: Types are derived from Prolog types, such as *hrn*, which is a Prolog atom that must be a human-readable name (HRN), *pkey*, a string containing a public key, or *cert*, which should be a valid x509 certificate.

to express the conditions of allocations and also to whom these resources should be granted. Contracts use the security and allocation policy elements, which can be extended by the security architects in federation. Security polices are complex predicates that define the required set of certificates for the policy to be true; therefore, they use certificate retrieval functionality and also define a set of known certificate types at a lower layer. Allocation policies use elements for various resource types, including third party resources that are granted to this organization from others. Also, allocation polices use the FPL resource constraint elements, which are implemented using the accounting and capacity primitives.

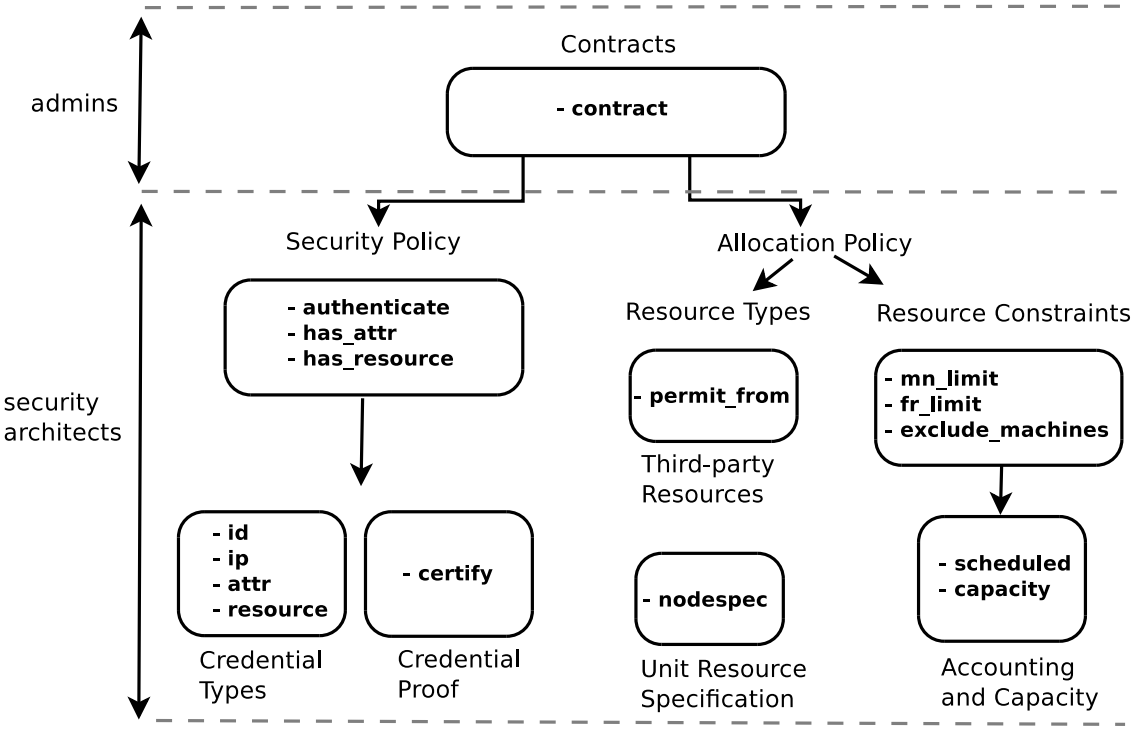


Figure 5.1: Relationship among various categories of FPL elements. Contracts are used by admins, while security architects create and extend security and allocation policy elements at a lower level using CERTDIST.

5.1 Federation Statements

Federation statements are predicates that can go into certificates and, therefore, determine the credential types in a federation. From the CERTDIST point of view, these are the public trust relationship policies of an organization, as discussed at Section 4.3.2. Using such statements, an

organization makes a declaration about other principals, such as authenticating users or delegating resources to other organizations. The first parameter in all these statements is the organization's own name, the second is the name of the subject, and the rest are statement information. FPL uses human-readable hierarchical names (HRN), as exemplified in Section 3.3.1. So, an organization called plc.eu can make a policy such as `ip`(plc.eu, plc.eu.inria, '139.19.142.6'), declaring the network address of another organization. Similarly, `id` statement binds a subject to a public key and also a boolean, "authority," which, if set, allows the subject to create names under its namespace. The `attr` statement binds a subject to an attribute at a specific location. For example, plc.eu can give record lookup right to plc.eu.inria at the plc.eu namespace. In addition, a boolean, "delegate" determines whether this attribute can be delegated to others. The statements `ip`, `pubkey`, and `attr` can simply reside at a local configuration file of an organization, or they can be looked up from other sources, such as a relational database.

The `resource` statement denotes resource delegation, and we call the credentials containing this statement *permits*. In CERTDIST terms a permit is a conditional credential and has the format mentioned in Section 4.3.2, containing conditions as its last field. The source of the delegation is denoted by the fourth field, `Resource`, which can either be '-', for local resources, or another permit, expressing third-party resources. The `resource` statements are derived from contracts made by the organization, which will be discussed further in this chapter.

5.2 Statement Proofs

Federation statements can be proven by the FPL predicate `certify`, which uses CERTDIST's built-in `credential` primitive to retrieve certificates and prove a statement. It has four arguments, where only the first, `Statement`, is mandatory. It finds a set of certificates that proves a federation statement, `Statement`, is true. It separately authenticates the `Signer` field of a federation statement in order to retrieve its public key, finds a certificate signed by this key, and contains the federation statement. The `From` field specifies an HRN to which to send authoritative queries, and `Method` is a sequence of characters determining types and order of queries. For example, with Method = `slda`, it will check for certificates first at the certificate store, then local policy, then DHT, and, finally, authoritatively. The final list of digital certificates retrieved to prove `Statement` is unified with the last argument, `CertList`. The detailed implementation of the FPL statement

proof element will be explained in Section 7.2.1.

5.3 Security Policy

Security policy defines the set of digital certificates for authentication and authorization of principals in a federation. These policies are constructed by using the lower-level FPL elements previously described, the credential types, and proof elements. In this section we give some example polices and the corresponding FPL elements for them. One important feature of FPL is that such elements can easily be extended in the future by federation security architects, as required, when the federation evolves, resulting in new elements for the federation language.

Authentication policy is implemented by the `authenticate` element, which binds an HRN to a public key. There is an optional `Authority` field, which determines whether or not `HRN` can create new objects under its namespace. We next illustrate how higher-level FPL elements are implemented using the lower-level elements below, with an example authentication policy.

```
authenticate(HRN, Pubkey, 1) :- rootHRN(HRN), rootPubkey(Pubkey), !.
authenticate(HRN, Pubkey, Authority) :-
                            get_authority(HRN, AuthHRN),
                            authenticate(AuthHRN, AuthPubkey, 1),
                            Statement = pubkey(AuthHRN, HRN, Pubkey, Authority),
                            certify(Statement, AuthPubkey).
authenticate(HRN, Pubkey) :- authenticate(HRN, Pubkey, _).
```

The policy defines `authenticate` recursively, with root being the base of recursion. It assumes the public key and the HRN of the root is known. For nonroot HRN, the policy first needs to authenticate the parent of an HRN, with `Authority` set to 1. The `get_authority` predicate is a utility element, which is implemented using built-in Prolog string operators and which prunes the last piece of an HRN, obtaining its parent. For example, if the HRN is 'plc.eu.inria', then AuthHRN gets the value 'plc.eu'. The recursive call performs the authentication of the parent, and the `certify` primitive finds a digital certificate that is signed by the parent and contains the statement that confirms `HRN` is bound to `Pubkey`. Since the third field of `authenticate` is optional, the two-field version of the predicate will just verify the full predicate and ignore its last field.

There are much more complex authentication policies, which can easily be expressed by FPL. For example, there is one candidate GENI authentication policy in which the authentication and the ability to create a new object have to be expressed with separate certificates. In that case, instead of `Authority`, the policy has to look for `attr` statements to verify that an object has creation rights, such as the "Register" attribute. Furthermore, this attribute can be delegated to others; therefore, the creator of an object is not necessarily its parent in HRN namespace. The choice among various security policies and implementation of additional policies is left to security architects in our federation framework.

Roles and attributes are used to determine the authorization for various operations in a distributed system, such as modifying database records or creating new objects. The `has_attr` predicate implements the policy for assigning attributes to principals. For example, `has_attr`(plc.eu.matt, plc.eu, 'register', 1) says that plc.eu.matt has the right to create new objects at plc.eu namespace, and he can further delegate this right to others. For this predicate to be proven, a set of digital certificates containing `id` and `attr` statements has to be collected. Similar to the definition of `authenticate` predicate, we express such policy using lower-level FPL elements as well as the `authenticate` element itself.

Finally, the `has_resource` predicate evaluates to true if there is a resource path from the source to the subject of a permit. As discussed before, a permit is a `resource` statement, which is recursively defined by possibly many such statements. The `has_resource` predicate obtains the digital certificates to prove that there is actually a chain of delegations that allows `To` to possibly acquire resources. The implementation of these FPL elements is further discussed in Chapter 7.

5.4 Contracts

In the federation framework, contracts determine how an organization's resources can be granted to others. An organization creates a contract in an out-of-band way, such as negotiating with other organizations, or unilaterally by donating resources to other organizations. Once a contract is decided, an organizational admin encodes it using FPL in the organization's local policy. In CERTDIST terms, contracts are part of an organization's private trust relationship policies. Such policies are private to an organization and are not directly placed inside digital certificates, as discussed in Section 4.1 and Section 4.3.2.

The FPL element for expressing contracts, `contract`, is the highest-level element, which uses both the security and allocation policies. We explain contracts in top-down fashion by first giving an example contract and later explaining the additional FPL elements in greater detail in subsequent sections. The `contract` predicate has a unique identifier `ID` for each contract, validity dates from `Start` to `End`, and shows delegation of a set of resources, `Resource`, to a set of principals, `To`, subject to `Constraints`. `Conds` are conditions on `Resource` and `To` fields, determining what resources can be delegated and to whom.

We give an example contract that an organization might want to apply for allocation of its resources and show how it is expressed using FPL. Suppose an organization in Japan decides to grant only its resources located in South America to any organization in Europe, with a maximum limit of 100 virtual machines. For simplicity, we can express this contract in a more compact way as follows:

plc.jp gives to **(plc.eu.* org's)** resources from **(plc.south.*)** subject to **(max 100 VM)**

Such a contractual statement can be represented with an FPL `contract` element as follows. This example additionally shows details such as contract ID, which is a random unique identifier, and also the validity period. `To` and `SourcePermit` are two Prolog variables that represent the recipient and the resource paths, respectively. These variables can get values that adhere to the conditions in the right-hand side of the Horn clause, that is, `Conds`.

```
contract( 206257450416,'11-02-2013 21:00','02-02-2014 21:00',    % ID, Start, End
          To, SourcePermit,
          (rule(Cred,NodeList) :- mn_limit(Cred,NodeList,100))  % Constraints
         ) :- permit_from(plc.south,SourcePermit),              % Conds
              begins_with(To,plc.eu),
              has_attr(To, To,register, _),
              not(is_onpath(To,SourcePermit)).
```

The constraints on the resource are placed using a `rule` predicate, which has the same format as a CERTDIST conditional credential's *Rule*, defining the credential conditions, as described in Section 4.3.2. Contracts use FPL allocation policy elements to construct rule conditions. In the

example, we see that `rule` gets two parameters and applies an allocation policy, `mn_limit`, with a limit of 100 machines. We discuss this and other allocation policies later in more detail.

Contracts can grant resources that do not reside locally in the organization, which we call as third-party resources. Such resources that were previously obtained from other organizations can be queried using `permit_from` element. In the example, `SourcePermit` will match to any existing permit having a source with name beginning with plc.south.*.

Contracts can use FPL security policy elements to define the recipient of the resources. The `has_attr` predicate places the condition that the recipient, `To`, must be an organization; that is, if can create objects in its own namespace. The `begins_with` element is a utility that checks if the recipient has an HRN beginning with plc.eu. Finally, the contract uses another utility predicate, `is_onpath`, to avoid cyclic delegations by making sure it does not give permits that were actually passed from `To`, by looking at its delegation path.

5.5 Resource Specification

Security architects specify the types of resources in a federation. One common resource in testbeds and clouds is a virtual machine, which we represent with `nodespec` element. A node has properties such as host name, geographical region, disk, number of cores, CPU, memory, and bandwidth. Depending on the context, a nodespec can represent a requested resource by a principal or an already allocated amount of resource to be used for accounting. An organization keeps its overall capacity using such resource unit types, as well as, others' accounting information for their consumptions.

5.6 Accounting and Capacity

Every organization in a federation has accounting information that keeps track of the current resource usages of its local resources. Together with the capacity information, this is important for deciding whether or not to allow a future request. Accounting and capacity are used in defining allocation policies, which will be explained in detail in Section 5.7.

Accounting of resource allocations is done with respect to permits. A permit is a digital certificate that contains a *resource* statement, as discussed in Section 5.1. An allocation request is always performed with a permit parameter. Therefore, instead of the requester, resource consumptions are accounted with respect to the permit that was used to allocate resources. The

allocated resources are added to the account of the requester permit and also to its parent and ancestor permits. So, if a permit was delegated to others, then the allocations made by that child permit are also accounted to the parent. The parent of a permit is denoted by its recursive field, `Resource`, as seen in Table 5.1.

FPL elements `scheduled` and `capacity` tell how much resource is currently allocated to a permit and how much resource the permit can possibly allocate at an organization, respectively. `Total` field shows the total set of resources consumed by a permit and its descendants, and `NodeList` is the list of nodes allocated only by the permit itself. Capacity of a permit is obtained by running the constraints program over all the resource pool of an organization and obtaining the maximum possible resource set. The total amount of CPU, memory, and bandwidth are set to the `Capacity` field. We give the details of the implementation of these two elements in Chapter 7.

5.7 Resource Constraints

FPL resource constraint elements are used in a contract to express how much of a resource can be granted to principals. These constraints depend on the temporal dynamic state at an organization, such as current aggregate allocations of the requester, current amount of resources at the organization, time, calendar events, and so on. In our framework, such constraints are only based on the accounting and the capacity-related state that were discussed in Section 5.6. As more dynamic parameters are needed in the future, new corresponding FPL elements can be introduced by the security architects in the federation.

There are three example constraints seen in Table 5.1. Each of these rules places a restriction on the set of nodes, `NodeList`, that can be allocated by a permit. The `mn_limit` rule limits the number of virtual machines that can be allocated with a permit, and `fr_limit` sets a fraction limit, allowing a fraction of overall capacity of a permit to be granted to a child permit. Also, `exclude_machines` exempts a set of machines from allocation. The first two rules use the `Permit` parameter to retrieve the accounting information of `Permit` and place a limit based also on its previous allocations. The last rule specifies a set of host names `EList` that will be exempt from the requester and, therefore, should be avoided in a request.

In order to illustrate the definition of resource constraints that use accounting and capacity, we give the implementation of the `fr_limit` below. Lines 2–3 find the capacity of the parent permit.

Lines 4–6 find the total amount of resources, past and new, that permit wants to allocate. Lines 7–8 apply the fraction limit by checking whether the total amount is less than the fraction of the capacity.

```
1. fr_limit(Permit, NodeList, Limit) :-
2.      Permit = delegate(_, _, _, SourcePermit, _),
3.      capacity(SourcePermit, SourceCap),
4.      scheduled(Permit, _, Used_Resource).
5.      append(Used_Resource, NodeList, TotalNodes),
6.      nodes_resource(TotalNodes, Tot_Resource),
7.      tuple_times(SourceCap, Limit, SourceLimit),
8.      tuple_leq(Tot_Resource, SourceLimit).
```

5.8 Third-Party Resources

Contracts refer to third-party resources using the `permit_from` element. Permits are resource credentials that are obtained from organizations and delegable to others as third-party resources.

Permits are automatically generated as a result of contracts. Although FPL handles permit generation without admins seeing the complexity, here we illustrate how this works. For example, consider the contract given in Section 5.4, where plc.jp gives European organizations resources from South American organizations. Assume further that plc.jp already obtained some resources from plc.south.rnp. Then, the example contract can generate a permit, as below. Lines 2–4 contain the permit obtained from a third-party organization, plc.south.rnp, which is inserted within the encapsulating permit as a result of the `permit_from` statement in the example contract.

```
1. resource( plc.jp, plc.eu.inria, 206257450416,
2.           resource( plc.south.rnp, plc.jp, 126047660482, -,
3.                     (rules(C, NodeList) :- fr_limit(C,NodeList,0.75))
4.                   ),
5.           (rules(C,NodeList) :- mn_limit(C,NodeList,100))
6.         )
```

A permit is stored at its subject after it is generated. In our example, the permit signed by plc.jp will be stored at plc.eu.inria. After that, this permit itself can be referred to as a third-party

resource in the contracts that are or will be signed by plc.eu.inria. The FPL `permit_from` element is basically an interface to the permit store that is located at every organization, which will be further be explained in Chapter 7.

5.9 Basic Policy

Basic policy is composed of a principal's public and private keys, IP and HRN, and the root's name, address and key. These provide the fundamental knowledge base of a principal in order to participate in the federation. We have already used such policy elements in the definition of higher-level elements such as authentication policy. Basic policy is not shared with any other principal, but just used by the local policy. In CERTDIST terms, it is part of the private trust relations of an organization.

Chapter 6

Resource Discovery

This chapter explains the resource discovery and allocation piece of our federation framework, called CODAL, Contract Based Resource Discovery and Allocation. CODAL is built as a layer on top of FPL, hence CERTDIST, and uses policies and trust management while discovering and allocating resources. This is because resource discovery in a federation depends not only on resource availability, but also on federation policies; therefore, resource discovery must be policy-aware.

In general, the resource discovery problem is about locating a set of resources that fulfills the requirements of a program, service, etc. For example, in the area of computational testbeds, individual experiments have computational resource requirements, such as virtual machines with specific properties, that have to be mapped to a set of physical testbed resources. Previous work on resource discovery include SWORD [2] and Rhizoma [68], both deployed on PlanetLab, and aim at discovery of computational resources for developers running PlanetLab slices or cloud computing services.

Resource discovery in federations has some unique challenges compared to previous work that is designed for single administrative domain environments. In a federation, there can be numerous organizations, each willing to utilize resources of others. One problem is that nobody has a global view of all available resources and constraints on them. In a non-federated system, a user request is satisfied from the available resources, that are learned globally without any restrictions throughout the system. On the other hand, we assume that, in a federation, some policies and available resource information may be private to organizations and undisclosed. This requires a

more distributed request-to-resource mapping algorithm than the previous work. Another problem in federated discovery is discovering the organizations themselves, because organizations can join or leave any time in a large federation. An organization should be able to constantly learn the set of other organizations from which it can allocate resources. Such a mechanism depends on actual agreements between organizations and, therefore, has to utilize digital constructs that represent those agreements in order to be able to discover new organizations on the fly. Finally, in federated resource discovery, it matters who the requester is, as opposed to previous discovery systems. The discovery system should be able to authenticate and authorize by collecting and using digital certificates in order to constrain the allowed amount of resources for a requester. This requires high-level primitives that handle the access control functionality for the discovery system.

In the rest of this chapter we first describe the key pieces of resource discovery systems in previous work. Next, we elaborate on the federation-specific challenges for these pieces. Finally, we describe our discovery and allocation system, CODAL, which is built on top of the FPL layer in our federation framework. CODAL uses FPL elements to discover organizations, runs a distributed collaborative matching algorithm to find resources, and performs access control, again using FPL.

6.1 Resource Discovery Overview

Figure 6.1 shows three main components of resource discovery systems studied in the past: (1) resource specification R, given by a user of the discovery system, that is, the creator of a service or experiment that needs to use federation resources; (2) a resource information subsystem which gathers information about available resources A, a subset of which can be used to fulfill the request R; and (3) a "mediator" algorithm that fulfills the request R by using resources from A in order to arrive at a final resource mapping, FM.

6.1.1 Request Specification

The request specification R defines a set of resources, where each element in the set is called a *unit*. R has three main pieces: (1) Unit constraints define the properties of individual resource units, where each such constraint is about one unit only and does not relate to any others. For example, a user request in testbeds and cloud computing can be 30 virtual machines, or nodes. A user can place a constraint for each node, such as it should have at least 2.5 GHz of CPU, should

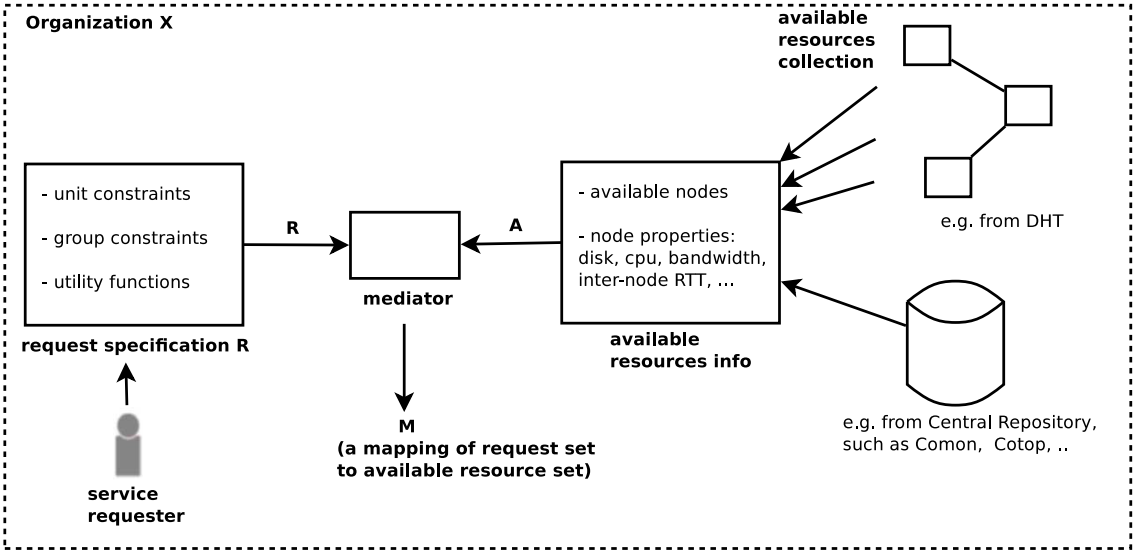


Figure 6.1: General terminology and pieces in a resource discovery system: request specification and available resource information are input to the mediator, which determines the physical set of resources to be allocated for the request.

be located at North America, and so on. (2) Group constraints define how units relate to each other. For example, the user may want 10 nodes that are not more than 50 msec RTT apart from any others. (3) A utility function specifies the preference among different possible sets of resource units that fulfill the unit and group constraints.

The SWORD system has unit constraints that are simple upper and lower bounds on the disk, CPU, memory, and so on, of a node, and group constraints, which are only pairwise constraints such as bandwidth between two nodes. On the other hand, Rhizoma is based on constraint logic programming (CLP) and, therefore, has much more expressive unit and group constraints, such as constraints involving any subset of nodes in a user request R. SWORD allows the user to specify a penalty value, and Rhizoma has cost and utility functions, again in CLP. The penalty, cost, and utility are used by the discovery system to choose among several possible matches for a request R.

6.1.2 Gathering Resource Information

Available resources and their properties are periodically collected by a subsystem that ensures all resource information is up to date. SWORD collects node properties and stores them in a DHT that runs in all nodes. Rhizoma retrieves resource information from preexisting repositories populated by monitoring services, such as CoMon, S3, and so on, as illustrated in Figure 6.1.

What is common for both discovery systems is that all resource information is collected without restrictions on the requesting user and is available to the mediator to process.

6.1.3 Mediator

The mediator gets the resource request specification and the available resources information as input, and maps the request to a set of physical resources, arriving at a final mapping FM. In SWORD, this mediator is called the optimizer, which is potentially run by any node in the DHT. Once a node gets a discovery request, the SWORD daemon gathers resource data and feeds it to the optimizer together with the node&group constraints and penalty function. Rhizoma uses the ECL$^{\text{i}}$PS$^{\text{e}}$ constraint solver as mediator, which processes the declarative logic rules to find a resource set that fits to constraints. The common part of both systems is that the mediator performs discovery in the light of all resource availability and request constraints. These are user specified resource constraints, as opposed to allocation policies specified by the resource owner, hence are not as complex as FPL policies such as ones involving credentials, capacity and accounting as described in Chapter 5.

6.2 Federation-specific Challenges

The federation-specific challenges of resource discovery are mainly caused by the existence of multiple autonomous administrative domains instead of one. We investigate these challenges in terms of three specific subproblems. First, a specific request should be able to be fulfilled from possibly multiple organizations, yet still satisfy the unit and group constraints. Second, new organizations should be periodically discovered by other organizations so that they learn possible new sources of resources. Third, there should be access control for discovering and allocating resources across organizational boundaries.

6.2.1 Allocating from Multiple Organizations

Because of the nature of a federated environment, not all constraints may be available to a single mediator to perform the request-to-resource mapping. For example, imagine an application in a federation that uses Rhizoma for discovering resources. The Rhizoma daemon would want to retrieve available resources from many federation participant organizations, because the application

possibly wants to span resource units from several organizations. We can call each such organization a *peer*, that is, organizations from which one can possibly allocate resources. It may not be practical to collect resource information from numerous peers, yet ensuring all are up to date until a final mapping, FM, is calculated that satisfies the user request. Even if the resource information is collected from peers, this may not be enough for FM to be valid because of peer-specific constraints. Peers can have their own preferences in allocating their own resources that are not always revealed to other organizations. For example, organization can have an internal policy to balance the allocations between its two datacenters. Therefore, in this thesis we assume there is a separate mediator at each organization, and a user request is fulfilled by asking one or more of such peer's mediators.

Satisfying group constraints of a resource request is particularly difficult when it requires resource units from multiple organizations. For example, think of a CPU-intensive parallel application, which has the requirement of 50 nodes with a sum of CPU that is more than 150 GHz. Suppose that none of the peers can give this much of its resources at once; therefore, it has to be allocated from more than one peer. This request is first sent to Org_1, which has only 15 spare nodes at the moment. Org_1 needs to have a way of knowing if group constraints would be satisfied if it granted these nodes, but it cannot process this particular group constraint since it does not have access to the remaining 35 nodes' properties. The request can be sent to another peer, Org_2 which gives 15 more nodes, and, finally, to Org_3, which gives the remaining 20 nodes. The group constraints can fully be satisfied only by the last mediator, the one at Org_3, which knows all units in the mapping. Such a scenario requires some mechanisms and/or assumptions on the group constraints that allow incrementally satisfying group constraints, so that the mediators arrive at mappings that help the subsequent mediators fully satisfy a group constraint.

6.2.2 Discovering Organizations

Organizations in a federation should be able to learn about other organizations from which they can possibly allocate resources. Since there is no limit on the federation size, there can be organizations joining the federation at any time. As an example, new universities have been joining PlanetLab even years after its establishment, from all around the world. The frequency of joins can be much more for cloud computing federations, where an individual organization is potentially any company with spare capacity. Therefore, each organization should perform discovery continuously

to keep its peer knowledge up to date. An organization may be directly known by a small set of organizations, rather than all other organizations in a federation; however, multi-hop allocation agreements may allow an organization's resources be used by others as soon as it joins federation. For example, in the contract example of Section 5.4, an organization in Japan grants to a European organization its resources that are located at any organization in South America. This means that if the European organization learns new organizations in South America in the future, they will also be available to the Japanese organization. Therefore, organizations should possibly be able to discover new sources of federation resources any time, arising from their existing agreements with other organizations in the federation. It is also possible that an organization is known by only a small number of others; therefore, it is possible that an organization can discover only part of the federation.

6.2.3 Access Control

Federation necessitates complex authentication and authorization even for discovering federation peers, in addition to allocation of actual resources. These require proofs about a requester's identity, attributes, and resource bindings, that may need many certificates with complex policies. For example, the authentication of a requester as one of the subjects of a contract requires a chain of identity certificates, as discussed in Section 5.3, or a slice membership policy requires a complex set of certificates, as discussed in Section 3.3.1. In addition, the implication of discovering organizations over indirect paths, as mentioned in Section 6.2.2, is that the authorization should be done over indirect paths as well. This again requires a chain of credentials that prove that the requester is eligible to obtain the resources, subject to specific constraints. All these functionalities are beyond the capabilities of resource discovery systems studied in the past, and requires a distributed trust management capability underneath.

6.3 CODAL

This section explains our federated resource discovery system CODAL (contract-based discovery and allocation system), which is designed to overcome the challenges discussed in Section 6.2. High-level components of it are depicted in Figure 6.2, where the organizational discovery utilizes FPL contracts and the querier and mediator utilize FPL security and allocation policies. We

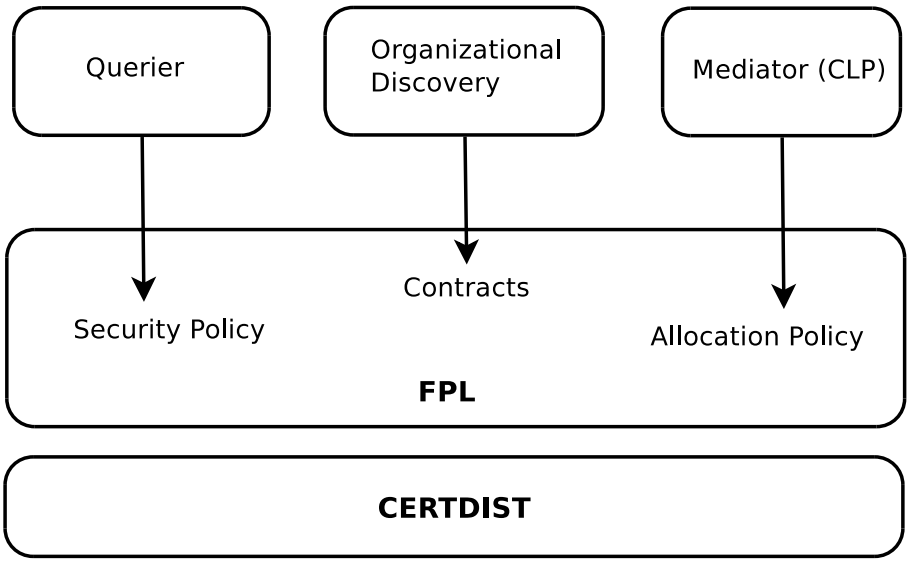


Figure 6.2: CODAL uses FPL contracts for discovering organizations and uses FPL security and allocation policies for access control.

first explain the querier, which performs collaborative allocation from many organizations in a federation. Next, we discuss the resource discovery and mediator in detail.

6.3.1 Querier

CODAL querier performs a collaborative allocation, which is depicted in Figure 6.3. Querier is invoked by a user submitting a resource request R to the querier at user's organization Q. Querier tries to fulfill the request from a set of known peer organizations P_i by obtaining a partial match PM_i at each step and feeding this partial match to the next query as parameter. Querier also passes the required digital certificates for authentication and authorization of Q at each P_i, which is not shown in the figure. Such certificates are obtained by using the FPL security policy elements.

Group Constraints

One challenge in allocating a request from multiple organizations is the group constraints, as mentioned in Section 6.2.1. Recall a group constraint is about a collection of resource units, rather than individual units, such as, four virtual machines must be in different geographical regions. We introduce a class of group constraints, called *subset-aware group constraints*, that are suitable to use in a collaborative allocation model. We can also think of this as a programming

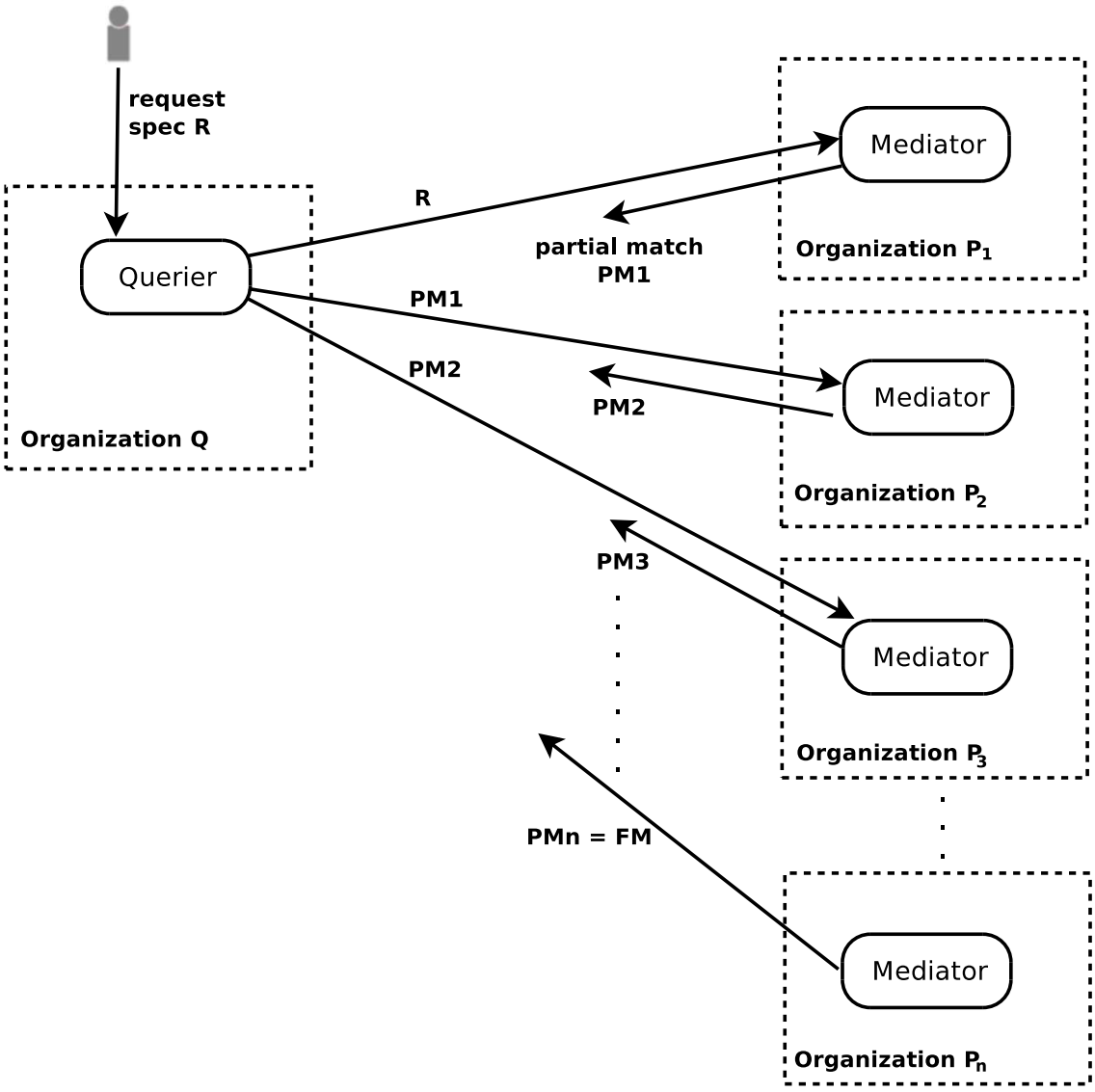


Figure 6.3: An organization Q allocates a resource request R from multiple peer organizations by sequentially querying each and obtaining partial matches PM_i, until the request is fully mapped to physical resources.

technique applied to specify the constraints. With subset-aware group constraints, it is possible to find a solution to R from multiple federation participant organizations. The basic idea behind such type of constraints is that the constraint function can operate on the subsets of the units in a request R, as well. Also, if the function satisfies a subset, it should ensure that there is a superset that satisfies the function. We first introduce the concept of subset-awareness, give examples of some constraints having this property, and show how to possibly alter existing constraints to make them subset-aware.

Here are some definitions about resource requests and constraints. A user resource request specification R contains an ordered set of N units. Each unit is U_i, $i = 1, ..., N$, where U_i is a k-tuple containing a unit's sequence number and properties. In the testbeds context, we can think of a unit as a virtual machine with properties such as CPU and disk, where each is a *field* in the tuple. Tuple fields are referred to as ${\mathrm{U}_i}^{\mathrm{j}}$, $j = 0, ..., k-1$, where ${\mathrm{U}_i}^0 = i$, showing the sequence number of the unit. k can be around 10 for nodes and depends on the level of detail in representing a resource unit. A tuple field can either be a *variable* or a *value*. For instance, in a request, the host name field of a VM can be set to a value if it is important in which physical machine the VM should reside; otherwise, it can be left as variable. Or, the memory field of a VM can be set to a variable, which is further used as a parameter in unit and group constraints. A *mapping* of a unit U_i is another tuple that is exactly same as U_i, except that all its variable fields are set to some values. We can represent a mapping of U_i with $\mathrm{U'}_i$, and there can be alternative mappings of the same unit, such as $\mathrm{U''}_1$. Two mappings X and Y are *distinct* if they are for different units—in other words, $X^0 \neq Y^0$.

Group constraints of R are defined to be a function $c(x)$, which gets as a parameter a set of distinct mappings and returns true or false. A *full match* FM is a set that contains mappings $\mathrm{U'}_i$, $i = 1, ..., N$ and $c(FM)$ is true. Note that since there can be alternative mappings of any unit, it is possible to find more than one full match for a request R, which can be referred as FM_i, $i = 0, 1, ...$ A *partial match* PM is a set of distinct mappings for only some of the units in R but still satisfies the group constraint. In other words,

$$PM \text{ is a } \textit{partial match} \iff 0 < |PM| < N \text{ and } c(PM) \text{ is true}$$

Again, there can be many partial matches for a resource request, which can be shown as PM_i, $i = 0, 1, ...$ A group constraint $c(x)$ at a request R is *subset aware* if there is at least one partial

match PM, and for all partial matches there exists at least one full match that contains it. In other words:

$c(x)$ is *subset aware* $\iff$

$(\exists PM$ s.t. PM is a partial match$) \wedge$

$(PM$ is a partial match $\implies \exists FM$ s.t. $(FM \supset PM) \wedge FM$ is a full match$)$

"There exists" in this definition is independent of resource availability; rather it is about theoretical existence of mappings that can result in a full match.

Subset-aware group constraints allow us to solve the problem of finding a full match in several simpler steps. This is achieved by finding a sequence of partial matches, each a subset of the next one, until a full match is found. Finding one particular partial match is easier and can be performed within a single organization. Then, another organization can use this partial match to find a full match or, if it cannot, to find another partial match containing first partial match. As a result, a full match can be found collaboratively by many organizations.

Examples of Group Constraints

One example of a group constraint from testbeds and cloud computing is that the set of virtual machines must reside at distinct physical machines. Specifically, the user request R contains $N = 100$ nodes and has the constraint $distinct(x)$, which checks $\mathrm{U_i^1}$ of the given mappings to be different. The $\mathrm{U_i^1}$ field contains the host name of the physical machine, such as a DNS name. $distinct(x)$ is subset-aware because if any subset of 100 nodes, say the first 5, are distinct, it is still possible to assign values to remaining $\mathrm{Node}_i{}^{\mathrm{j}}$, $i = 6, ..., 100$, $j = 1, ..., k$ which satisfies $distinct(x)$, which is a full match. Whether the federation has those remaining 95 nodes or not is a matter of availability, and actually a full match may not be found in the federation because of lack of resources. In a federation with sufficient resources, such a request can be matched from as many as 100 organizations, each organization but the last one finding a partial match and last one, a full match.

Another example is such that $N = 50$ nodes are required with the constraint called $cluster(x)$, which says nodes have to be in clusters of 5 nodes with maximum RTT of 5 ms within each cluster and minimum RTT of 50 ms between any two clusters. $cluster(x)$ for $N = 50$ is also subset-aware. Any subset of nodes that satisfy $cluster(x)$ will have one or more full clusters, which still allows a full match by properly selecting other nodes and their properties. Subset-awareness is always

defined in the context of a request R. For example, if $N = 51$, there would be no possibility of a full match in this example, even though there are partial matches; therefore, $cluster(x)$ is not subset-aware for $N = 51$.

Group constraints that are not subset-aware can possibly be altered in a way to make them subset-aware. For example, a network experimenter wants 20 nodes that communicate with a central server and places a group constraint called $bound(x)$. It requires the added total of some metric such as RTT or packet loss of each machine from the server be less than threshold T. $bound(x)$ is not subset-aware, because even though the sum of metrics for a subset of machines is less than T, it does not guarantee there exists a full match containing those machines, since any additional node has definitely a nonzero RTT, packet loss, and so on. It is possible, though, to write a subset-aware version of $bound(x)$. For example, $bound\_avg(x)$ ensures the average of the metrics of nodes is bounded by $T/20$, which effectively achieves $bound(x)$ for 20 machines and is subset-aware. For more complex group constraints, the programmer can develop their subset-aware versions by treating subsets with varying sizes differently and writing the conditions so that those subsets will lead to a full match.

Note that, theoretically there are cases where it is very hard or not possible to come up with a subset-aware equivalent of a group constant. For example, we could restrict the hash of the all node names concatenated to be a specific value. In that case, selection of the node set is not an incremental process, that is, finding a subset of nodes does not give any benefit for finding a superset. Nevertheless, we believe most actual group constraints can have a subset-aware equivalent.

6.3.2 Organizational Discovery

CODAL discovers new organizations in a federation through contractual relationships among organizations. A resource recipient in an FPL contract periodically queries the contract signer to get the updated resource sources and routes. We assume that an organization that is a subject of a contract, that is, granted resources, learns about this through out-of-band means. For example, in the contract example of Section 5.4, plc.eu.inria is aware that plc.jp gives resources to it; therefore, it can direct organizational discovery requests to it.

When a discovery request arrives at an organization, all contracts are executed to check whether the requester should be granted any resources. This is also depicted in Figure 6.2, where the

organizational discovery uses FPL contracts. If the caller conforms to any contract definitions, a set of permits are returned to the caller. A permit is a digital certificate expressing a set of resources determined by the contract it is generated from. For instance, the permit example of Section 5.8 can be obtained by plc.eu.inria, upon a discovery request sent to plc.jp. For efficiency, the discovery request can return only the content of the permit rather than the whole digital certificate, which can later be obtained if/when the recipient actually wants to use it to allocate resources.

6.3.3 Mediator

A resource request sent to an organization arrives at the mediator, as shown in Figure 6.3. CODAL's mediator performs "partial matching" of a request to available resources, meaning, it can find resources that only partly satisfy the request, so that the request can be fulfilled from multiple organizations. Mediator performs constraint logic programming (CLP), similar to previous work such as Rhizoma [68]. CLP allows processing of logic rules on system parameters, together with the range constraints on those parameters, in order to find a solution. Our mediator uses FPL allocation policies in order to impose policy restrictions set by the organizations, which is also shown in Figure 6.2. It also applies user-provided constraints such as upper/lower bounds on resource properties, such as memory, CPU, and disk. One particular problem in mapping resource requests to available resources is the computational complexity of the mapping algorithm. CODAL limits computational complexity by restricting the resource requests to have only a particular kind of subset-aware group constraints.

Computational Complexity

The combinatory problem of mapping of N node descriptions to M available physical nodes has high computational complexity. There are $\binom{M}{N}N!$ possibilities that can possibly satisfy node and group constraints in R; therefore, a brute-force method that tries each is impractical. Moreover, for collaborative matching, there are many more possibilities arising from the partial matches, as well, which can be given by the following expression:

$$\sum_{i=1}^{min(M,N)} \binom{M}{i}\binom{N}{i}i! = MN + \binom{M}{2}\binom{N}{2}2 + \binom{M}{3}\binom{N}{3}6 + ... \tag{6.1}$$

This is because a partial match is allowed to find any number of nodes for R instead of N. In other words, although the partial matching will try to find as many nodes as possible for R, it may fail and try to map fewer nodes to R. The expression adds up possibilities from each partial match i, where i number of nodes that are assigned physical resources. As a result, a partial match could result in matches of only 1 node or up to N or M, whichever is smaller.

In previous work, the computational complexity is bounded by avoiding checking every possible combination. This is done by using heuristics to find utility-maximizing combinations rather than strictly optimal solutions. For example, SWORD makes use of presorting and pruning on candidate nodes, and Rhizoma performs a hill-climbing approach to add or remove a few nodes to the solution at a time to improve utility. Similarly, in our case we do not seek an optimal solution, but try to find plausible solutions and confine the computational complexity.

We limit the complexity of a partial matching operation with the observation from Equation 6.1 that as the number of nodes to be matched is smaller, the complexity drops significantly. For example, if the mediator searches a partial match with only one additional node, the complexity is bounded by $O(MN)$. We first introduce a classification of subset-aware group constraints and show how complexity can be bounded in different levels for different classes.

The *degree* of a subset-aware constraint $c(x)$ in a request R is a measure of the additional number of mappings that should be added to a partial match in order to arrive at a larger partial match or a full match. For example, the degree of $distinct(x)$ is 1, but that of $cluster(x)$ is 5 because, to every partial match for $distinct(x)$ can be added just one additional mapping in order to arrive at another partial or full match. For $cluster(x)$, on the other hand, at least five mappings have to be added to a partial match for it to again satisfy the constraint.

Formally, degree is defined as follows. For a request R with group constraint $c(x)$, let PM_i be any partial match and M_i be a partial or full match that has the smallest number of mappings and is a proper superset of PM_i. Then, over all partial matches PM_i, the degree of $c(x)$ in R is $max(|M_i| - |PM_i|)$. Degree is inversely proportional to the maximum number of different organizations from which a request R can be fulfilled. For example, for $distinct(x)$ the maximum number of organizations is N, whereas for $cluster(x)$, it is $N/5$.

By limiting the degree of subset-aware group constraints, we can limit the mediator complexity. A partial match can be found with adding only "degree" number of additional mappings to another partial match. In addition, one can alter constraints to reduce their degree. For example, cluster(x)

can be changed to allow the partial matches to have incomplete clusters so that they can be completed by nodes from several organizations rather than nodes from only one organization. In that case, the constraint would be a more complex program, rather than simply checking cluster completeness. It is the programmer's task to decide which particular subsets of mappings have the chance of leading to full matches.

In our current mediator implementation, we assume degree-1 subset-aware group constraints; therefore, the computational complexity of the mediator is $O(MN)$. In addition, the mediator optimizes the resource selection in order to increase the organization's resource utilization and to minimize idle capacity. We provide a detailed explanation of the mediator in Section 7.3.4.

Chapter 7

Implementation

This chapter describes the implementation details of the federation framework, which consists of the previously discussed mechanisms, CERTDIST, FPL, and CODAL. We specifically explain the data structures and algorithms used to implement the individual components depicted in Figures 4.2, 5.1, and 6.2. In addition, we explain the interfaces used in interaction among components within a mechanism and also between any two mechanisms. This chapter is crucial to understanding the performance characteristics of the federation framework, which will be discussed in Chapter 8.

7.1 CERTDIST

CERTDIST is implemented with Prolog and Python languages, because these two have strong capabilities for different tasks, such as policy specification versus network programming and supporting diverse data structures, respectively. Functions implemented in different languages can call each other through network sockets, where the function name and parameters are encoded in JSON messaging format.

Figure 7.1 depicts the implementation details of CERTDIST. The individual components that were previously described in Figure 4.2 appear as subsystems with interfaces in this figure. Among these, the certificate storage subsystem, certificate retrieval logic, and a part of the remote query subsystem are written in Prolog; on the other hand, the event subsystem, cryptography library, DHT peer, and XMLRPC functionality of the remote query subsystem are written in Python.

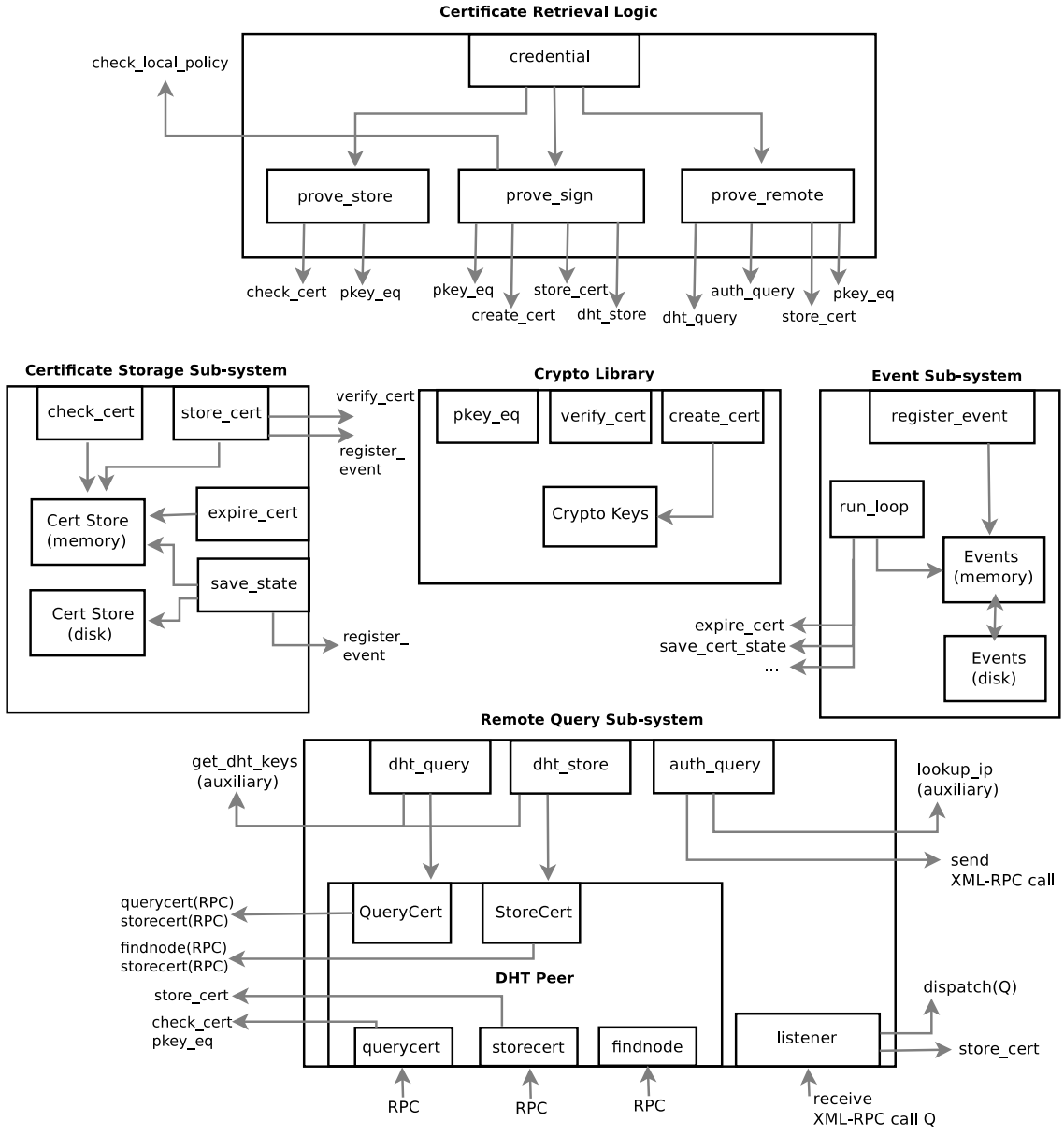


Figure 7.1: Interfaces and data structures comprising the main CERTDIST design components.

The figure does not show interprocess communication details.

A larger box in the figure represents a subsystem, and the smaller boxes within a subsystem represent individual functions or data structures that comprise the subsystem. The smaller boxes that touch an edge of the larger box are interface functions or event handlers that expose the functionality of this subsystem to others. An arrow from a function to another function or data structure shows a function call or usage of the data structure. We will explain the process flow of these operations in the next sections.

7.1.1 Certificate Storage Subsystem

A certificate storage subsystem keeps all the digital certificates that are obtained by the CERTDIST daemon, either signed locally or obtained remotely. Since signing or obtaining a certificate remotely is an expensive operation, the certificate store is the primary location for checking for certificates when they are needed by the policy. The subsystem consists of in-memory and disk data structures, interfaces to store and check for certificates, and event handler functions that remove the expired certificates from the store or synchronize the memory and disk states. Here we explain the implementation of these functions and data structures in Prolog.

Certificate Record

X509 certificates are stored in memory as Prolog facts and can be saved in a file as Prolog programs. These facts have the structure shown:

```
cert_directory(Id, Keys, Statement, SignerPubkey, Cert, ExpireTime)
```

`Cert` is a X509 digital certificate in PEM format, and the other fields are used for indexing certificates. `Statement` is the content of the certificate, excluding its expiration time or signature. It is a CERTDIST public trust relationship policy, which is a Prolog structure, such as the federation statements of FPL in Section 5.1. `Id` is an integer that uniquely identifies the content of a certificate, independent of its expiration time. It is obtained by getting the hash function of a canonical version of `Statement`. In some cases `Statement` can contain variables, such as when it contains programs expressing certificate conditions. In that case, any variables appearing in it are given a uniform format such as Var1, Var2, and so on, in order to keep the hash from being affected by specific variable names. `Id` is used internally to easily locate a certificate in the store, such as

when replacing a certificate with another having a later expiration time, or finding unsynchronized certificates between memory and disk. `SignerPubkey` is a self-signed certificate, where its public key is the signer of `Cert`. A self-signed certificate is the usual way to pass and store public keys. `ExpireTime` is the integer value corresponding to the date and time of expiration of `Cert`. `Keys` is a set of DHT keys, which can index a certificate. Similar to `Id`, each key is an integer value obtained from a hash function, but this time from a subset of type and fields of `Statement`. DHT keys were described in Section 4.2.2.

Interfaces

The interface `check_cert(S, K, Record)` is used to search certificates in the local certificate store. Given a statement `S` as parameter, it tries to unify it with `Statement` field of any of the records in the store. In addition, optional field `K` can be used to specify a key to narrow down the search, such that a record must have this key in its `Keys` field. `Record` is a Prolog variable that is populated with the result of the search, where each matched record is unified with `Record`, making the `check_cert` predicate true. The search is performed in the in-memory facts and is, therefore, handled by Prolog unification.

The interface `store_cert(Cert, SignerPubkey, K)` stores a digital certificate record in the store. The arguments are a certificate in PEM format, the public key of its claimed signer, and an optional DHT key. It performs this in three steps, as seen in Figure 7.1. First, `Cert` is verified not to be expired and signed by `SignerPubkey` by calling the crypto function `verify_cert`. Second, if the certificate is valid, any existing certificates with earlier expiration time are removed from the store, and the record is created with index fields and inserted. If there is already a newer certificate in the store, insertion is not performed, but `K` is added to `Keys` field of the existing record, unless it is there. The second step is all performed in a mutex so that the store is kept synchronized since many threads can perform operations on the certificate store. Finally, if the certificate was inserted, an event is registered, using `register_event`, to remove the certificate when it expires.

Event Handlers

There are two event handlers, `cert_expire` and `save_state`, that modify the memory and disk data structures. The former gets a certificate `Id` and removes it from the store by simply deleting

the fact from Prolog knowledge base. The latter reads all the records in the memory and writes them in a file; it reregisters the same event for a later time, so that certificates are saved periodically. In system initialization, the certificates are read from the file to memory and the expiration events are registered, which is not shown in the figure.

7.1.2 Event Subsystem

Event subsystem runs a continuous loop as a separate thread to check for events that are generated by the system or its users, and executes the corresponding event handlers. Events are registered with `register_event`, which gets their types, execution times, and parameters and places them in a queue to be executed later. Event subsystem is written in Python, which allows easy implementation of data structures like queues, and allows using classes to represent different event types and their execution parameters. Some events can be stored on disk to avoid loss of state in case of node failures or restarts.

System events are certificate expiration and saving certificates to file, both registered by the certificate storage subsystem, as discussed in Section 7.1.1. Applications can register events, as well, as in the case of spontaneous user requests. For example, a user requesting to create an experiment in a testbed will cause an event to be registered with an execution time as the current time. The event handler for this would be similar to the querier of CODAL, as discussed in Section 6.3.1, and will try to allocate resources for the user.

7.1.3 Certificate Retrieval Logic

Certificate retrieval logic implements the `credential` predicate of CERTDIST, as explained in Section 4.3.3, which retrieves a digital certificate that matches a statement. The complete prototype of this predicate is as follows:

```
credential(Statement, SignerPubkey, From, Method, X509Cert)
```

`Statement` and `SignerPubkey` are fields that define a credential, which are the content and signer of a certificate, respectively. Variables can appear as part of `Statement`, expressing partial knowledge about the credential, as discussed in Section 4.2.2. `From` is optional, and is a hint about where to retrieve a certificate from, such as an HRN for the principal in a federation. `Method` is an ordered sequence of a subset of letters l, s, a, and d, which specifies the sequence of methods used for

certificate retrieval: local signing, certificate store, authoritative, and DHT queries, respectively. The default value of this parameter is "slda". `X509Cert` is an optional parameter that is a variable, which is populated with the value of the retrieved certificate upon successful execution.

Certificate retrieval uses functions from other subsystems, such as certificate storage, cryptography, remote query, and DHT. As part of its processing, the `prove_store` predicate is first invoked to find a certificate record from the store by using `check_cert` interface. If not found, second, the `prove_sign` predicate checks the public trust relationship policies, such as federation statements in FPL, to unify `Statement` with any such policies. If found in the second step, a certificate is created that contains `Statement`, by making use of the interface `create_cert`. A newly created certificate is stored in local store and also possibly in DHT by using interfaces `store_cert` and `dht_store`, respectively. Only the signers store certificates at DHT, which avoids duplicate store operations and ensures the most up-to-date certificates are at DHT. Finally, the `prove_remote` predicate can perform authoritative or DHT queries to retrieve a certificate with a content that matches `Statement` and next stores this certificate in the certificate store, so that it can later be reused. All three methods will also check if the signer of the found certificate has the public key `SignerPubkey`, if this field was specified at the `credential` predicate. This is done by the `pubkey_equal` cryptographic function. The `credential` predicate will backtrack on possible certificates if it finds more than one candidate that matches `Statement` and `SignerPKey`.

7.1.4 Cryptography Library

We use the cryptography libraries from Python OpenSSL and M2Crypto packages to implement three functions: First, `pubkey_equal(PKey1, PKey2)` checks if two given self-signed certificates have the same public key, which involves extracting the field from the certificate and comparing. Second, `verify_cert(X509Cert, PKey, Statement, ExpireTime)` checks if a given digital certificate `X509Cert` is signed by a given public key `PKey` and not expired. If it succeeds, it extracts the content of the certificate into `Statement` and its expiration time to `ExpireTime` variables. Finally, `create_cert(Statement, ExpireTime, X509Cert)` creates a digital certificate `X509Cert`, which contains `Statement` encoded in its "subjectAltName" field, with an expiration time of `ExpireTime`. Certificate creation uses the private key of the principal whom the CERTDIST daemon represents, where the keys are generally stored as private trust relationships in CERTDIST.

7.1.5 Remote Query Subsystem

The remote query subsystem serves as the gateway by which CERTDIST daemon connects other CERTDIST instances over the network, sends queries, and listens for incoming queries. It exports three interface functions for other components to use that are in the form of Prolog predicates. A call to the interface function succeeds if the predicate evaluates to true, and its results, if any, appear as part of the predicate fields.

The first of these interfaces is `dht_query(Statement, SignerPubkey, ResultCerts)`, which performs a DHT query in order to find certificates that have content `Statement` and are signed by `SignerPubkey`. It first obtains DHT keys by calling the `get_dht_keys` auxiliary function with `Statement` and `SignerPubkey` parameters. Next, it performs a DHT query using the DHT daemon interface called `QueryCert`. The obtained certificates are placed in the `ResultCerts` list.

The second interface is `dht_store(Statement, SignerPKey, X509Cert)`, which gets information about a certificate and stores a record for it in DHT. It first obtains DHT keys and next stores the certificate at DHT with the keys by invoking the DHT daemon interface `StoreCert` for each key.

Finally, `auth_query(Destination, Query, Certs)` performs a remote authoritative query `Query` at another CERTDIST daemon located at `Destination`, passing any required certificates, `Certs`. The call is made using the Python XMLRPC libraries, where the channel can be configured either to SSL or plain unencrypted. It uses an auxiliary function first to look up IP, which must be implemented by the application-specific policy, as discussed in Section 4.2.2. `Query` can be an application or system defined operation. An example of the latter is the certificate retrieval operation, which is invoked by the certificate retrieval logic, by `prove_remote`, as depicted in Figure 7.1. An example of an application-defined operation is the individual request made by the CODAL's querier.

Any authoritative query from other CERTDIST instances arrives at the XMLRPC listener of the remote query subsystem. The listener first stores any input certificates for the query and next passes the query to the dispatcher, together with the public key used in the SSL connection. The dispatcher invokes the access control policies for the query located in the application-specific policy. The policy for answering remote certificate requests can perform application-specific checks and call `credential` with appropriate parameters to find certificates locally or remotely to answer the request.

7.1.6 DHT Peer

We use the Kademlia distributed hash table (DHT) and an implementation of it in Python, called Entangled DHT, in order to replicate, distribute, and retrieve certificates. DHT peer is part of the remote query subsystem, as shown in Figure 7.1, and provides routing for store/lookup operations, but not storage. All the data distributed through DHT are stored in our certificate store. Each CERTDIST instance runs a DHT peer as a separate daemon and communicates with it through network sockets. Distributing certificates in a DHT layer increases efficiency in certificate lookups, increases fault tolerance in the face of organizational servers' temporary failures, and also enables certificate retrieval even when the signer is not explicitly known. We first explain the usual operation of a Kademlia DHT system and next discuss our modifications to use within CERTDIST.

Kademlia Overview

In a Kademlia network every peer has a unique ID and keeps two main data structures, a *routing table* and a *data store*. The routing table keeps neighbor peer ID and addresses, and the data store keeps key-value pairs. Kademlia replicates a key-value pair at k number of DHT peers, where k is a system parameter, which is 8 by default.

Kademlia has three main protocol messages or RPCs. First, *store(K, V)* creates an entry in a peer's data store with key-value pair K, V. Second, *find_node(K)* returns the neighbors with ID closest to key K in a peer's routing table. Finally, *find_value(K)* returns the value V if the pair K, V exists in the data store, else returns the closest neighbors, exactly like *find_node(K)*.

Kademlia has two main high-level functions that use the protocol messages. First, *StoreValue(K, V)* performs a series of *find_node(K)* RPCs to find k nodes with closest IDs to key K and stops when no more closer nodes are found. Next, it executes *store(K, V)* RPCs at this final list of nodes. Second, *GetValue(K)* similarly performs a series of *find_value(K)* RPCs to find closer nodes, until a node returns a value V or no more closer nodes are found. If the result is found, it finally performs *store(K, V)* at the node with ID closest to K that did not have the result.

CERTDIST-related Modifications

Each CERTDIST instance runs a modified version of a Kademlia peer daemon. We introduce additional RPC functions that communicate with the local certificate store. Second, we implement

additional higher-level operations specialized for certificate lookup/store that use previous and new RPC functions.

We implement two additional RPC functions. First is *storecert(K, V)*, which is similar to *store(K, V)* but interconnects with CERTDIST and inserts V to the its certificate store. V is always a list of certificates, where each is stored by indexing with the key K, using the `store_cert` interface function. If insertion to the certificate store was successful—that is, at least one record is created having key K—the key K is also inserted to the Kademlia data store, so that it can be found in later DHT query operations. We use the Kademlia data store only to check if a key exists in the peer; we actually never use a value from its default data store but always store and look up values in the certificate store.

The second RPC function is *querycert(K, Q)*, which is similar to *find_value(K)* but has an additional parameter, Q, which is a query to be asked to the certificate store. The format of Q is again a Prolog predicate: `dq(K, Statement, SignerPKey, ResultCerts)`. K appears two times in call parameters, where the former is seen only by the DHT code in Python, used while checking if this key is in its data store. If not found, it will return neighbor peers, as *find_value* does. If found, *querycert* invokes the query `dq`, which is not depicted in the figure and is handled by Prolog. The processing of `dq` is such that it first uses `check_cert` interface function to check for certificates with content `Statement` and having key K. Next, it verifies that it is signed by `SignerPKey` using the `pkey_eq` function. The list of matching certificates is assigned to the `ResultCerts` variable. Function *querycert* returns the final `dq` structure as the result to the caller.

Function *querycert* keeps the key space synchronized between the DHT data store (Python) and the Prolog CERTDIST certificate store (Prolog). Certificates can expire and be removed from the certificate store after a time; therefore, keys that no longer exist have to be removed from DHT's data store, as well. Function *querycert* checks keys at the certificate store and deletes them from the DHT's data store if necessary, whenever a query returns no result.

We implement two high-level functions, *StoreCert(K, V)*, and *QueryCert(K, Q)*, which are similar to *StoreValue(K, V)* and *GetValue(K)*. They use both the newly implemented functions and some of the Kademlia RPC functions. First, *StoreCert(K, V)* performs a series of *find_node(K)* RPCs until k peers are found and next performs *storecert(K, V)* RPCs at each node in order to store certificates. Second, *QueryCert(K, Q)* performs a series of *querycert(K, Q)* RPCs until a node returns a result. Again, if a result is found, it is stored at the node with ID closest to K that

did not have the result, using *storecert(K, V)* at that node.

QueryCert(K, Q) does not ensure all possible certificates that match to a query Q will be found, but at least one, if it exists in DHT, will be found. This is because it stops sending RPCs as soon as one node returns a result, which may or may not have all relevant certificates stored in it. While this functionality is enough for the current state of the policies in the federation architecture, we see it as future work to implement more sophisticated *QueryCert(K, Q)*, which can retrieve results from many peers and combine them to arrive at a comprehensive result set for a query.

7.1.7 Application-specific Components

There are application-specific components of CERTDIST, as mentioned in Section 4.2: access control policies, trust relationship policies, auxiliary functions, and event handlers. Since these are application specific, their implementation is not part of CERTDIST; however, we talk more about these pieces in implementation of FPL and CODAL in later sections. Hence, Figure 7.1 does not show such components but still depicts some calls to these components.

For example, a listener invokes an application-specific `dispatch` that calls an access control policy. Applications can use CERTDIST by implementing a set of access control functions, placing their prototypes in a dispatcher so that they can be invoked upon an authoritative query.

Certificate retrieval logic consults the application's public trust relationship policies while signing a new certificate, as depicted by the `prove_sign` interface invoking the application-specific `check_local_policy` function.

Similarly, `dht_store` and `dht_query` interfaces obtain the DHT keys by invoking `get_dht_keys`; also `auth_query` looks up IP address of the query destination by calling address resolution primitive `lookup_ip`. DHT key and IP lookup are part of the application-specific auxiliary functions of CERTDIST.

Finally, there are event handlers written by programmers that handle application-specific events, possibly collecting certificates first and then invoking the `auth_query` interface of the remote query subsystem, which is not shown in this figure.

7.2 FPL: Federation Policy Language

This section explains the implementation of specific FPL elements that are shown in Table 5.1, and depicted in Figure 5.1. These elements are implemented in Prolog and utilize CERTDIST primitives such as certificate retrieval and data structures such as conditional credentials, as mentioned in Sections 5.1 and 5.2. Among these elements, we have already explained how to write security and allocation policy FPL elements and provided examples of FPL contracts in Sections 5.3, 5.7, and 5.4, respectively. In this section, we focus on the implementation of lower-level FPL elements, that is, the statement proofs, accounting and capacity elements, and primitives to express third-party resources.

7.2.1 Statement Proofs

Federation statements are proven by the `certify` predicate of FPL. As explained in Section 5.2, it has three input arguments, `Statement`, `From`, and `Method`, and a result argument, `CertList`. We give part of its implementation, excluding its optional arguments, for simplicity. This illustrates usage of Prolog built-in predicates and the CERTDIST built-in `credential` predicate in implementing FPL elements.

```
certify(Statement) :- arg(1, Statement, SignerHRN),
                      not(var(SignerHRN)), !,
                      authenticate(SignerHRN, SignerPKey),
                      From = SignerHRN,
                      credential(Statement, SignerPKey, From).
certify(Statement) :- credential(Statement, SignerPKey),
                      arg(1, Statement, SignerHRN),
                      authenticate(SignerHRN, SignerPKey).
```

A statement can be proven in two ways, shown by two definitions of the `certify` predicate, where the first handles the case where signer of a statement is known and the second handles the case otherwise.

In the first case, the Prolog built-in `arg` predicate gets the first parameter from a given predicate. In our code, this predicate is a federation statement, `Statement`; therefore, the first field

is the principal who utters it, which is `SignerHRN`. If `SignerHRN` is known, that is, it is not variable, it is authenticated first in order to find its public key, `SignerPKey`. Then, we can call the CERTDIST built-in `credential` predicate to prove `Statement` with that public key. In addition, we provide `credential` with a destination, `From`, to which it can direct its authoritative query.

In the second case, the signer is not known; therefore, the only way a certificate can be found is to check it in DHT. The call to `credential` will query DHT to see if there are any certificates with contents matching `Statement`. If it executes successfully, it will have a value at `SignerPKey` field. The last `authenticate` predicate will verify that the statement is uttered by `SignerHRN`.

In its full implementation, the additional optional fields for `certify` are passed to the `credential` predicate, which can help the proving process. For example, providing a `From` field will allow the second part of the proof to perform authoritative queries as well. The full implementation of `certify` considers any combination of the mandatory and optional fields and calls `credential` with appropriate parameters one or more times until the certificate is found.

7.2.2 Accounting and Capacity

Accounting and capacity FPL elements are explained in Section 5.6 as crucial pieces for expressing allocation policies. In this section we will explain the implementation details of those language primitives, specifically, `scheduled` and `capacity` FPL elements.

The **scheduled**(Permit, Nodes, Total) element keeps the set of resources allocated by a permit and its descendants. Every organization stores such Prolog facts, one per permit that has been used to allocate resources at the organization. These facts are updated each time a resource allocation happens in the organization, specifically by the **account_nodes** element. They are removed when an allocation expires. Permits have an hierarchical structure, where permits delegated from another are children of that permit, as mentioned in Section 5.6. Allocations performed by a descendant are accounted for the ancestors as well.

For example, consider the example permits that were mentioned in Section 5.8.

$P1$ = **plc.south.rnp** gives to **plc.jp local** resources subject to **max 75%**

$P2$ = **plc.jp** gives to **plc.eu.inria P1** resources subject to **max 100 machines**

Let's say first plc.eu.inria allocated $Node_A$ by using $P2$ at plc.south.rnp. After that operation, the accounting structures at plc.south.rnp organization would look as follows:

`scheduled`($P1$, [], [$Node_A$])

`scheduled`($P2$, [$Node_A$], [$Node_A$])

$P1$ is the parent permits and therefore keeps $Node_A$ in its $Total$ list. Next, let's say plc.jp allocates $Node_B$ by using $P1$ at plc.south.rnp. The accounting structures will be updated to the following state.

`scheduled`($P1$, [$Node_B$], [$Node_A$, $Node_B$])

`scheduled`($P2$, [$Node_A$], [$Node_A$])

These Prolog facts are stored in memory and also in the disk for persistence. Access to these facts is performed in a mutex, so that they are never in an inconsistent state caused by a multithreaded operation.

The second FPL element, **capacity**(Permit, Capacity), is a predicate that computes the maximum amount of resources that a particular permit can allocate at an organization, as constrained by the policy program in the permit. This predicate is dynamic since the policy program can contain temporal parameters such as current set of organizational resources, time, and so on. The `capacity` element runs a binary search algorithm in order to find the maximum amount of resources that permit's policy allows. For example, it first tries to allocate all the memory/CPU/disk of a machine, say, amount X. If policy allows, it is added to the computed capacity; otherwise $X/2$ is tried, and so on. This is performed for all nodes in the organization. Since capacity can result in different values at different times, this computation can be done every time a permit is used to allocate at an organization, or the result of the computation can be cached for a short time for efficiency.

7.2.3 Third-Party Resources

In this section we explain the implementation details of the FPL element that is used to refer to third-party resources, namely, **permit_from**(From, Permits), which was also discussed in Section 5.8.

Every organization has a permit store, which stores the permits issued by other organizations to this organization. Similar to certificates in the certificate store, permits are deleted from the store when they expire. They are kept in memory and also persisted on disk, in case the node has to restart. The `permit_from` element is an interface to the store that returns permits originating from a particular organization. This interface is used by organizational admins in writing contracts, specifically to express third-party resource delegations.

The permit store is populated continuously at an organization X as long as there are other organizations that have contracts granting resources to X. First, other organizations generate the permits from those contracts; second, those permits are obtained remotely by X and stored at its store. We talk about the mechanism that realize the second step in Sections 6.3.2 in discovery and allocation and its implementation in Section 7.3. The first step, on the other hand, is basically performed with the following Prolog rule.

```
delegate(Signer, To, ID, Resource, Constraints) :-
                          myHRN(Signer),
                          contract(ID, T1, T2, To, Resource, Constraints),
                          timenow(T), time_between(T, T1, T2).
```

This tells us that if the contract has a valid date and time, a permit, which is a `delegate` statement, can be issued to any organization that will satisfy the contract conditions (`Conds` of Section 5.4). This means a contract can give rise to one or more permits, which are possibly to different recipients and for different resource sources.

7.3 CODAL: Resource Discovery

In this section we give the implementation details of CODAL. Specifically, we explain the resource request specification, organizational discovery, querier, and mediator algorithms that were talked about in the CODAL design in Chapter 6.

7.3.1 Resource Request Specification

Users specify the requested resources, unit, and group constraints with a data structure called RSPEC. Implementation of RSPEC reflects the goal of finding at least one possible solution rather than optimizing among multiple possible solutions. This is because RSPEC can contain large request sizes, which possibly need to be matched from multiple organizations, which is a big challenge itself. Therefore, RSPEC does not contain complex utility or cost functions, as in the case of previous work on discovery systems. We illustrate RSPEC in the context of computational testbeds and use resource units as virtual machines, namely, nodes. A node is represented by a `nodespec`, which we discussed in Section 5.5; nodespec has seven main properties: host name,

geographic region, disk size, number of cores, CPU in terms of GHz, memory, and bandwidth. The node and group constraints are built by using those seven parameters from each node. One final field of nodespec in RSPEC context is `NodeAssigned`, which is 0 or 1, indicating whether or not this nodespec is yet assigned to a physical resource.

```
rspec(NodeList, Constraints)
nodespec(Name, Location, Core, Disk, Cpu, Mem, NodeBw, NodeAssigned)
distinct(NameList)
```

The `Constraints` field of an RSPEC is a list of lists of length $N + 1$, where N is the length of `NodeList`, list i keeps the node constraints for node i node, where $i = 0, 1, ..., N - 1$, and the final list keeps group constraints. Node and group constraints are Prolog expressions, such as predicates from common schema. An RSPEC is said to be satisfied, or fully matched, when all nodespecs in it are assigned and `Constraints` hold true.

The `distinct` predicate is used in our schema as an example subset-aware group constraint. It simply gets the list of names of nodes and return true if they are distinct. This is a natural user-side policy, since testbed users generally choose to allocate one virtual machine from each different physical machine. It is easy to generate such new elements to the schema because Prolog is a declarative and expressive language.

Below is an example RSPEC, which contains two nodes. This RSPEC denotes a request which is yet to be fulfilled, since nodes are not assigned yet. The per-node constraints are represented by the first two lists, which are basically lower bounds on the node resource metrics. The group constraints contain one predicate, which makes up the last list of the `Constraints`.

```
[ [ nodespec(Name1, 1, C1, D1, Cpu1, M1, Bw1, 0),
    nodespec(Name2, 2, C2, D2, Cpu2, M2, Bw2, 0)],
  [ [ C1>=1, D1>=5, Cpu1>=0.5, M1>=100, Bw1>=1000 ],
    [ C2>=2, D2>=10, Cpu2>=1, M2>=500, Bw2>=10000 ],
    [ distinct([Name1, Name2]) ] ] ]
```

7.3.2 Querier

CODAL's querier performs a series of allocation requests to the peer organizations, as mentioned in Section 6.3.1, which we call collaborative allocation. These peer organizations are basically

the origins of the permits in the permit store. As discussed in Section 7.2.3, the permit store is populated as a result of contractual agreements between organizations, where each permit indicates resources either from the direct contract partner or a third-party source.

Querier performs queries in a specific sequence special to the RSPEC in order to minimize the time to fulfill the request. For example, if the RSPEC needs nodes from North America, then a permit having an origin of a North American organization is used first to perform a query to that organization. More complex querier implementations are possible as future work, such as parallel remote calls, so that resource sets that are independent in terms of group constraints can be acquired in parallel.

7.3.3 Organizational Discovery

Organizational discovery is achieved by contract partners periodically querying each other, as discussed in Section 6.3.2. The receiver of such a query executes the contracts with the Prolog rule that is mentioned in Section 7.2.3. The resulting permits obtained from executing the contract are returned to the caller.

Both the caller and the callee organizations perform some optimizations so that a query does not cause overhead for the callee. The execution of contracts that give third-party resources can be expensive, because the complexity increases linearly with the number of permits in the permit store. If the store size is large, both the time to compute and the size of the resulting delegation permits can be large. To solve this problem, the callee limits the maximum number of permits that are returned per call by stopping the execution once that number of permits is reached. Also, the caller passes the signatures of the set of permits that it already obtained from previous calls so that the callee does not return these permits again. Finally, the caller can adjust the periodic discovery call frequency by deducing from the response times of the callee whether or not it is overwhelmed by many requests.

7.3.4 Mediator

The RSPEC is mapped to available resources using constraint logic programming (CLP), as discussed in Section 6.3.3. In this section we explain its implementation in Prolog, the pseudocode of which is shown in Figure 7.2, with the `match_resources` predicate.

The algorithm runs a loop with maximum N iterations, where N is the number of nodes in

```
1.  match_resources(Permit, RSPEC) :-
2.        Used ← [],
3.        get_group_cons(RSPEC, Group_Cons),
4.        while there are unprocessed nodes in RSPEC:
5.              select_nodespec(RSPEC, Node, Node_Cons)
6.              process_node(Node, Node_Cons, Group_Cons, Used).
7.
8.  process_node(Node, Node_Cons, Group_Cons, Used) :-
9.        match_to_instance(Node, Node_Cons),
10.       match_to_resource(Node, Node_Cons, Group_Cons, Used),
11.       can_alloc(Permit, [Node | Used]),
12.       set_assigned(Node),
13.       Used ← [Node | Used].
14. process_node(Node, Node_Cons, Group_Cons) :- true.
15.
16. match_to_instance(Node, Node_Cons) :-
17.       for i=0 to 4:
18.             member(Node[i+3], Range$i$),
19.             run_constraints(Node_Cons),
20.       Node ← enhance_match(Node, Node_Cons).
21.
22. match_to_resource(Node, Node_Cons, Group_Cons, Used) :-
23.       get_ordering(Node, Used, AvailNodes),
24.       member(Candidate, AvailNodes),
25.       for i=0 to 4:
26.             Candidate[i+3] > Node[i+3]
27.       Node[0] = Candidate[0],
28.       append(Node_Cons, Group_Cons, Cons),
29.       run_constraints(Cons).
30.
31. run_constraints([Con|Cons]) :-
32.       catch(Con, _, true),
33.       run_constraints(Cons).
34. run_constraints([]).
```

Figure 7.2: The mediator algorithm in Prolog pseudocode

RSPEC. At each iteration it selects one nodespec from RSPEC that is not assigned to a physical machine and not processed before and tries to find a mapping for it. The `process_node` predicate tries to map a nodespec to a physical node in three stages: First, `match_to_instance` finds a virtual machine (VM) instance type suitable for the nodespec; second, `match_to_resource` selects a physical organizational machine to host this VM; finally, `can_alloc` runs the allocation policies to check if this resource is allocatable by `Permit`. Since the second and third stages can be performed M times, the overall complexity becomes $O(MN)$, where M is the number of available physical machines.

The `match_to_instance` predicate uses Prolog backtracking at line 18 to find a VM fitting to the nodespec. Each Range_i, $i = 0, ..., 3$ is an ascending list of possible values for disk, CPU, memory and bandwidth, respectively. Node[k] is shorthand for field k of nodepsec Node, where $k = 0, 1, ..., 6$; therefore, Node[3] is the disk, Node[4] is the CPU of node, and so on. Minimal values are matched for all node properties and verified to satisfy the node constraints by running `run_constraints`. This is a predicate that simply runs Prolog expressions, such as inequalities, or predicates from schema. Once a nodespec is mapped to a VM instance, it is further refined to minimize the individual VM properties using `enhance_match` at line 20. This function performs a binary search for each node property i in order to find the least value that satisfies the node constraints. For example, if Node[5] (memory of the node) is matched to $\text{Range}_2[3]$, then we perform a search in interval ($\text{Range}_2[2]$, $\text{Range}_2[3]$). Binary search stops when some targeted granularity is met. There are different granularity values for each of the four node properties. If $\text{Range}_i[0]$ was matched, then no more enhancement is possible. We find match enhancement useful in situations where resources are scarce; for example, in our evaluation using PlanetLab in Chapter 8, RSPECs need to be matched very closely so that a high system allocation rate is achieved.

The predicate `match_to_resource` backtracks over available physical nodes of the organization, at line 24, to find one that has enough resources to contain the VM. The order of this list of physical nodes, `AvailNodes`, is generated by the `get_ordering` predicate, which performs "min-occupy-first" ordering. The "occupy" value for a physical node P that is a candidate to host a VM V is defined as follows. First, the occupy value of a node property, c_i, such as disk, CPU, memory and bandwidth, is the ratio of the current available amount to the available amount if V was placed in P, that is, $\text{P}[i+3]/(\text{P}[i+3] - \text{V}[i+3])$. The occupy value of a physical node, c, on the other hand,

is the maximum of occupy values of all its properties; in other words, $c = \max(c_i)$, $i = 0, ..., 3$. The `get_ordering` predicate orders the available nodes by ascending occupy values. This ordering stems from the observation that a physical node is not preferable to host a VM if it will use most of any of the node's resources. For example, if a VM with a high CPU requirement is mapped to a physical machine that can barely provide the required CPU, this mapping would leave all the remaining resources of the node, such as bandwidth and memory, useless, since there will not be any more possible VMs because the CPU is depleted. We call this the singular deprivation problem. We saw in our experiments that this problem can be significantly avoided by ascending occupy value ordering of nodes. Once a physical machine is assigned to the VM, the VM becomes fully mapped by setting its hostname at line 27, and finally group and node constraints can be run on the RSPEC at line 29.

The `can_alloc` predicate at line 11 runs the policies embedded inside a permit, as Prolog predicates. If all three steps successfully go through, this means a partial matching is found, which contains only one additional mapped node. If any of the node or group constraints or allocation policy are not satisfied, then this node is just skipped at line 14, and the next node is selected, and so on, until no more nodes are left in RSPEC. The algorithm can result in as many as N or as few as 0 mapped nodes in RSPEC.

Chapter 8

Evaluation

We evaluate our federation framework mechanisms, CERTDIST, FPL, and CODAL, to show that the framework can build federations with complex policy and many participants. The challenge of evaluating a federation framework is that building a large-scale federation from scratch would require a collective effort from many parties, such as many organizations adopting and using the framework. Instead, we perform an emulation of a realistic federation setting by using real system state and logs. We derive federation information, such as participant organizations, their resources and users, resource request patterns, and so on, from the current state of real systems and construct a federation setup with them. This approach is useful not only to show that our architecture and mechanisms are running properly and are efficient, but also to provide a realistic federation instance, which can be a candidate for existing systems to migrate to over time.

Our evaluation is in the context of testbeds, specifically the PlanetLab (PL) federation. PL consists of hundreds of universities and research institutions, called sites, each having its own users and resources, and, therefore, provides an ideal candidate for a large-scale federation. Today PL is partly federated, where a small number of autonomous regional subfederations such as PLE and PLJ branched out in the last few years and where each also contains many sites in their region. As mentioned in Section 2.2.1, this federation is based on direct trust relations and has few autonomous organizations. The sites are not fully autonomous, since their users and resources are administered and allocated by the regional PlanetLab. In our emulation, we treat each site as a separate autonomous organization in a large-scale federation. Since we have access to PL usage traces and the PL central (PLC) database, we can rerun user-request patterns as if they occur in

the fully fledged federated environment and observe federation processing.

PlanetLab serves not only as the source of our federation setup and traces to derive experiments, but also as the platform to run experiments on. Our experiments are run at PlanetLab by deploying the code, federation configuration, and usage traces at hundreds of machines located around the world. In the next sections we first explain our experimentation setup, and then talk about the specific experiments. Our results show that system performs in desired ways in terms of organizational discovery, resource allocation success rate, remote query times and success rates, and the effect of DHT on reducing load on organizational servers and improving certificate retrieval times. We published some of the evaluation in this chapter [59], where we used PlanetLab logs to evaluate CERTDIST by emulating API and machine login events. We will explain these in detail, as well as CODAL's resource allocation.

8.1 Experiment Setup

For our experiments we first emulate authentication and contract topologies that can allow authentication and resource exchange in the system. Next, we emulate user resource request, API operation, and machine log-in events that trigger operations requiring authentication and authorizations. In addition, per-organization resource sets are generated, on which the user resource request events operate.

8.1.1 Authentication Topology

We assign human-readable names (HRN) to any site, user, node and slice, which originally have flat names in PL. We designate one root authority called plc and four other top-level regional authorities, plc.south, plc.eu, plc.asia and plc.jp. Any site in PL is assigned an HRN under the top-level authority of its geographic region. The slice, user, and nodes are assigned HRNs under their site's HRN. For example, a site called "inria" in PL becomes plc.eu.inria, and a slice called "inria_demo" becomes plc.eu.inria.demo. Since we have access to the PLC database, we automatically obtain the geographical location of PL sites and place them under the right top-level authority. This database also includes the user, node, and slice records, which are again automatically given HRNs depending on the site to which they belong.

8.1.2 Machine Assignment

Our federation setup runs as a PL experiment spanning all machines and, hence, is itself a slice that uses a portion of resources at each machine. We select one unique machine to host each organization in the federation, where each such machine has a CERTDIST daemon that runs CODAL. Since there are about 350 sites in PL and more than 450 live machines, we have enough machines to emulate all organizations. Each organization machine is selected to be geographically proximate to it, so that it best represents the behavior of queries performed by and at its CERTDIST daemon in a real federation deployment. In addition to proximity, we also take into account machine load, such that the busiest organizations in our experimentations, which are the top-level authorities and the organizations performing or receiving many allocation requests, are assigned to the machines having the least system load in PlanetLab, according to data provided by the CoMon node-monitoring tool [49] statistics. These arrangements are enough to run our experiments to produce meaningful results, although they do not exactly represent a real federation deployment, where all organizations would have dedicated servers located in their particular locations. The effects of this on our results are discussed later.

8.1.3 Contract Topology

We generate contracts among organizations, which allow them to allocate resources from one another in our emulation. The layout of overall contracts in a federation is called the contract topology. Since there are no data in PL from which we can emulate federation contracts, we generate all contracts in our federation setup from scratch. Nevertheless, we generate contracts in such a way that any organization can possibly allocate resources from any others, which is parallel to the usage goal followed by the current PlanetLab. As a result, we believe our contract topology is reasonable, allows us to test resource allocation at a large scale, and can serve as a candidate topology for future federations.

Figure 8.1 shows the "linear" contract topology, which contains bilateral contracts between geographically adjacent top-level authorities. For example, resources at plc.jp are first known only to and can be acquired by plc.asia. Only through contracts of plc.asia with plc.eu and its subsequent contracts with others can those resources get to be known and used in all organizations around the world. Obviously, it would be less efficient for plc.south, plc, plc.eu, and plc.asia individually to make separate agreements with plc.jp to use its resources. This topology both

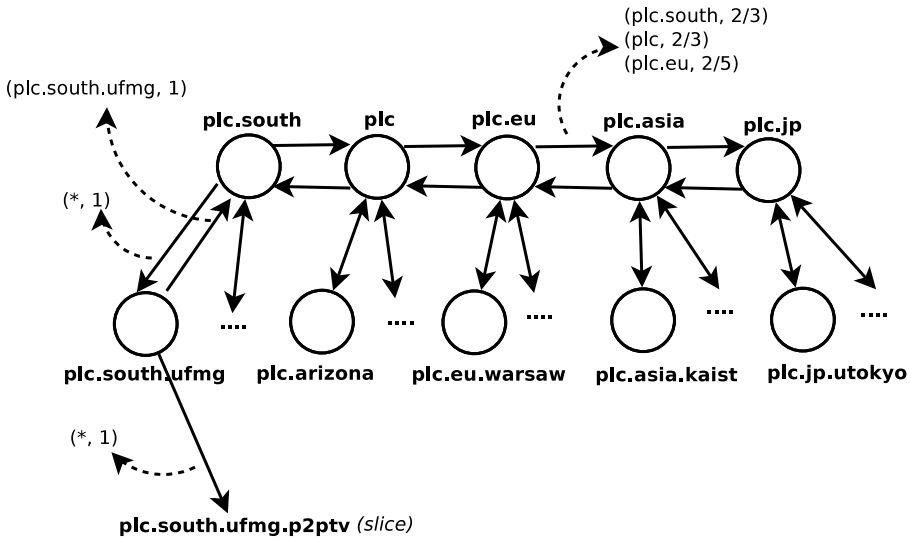


Figure 8.1: Linear contract topology contains contracts between top-level authorities, between top- and low-level authorities, and between authorities and their slices.

contains a small number of contracts and also allows a trust path for resource allocation from any organization to another in the federation. The figure also shows the contract relations within individual geographical regions. All authorities have contracts with the top-level authority such that they use the top-level authority as a proxy to obtain resources from other organizations and publish their own resources to other organizations.

In our emulated contracts we use only the `fr_limit` contract rule, which grants a fraction of organizational resources. This rule is independent of the resource size of an organization; therefore, it is simple to generate automatically and also can effectively distribute resources among many principals. For simplicity, we represent such a contract with the tuple (*ResourceSource*, *Fraction*). For example, in Figure 8.1, one of the contracts from plc.eu to plc.asia is (plc.south, 2/3), which actually converts to a contract rule in our experiments, one that has a `permit_from` predicate in its conditions (RHS). This contract grants resources from any organization at South America, with a fraction limit of 2/3. In other words, plc.eu grants to plc.asia 2/3 of any resources that it can possibly obtain from organizations with HRN plc.south.*.

The fraction values in the contracts are selected according to a heuristic of fair sharing among geographical regions. For example, plc.eu grants 2/5 of its own resources to plc.asia because the number of regions at plc.asia's branch are 2 and the total regions are 5. On the other hand, it grants 2/3 of its resources at South American organizations (plc.south.*) because plc.eu itself

obtained only 3/5 of plc.south.* resources and should use only 1/5 of the total locally and pass the remaining 2/5 to plc.asia so that plc.asia and plc.jp can use the same amount as plc.eu does.

Between two top-level authorities:
$C_{T[i],T[i+1]} = \left\{ \left(T[i], \frac{l-i-1}{l} \right) \right\} \cup \bigcup_{k=0}^{i-1} \left\{ \left(T[k], \frac{l-i-1}{l-i} \right) \right\}$ $C_{T[j],T[j-1]} = \left\{ \left(T[j], \frac{j}{l} \right) \right\} \cup \bigcup_{k=j+1}^{l-1} \left\{ \left(T[k], \frac{j}{j+1} \right) \right\},$ for $i = 0, ..., l-2$ and $j = 1, ..., l-1$
Between a top- and a low-level authority:
$C_{T[i],L_{i,j}} = \{(*, 1)\}$ $C_{L_{i,j},T[i]} = \{(L, 1)\},$ for $i = 0, ..., l-1$, where $L_{i,j}$ is a low level authority at $T[i]$'s region
From any authority to its slices:
$C_{A,S} = \{(*, 1)\},$ where S is any slice of authority A

Table 8.1: The formula representing contracts in the linear contract topology, where a single contract is represented with the tuple ($ResourceSource$, $Fraction$).

The formula for any contract in the system is shown in Table 8.1, where $C_{X,Y}$ denotes the the set of contracts granted by X to Y, which are HRNs. The formula is in three categories: (1) contracts between top-level authorities only; (2) contracts between top- and low-level authorities, and (3) contracts between authorities and slices. T is the list of top-level authorities of length l, ordered according to their position in the linear topology, so $T[0]$ =plc.south and $T[4]$ =plc.jp in our case. First, the contracts between two adjacent top-level authorities in the linear topology reflect our fair-sharing heuristic. Each such contract grants a collection of resources from different owners. Second, top- and low-level authorities share their resources without any limit, shown by "1" as the fraction. The top- to low-level contracts again grant resources from many sources; on the other hand, low- to top-level contracts grant a single organization resources. Finally, authorities grant all their resources without limit to their slices. This is done by issuing only one contract, with multiple receivers. Unrestricted sharing within the individual regions increases resource utilization, although it can lead to unfair sharing among lower-level authorities. In a

real deployment, excessive consumer authorities can be prevented over time by configuring lower fractions in their contract rule.

For our experiments we introduce an additional parameter, γ, in order to observe the effect of changing contract rules. We alter any contract ($ResourceSource$, $Fraction$) in the system to ($ResourceSource$, $\gamma \times Fraction$). For different values of γ, observing the system behavior gives insight about the effect of contract rules on resource allocation success rate and time.

8.1.4 User Events

We emulate three types of user events: resource allocation requests, API operations on database objects, and machine log-in requests. We have access to PlanetLab logs that contain information about the parameters and patterns of past user requests. From these logs we can derive traces of those three types of events that would be expected to occur in a federation.

Resource Allocation Requests

The resource allocations happen in the context of slices in PL. Each slice in PL corresponds to a resource discovery/allocation request in our emulation, invoking the querier of CODAL to satisfy resource request. To emulate such events, the primary data we need to derive from PL logs are the RSPEC parameters of allocation requests. These data are not explicitly available in PL logs, because the PL allocation model leaves the resource discovery phase to the user. We have access only to the final allocated resource set, instead of the initial request RSPEC for a slice. Therefore, we choose to derive this request RSPEC from the final resource set allocated to the slice, which consists of virtual machines located at PL machines and their resource properties.

We map PL slices to allocation events with the assumption that several operation calls for a slice are consolidated into one. Normally, the nodes of a PL slice may be allocated with one or more calls to the PLC API as its users select and add them over time in the course of an experiment. On the other hand, we represent all these calls with a single allocation request event, which will allocate all nodes at once, because the primary challenge we see in federated allocation is not handling a relatively larger number of such calls, but allocating large request RSPEC sizes efficiently and from many organizations.

The RSPEC parameter of each emulated event contains as many nodespecs as the number of machines the corresponding slice has currently. The individual node properties in RSPEC are

derived using the per-slice resource consumption logs provided by CoTop measurement tool [21] in PlanetLab. CoTop provides memory, CPU, and bandwidth consumptions every 5 minutes, by each slice in PL and at each node at which it resides. For each slice node, we obtain the averages of such consumptions during 288 intervals, that is, one day, and place node constraints in RSPEC for that nodespec to have such averages as lower bounds. The other fields of RSPEC, geographical location and number of cores, are static values, which are obtained from CoMon node properties data. We designate a default value for disk usage for each slice node since we do not have access to such information directly; a disk is generally a less scarce resource in PL as compared to others and, hence, is not very important in VM's placement to a physical node. We leave all host name fields as variable in the generated RSPEC because we assume the user cares only about the properties of VMs, not the specific physical machines at which they reside. Finally, all RSPECs have the group constraint of all nodespecs reside in distinct physical machines since it is the case in PL's slice allocation model. At the end, we arrive at 460 resource allocation events overall in our emulation.

API Operation and Machine Login Requests

Another set of logs that we obtain from PlanetLab is the API operations performed at the PLC database by users and also administrators. These operations include adding and modifying records of users, nodes, slices, and so on, in the database, configuring and managing resources such as rebooting machines. We implemented a possible policy for such operations using our federation framework and executed traces in our emulation. Specifically, each such operation requires the chain of authentication and authorization certificates for the caller, the latter being specific to the operation to be performed. The authorization chain is similar to the proof of "can register" permission in the security policy in Section 3.3.1, which starts with a certificate signed by the root and ends with the final principal, which is the permission being given.

We derive federation traces from a PLC API log by determining the new origin and destination of each call. For example, a PLC API operation to update a node with the DNS name "node1.planet-lab.titech.ac.jp" will be mapped to a federation call with destination as the organization owning this node, which is plc.jp.titech, and will update the record called plc.jp.titech.node1. Also, by checking which user performed this operation, we determine an origin node close to the geographical location of a user's site in order to run the emulated traces at that location.

The last type of operation in PL that we map to a federation is the machine login requests,

that is, SSH, to the testbed machines. PL handles authorization by having each node download SSH keys of allowed slices every 20 minutes from the PLC database, so that when any user of a slice tries to log in to the machine, the node knows this slice has not expired. In the federation, on the other hand, there is not a central authority from which to obtain keys. In our emulation, each SSH requires two chains of certificates. The first is the caller's authentication certificates, which can be used for accounting at the logged-in node. Second is the proof that the caller is a user that belongs to the slice. Since a slice is an object with an HRN representing a group of users, this proof is similar to authentication, requiring a chain of certificates starting at the root. Similar to API operations, we determine the origin and destination of such calls and run in our experiments.

8.1.5 Organizational Resources

Any machine in PL is donated by one of the institutions that is part of the PL; therefore, per-organization resource sets are composed of such machines as resources. This machine-to-site mapping is obtained from the PLC database. Since some sites contribute a very small number of nodes, as low as 2, the node density can be quite low. To avoid this, we designate some subset of the authorities as data centers. Data center authorities manage all resources of a group of authorities in their region. Our experiments use a total of 50 different data center authorities, 2 at least from each region, overall managing a total of 644 machines. For each node, we extract the node properties, disk, core, CPU, bandwidth, and memory from CoMon node information and create a nodespec to represent the node. The list of nodespecs is encoded inside an authority-specific policy file for each data center authority.

8.1.6 Configuration and Trace Files

All the above configurations and events are encoded in per-organization configuration and trace files, respectively, and next placed in the PL machine that corresponds to that organization in the emulation. The configuration file first consists of basic policy for the organization, such as private keys, as shown in Table 5.1. Next, it contains records for slice and subauthorities if any, containing their public keys and IP addresses. Also encoded is the organizational resource set, federation peers list, and all contracts granted to others, which are obtained from the contract topology. The trace files contain a record for each event, which contains event execution time, caller HRN, operation name and parameters, and its specific destination, if any. Each trace file is

fed into the event subsystem of the CERTDIST daemon at the node for the organization, which invokes them at the exact time for each event.

8.2 Organization Discovery

After passing the system code and configuration and trace files to the node hosting each organization, the CERTDIST daemons in these nodes are started. We have a warm-up period, in which organizations query others to obtain permits and discover remote organizations by periodic `renew_permit` calls. Only after this warm-up period do the events start to be invoked.

Our experiments show that the permit-renewal period can be important for efficient organization discovery. Our strategy in choosing a period length is that the smaller authorities keep their permit-request period high so that they do not overwhelm top-level authorities. On the other hand, the request period between top levels is high, so that path information flows fast from one edge of the federation to another. In our topology, setting the inter-top-level period at 20 seconds, the top-to-low level authority period at 60 seconds, and the low-to-top level authority period at 120 seconds allows more than 350 authorities to learn 50 different permits within 5 minutes without needing excessive CPU or bandwidth.

8.3 Allocation Rate

Our first goal is to measure how successfully a user's resource allocation request can be fulfilled in a federated environment. We define the allocation rate of a single request as the ratio of the number of mapped nodespecs to the total number of nodespecs in an RSPEC. Some parameters that can affect the allocation rate are the size of the request RSPEC and also the contract rules, together with availability in peer organizations.

To measure allocation rate, we make an experiment run using only the resource allocation events. At the end of the experimentation, we gather results for the 460 RSPECs and calculate system average allocation rates. Figure 8.2 shows the average allocation rates with respect to RSPEC sizes, which are grouped by multiples of 40. The second graph shows how many data centers are contacted on average, again with respect the RSPEC size.

We perform four separate experiments with different γ values. As mentioned in Section 8.1.3, this parameter alters the contract rules such that, the higher γ is, the more generously the orga-

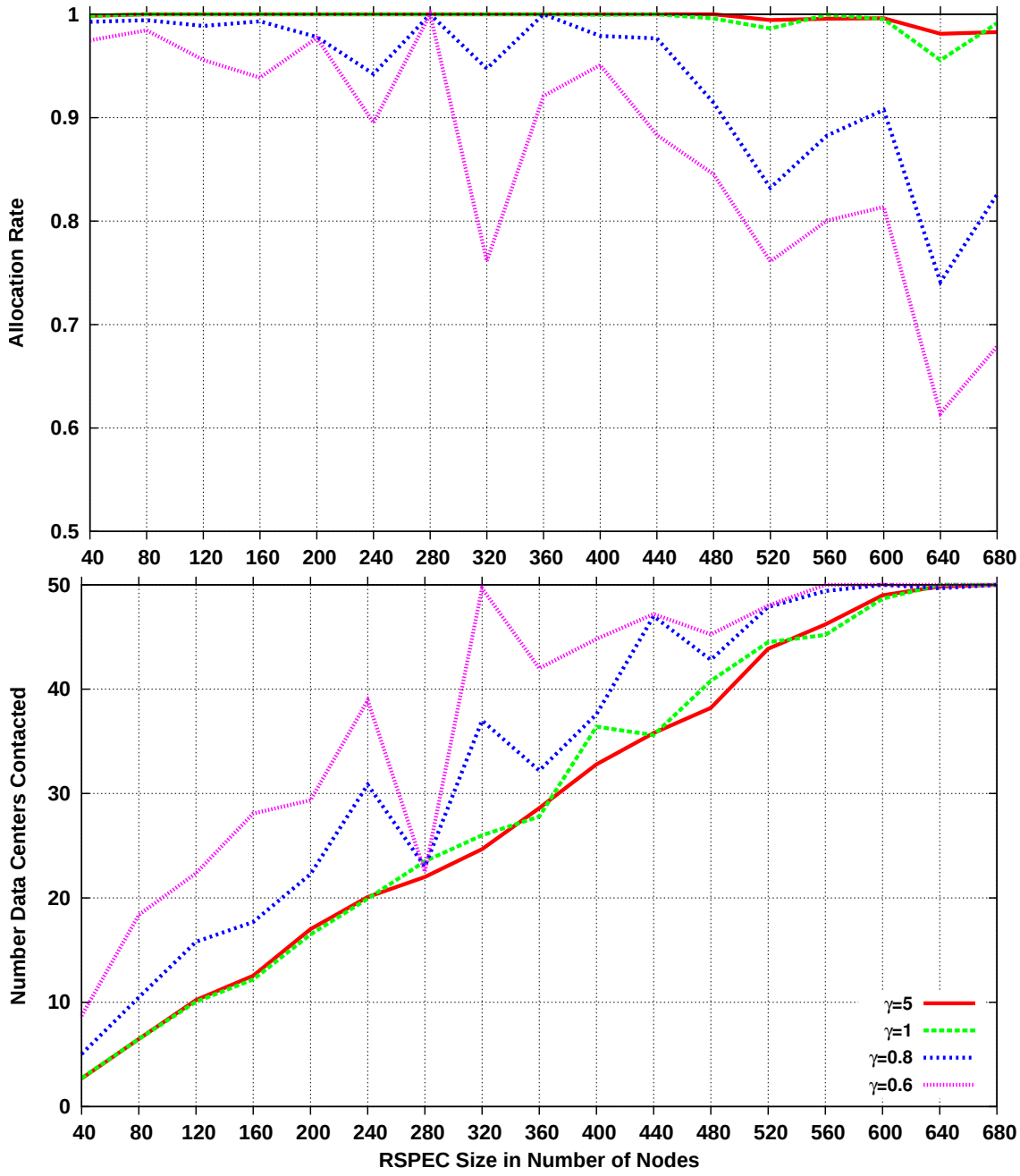


Figure 8.2: Allocation rate and number of data centers queried with respect to RSPEC size in number of nodes

nizational resources are granted. For example, $\gamma = 1$ represents the contract rules that achieve fair sharing according to the geographical region heuristic. On the other hand, $\gamma = 5$ causes all fractions in all contracts to flatten to 1 (fractions greater than 1 default to 1), thereby leading to a first-come, first-served type of allocation policy. The cases where $\gamma < 1$ study what happens when organizations behave less altruistically in sharing resources in contracts granted to others.

The allocation policy for ideal fair sharing, $\gamma = 1$, yields close to 100% allocation rates for smaller slices but slightly lower rates for very large slices, such as ones with more than 600 nodes. This shows us that the current resource sharing in PL is almost fair according to our contract policy. Next, we increase γ to 5 to eliminate any limits in contracts, where we see that the allocation rate increases slightly, especially for large RSPECs.

Not only the size of a requested RSPEC but also the individual nodespec properties in it can affect its allocation rate. For example, there are some slices in PL that reside at machines where they consume 100% of its CPU. In our emulation it can be hard to find a mapping for such a nodespec, because our mediator algorithm avoids such large allocations in order to avoid the singular deprivation problem, which is described in Section 7.3.4. For example, in the upper graph for $\gamma = 5$, the small decrease in allocation rate of RSPECs of sizes 480 to 520 is caused by the "coblitz" slice, which is one of the top 10 memory consumers and top 3 bandwidth consumers in overall slices in PL.

We emphasize that our federated allocation is not performing a single-point global optimization in order to arrive at a perfect fit of all requested RSPECs to all available nodes in the system. It is not possible to run such an optimization because there is no global knowledge of resources and policies since each organization manages its own resources and policies. Nevertheless, even with local optimization, we see that our resource allocation mechanism performs well, with more than a 95% average allocation rate for all RSPEC size categories. Moreover, we believe in real federation deployments, the number and diversity of resources at individual data centers will be much higher, increasing the very large nodespecs' chance of allocation.

We also experiment with smaller values of γ, 0.8 and 0.6, in order to show the result of restrictive policies on allocation rate. With $\gamma = 0.8$, any top-level authority would share only 80% of what it used to share with $\gamma = 1$. Figure 8.2 shows that the allocation rates of RSPECs, especially with size larger than 160, are affected adversely.

Allocation rates and the number of contacted data centers tend to be inversely proportional.

This is because, as long as an RSPEC is not fully allocated, it will cause more data centers to be queried. This can be seen by the fact that a decrease in the allocation graph generally corresponds to a spike in the data centers graph, and vice versa. Also, if the allocation rate of a specific request is less than 1, then the number of contracted data centers will always be the maximum, which is 50. This cannot be seen in the graphs because they depict only the per-interval averages.

8.4 Allocation Time

Another metric that is important for the efficiency of the federation is the resource allocation time. As seen in Figure 8.2, a single RSPEC allocation request can be fulfilled from many datacenters, where requests from each can take time. Therefore, overall allocation time depends on the size and characteristics of an RSPEC and resource amounts in data centers. As a better indicator of the system performance, we study and analyze the allocation times of individual requests to data centers.

The two required stages of allocating from a remote organization are first constructing the proof of a request and next performing the allocation RPC. Our experiments show that both stages can be performed in a few seconds with high probability, but sometimes it can take a long time, as in cases that are explained below. The main factor affecting proof time is availability of cached certificates. On the other hand, the main factors affecting remote allocation request times are RTT and bandwidth to the data center and the mediator algorithm execution time.

Proof construction is a complex process, which retrieves chains of certificates to prove eligibility for allocation. The final proof has to contain the chain of certificates, or, more specifically, permits. During proof construction, IP certificates can also be retrieved for address resolution and ID certificates, for authentication. For example, for plc.south.ufmg to allocate from plc.jp.utokyo, it has to collect certificates for allocation proof if they were not collected recently. It will request the proof certificates from plc.south, which may require resolving its IP by another set of certificate queries. Similarly plc.south will ask for proof certificates from plc, and so on, until it reaches plc.jp.utokyo. The proof is provided to its requester by signing an additional delegation certificate at each step. In the linear topology, as many as nine queries overall will have to be performed to construct proof of allocation from a single organization. Once proof is cached at plc.south, it can be reused to respond proof requests from other lower-level authorities as well; therefore, nine

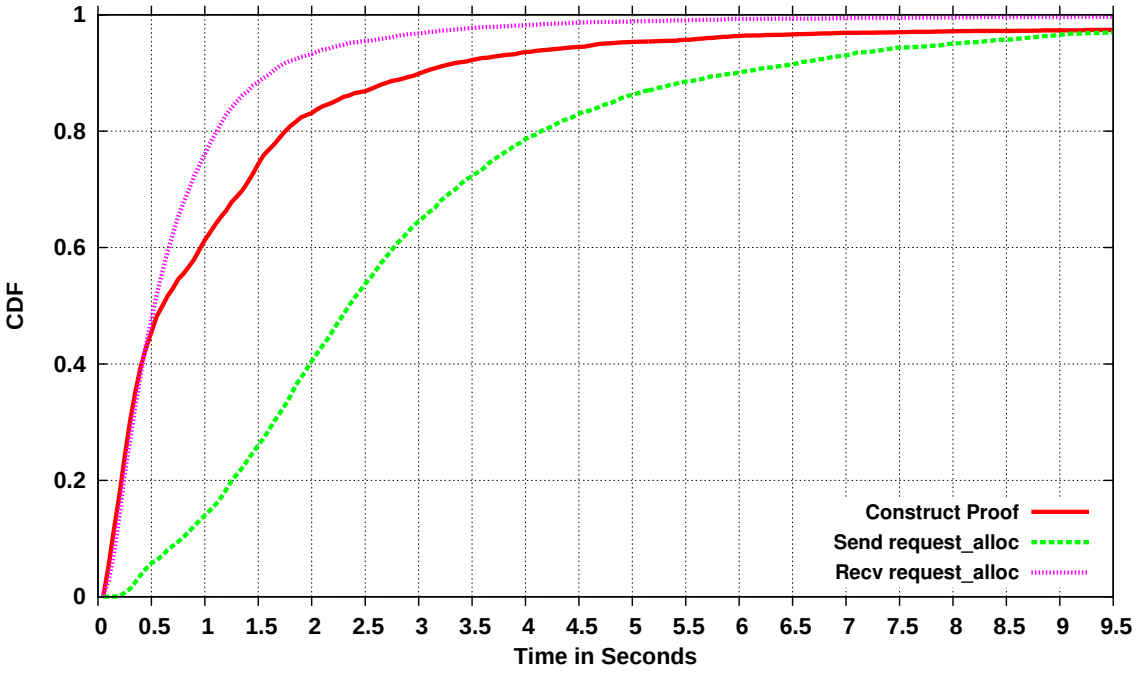


Figure 8.3: CDF of proof construction and received/sent allocation query times

queries may not be required for all proofs.

Figure 8.3 shows the cumulative distribution function (CDF) of proof construction time and the allocation request RPC times. For the latter, we show both the receiver-experienced query processing time (recv request_alloc) and the querier-experienced total time to get the response (send request_alloc).

We see that the proof is constructed fast, in less than a second for most of the requests; however, it can take long time for others. The first case happens when all the certificates of the proof are already cached locally. In that case, the only processing needed is to verify that they adhere to the policy, which can require cryptographic operations such as checking that public keys match between two certificates. Otherwise, remote queries will be made, which take much more time, such as the queries that take more than 0.5 seconds in Figure 8.3. In our experiments, although 97% of proofs end in 10 seconds, we can see proofs taking as much as 40 seconds. This depends on network conditions such as RTT and bandwidth in addition to the number of queries made. A proof consists of many certificates; therefore, tens of KB have to be returned for each query, adding to the delay, especially if bandwidth is not abundant in PlanetLab nodes. We note that local caching of certificates can be utilized much more. For example, it is possible to construct

the proofs of discovered resources before users' allocation requests and store the certificates, so that when a user allocation request happens, the existing proof can be used. In our experiments we construct a proof only when a request is made by the user, but we allow proofs to be reused among different users' requests; therefore, we partly benefit from local caching.

The graph shows the time it takes to process allocation requests with the received `request_alloc`. Its time is mostly caused by the mediator algorithm running time and a small constant time to process input certificates. We observe that algorithm time is proportional to the number of unmapped nodes in an RSPEC and the number of nodes owned by the data center organization. The running time also depends on the CPU availability in the organization server. Although running time can be higher for large RSPECs at slow machines, it is bounded and small for the vast majority of allocation requests.

The requester's perceived time to allocate, shown by sent `request_alloc` time in the graph, is much higher than the time to process the request. This is caused by the time lost in the channel in connection and transferring of query parameters and sending results back. The allocation requests can be performed in geographically distant locations and with low bandwidth, leading to delays. Allocation requests can include more than 100 KB of RSPEC and proof as parameters; and similarly the returned value can include a large result-RSPEC, which takes more time to transfer in low-bandwidth links. The additional time caused by channel delays does not invalidate our emulation but gives insight into what factors can be the cause of delays in a real deployment.

While there is room for improvement in the algorithm, the main source of query time comes from the channel delays; therefore, we can say algorithm time compares reasonably well with the channel delays. The total time taken for system-wide allocation of all 460 slices in experimentation is about 20 minutes.

8.5 Certificate Requests

This section evaluates the certificate requests in the emulated federation in terms of their time and incurred system load. Certificate retrievals happen during construction of the allocation proof, as discussed in Section 8.4, and also for SSH operations after resources are allocated and for the API operations. This section shows experiments that emulate only API and SSH events, which give results general enough for all certificate retrievals. Also, experiments in this section were performed

in earlier stages of the project; therefore, the authentication topology is different. However, this does not affect our analysis of the results. The summary of results of this section is that short certificate lifetimes can cause overhead in organizational servers, but DHT can significantly reduce the load on servers and improve the certificate retrieval times.

8.5.1 Organizational Load

Figure 8.4 shows the number of authoritative certificate requests arriving at the US authority server, plc.us. We can think of plc.us as equivalent to plc authority of our previous experiments, since it represents the North America geographical region. The graph depicts results for two separate runs, each for 1 hour, where the first of these experiments uses only authoritative queries to retrieve certificates. The second has DHT enabled—that is, it uses first DHT query, and if it fails, it performs an authoritative query. We can see that first checking the certificates from DHT significantly reduces possible excessive load in authorities.

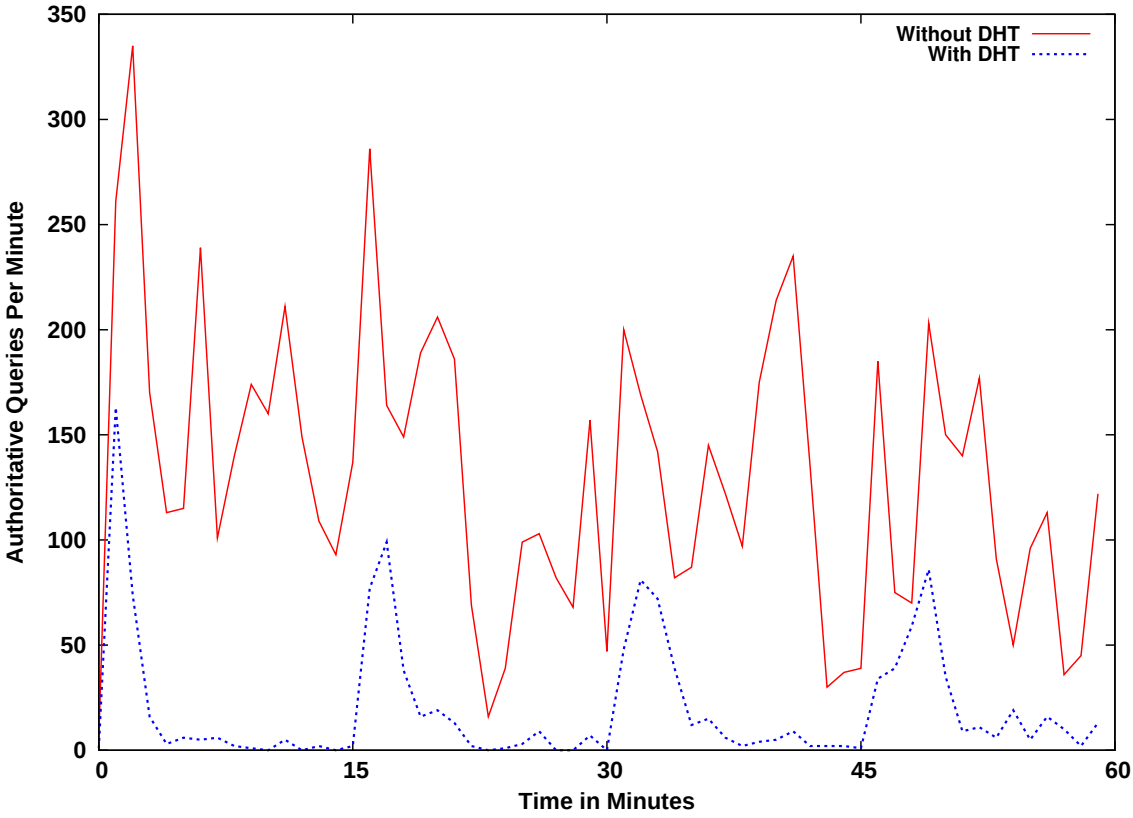


Figure 8.4: Number of certificate requests arriving at the US authority server for DHT-enabled vs disabled runs

There are spikes in the graph that occur periodically, caused by the 15 minutes of certificate lifetime that we designate for particular types of certificates. Whenever a certificate expires, it needs to be renewed by the principals using it, causing requests be made by all those principals at about the same time. For example, an authentication certificate of plc.us.princeton or plc.us will be required by all Princeton users as part of their authentication chain so that they can SSH to any machine in the system; therefore, certificates will be retrieved either from these authorities or their parents. We call this the *flash crowd* effect of certificate retrievals. One way to avoid flash crowds could be to give principals different certificates with more continuous range of expiration times; however, this causes more certificates to be signed at the server, which has a high computational cost. So far in our experiments, DHT caching was enough to distribute the load and avoid a high number of requests at authoritative servers.

Short lifetimes can be given to some certificates in order to achieve fail safety in the system. For example, certificates that bind public keys to HRNs have short lifetimes, first because users' private keys can be stolen, in which case a short TTL keeps the window of vulnerability short. Second, principals can misbehave, in which case the voucher of the principal should have a fast way of indicating that it no longer trusts that principal. In the experiments of this section, all certificates have 15 minutes of lifetime, that is, authentication, API authorization, and slice certificates. On the other hand, resource delegation certificates of evaluation in the previous section had long lifetimes since we assume resources are granted for longer time periods and do not need fast revocation.

Our results does not include certificate retrievals that occur during resource allocation. Since SSH and API events are much more than allocation requests, our results are still valid in the general case. Because there are about one million SSH events and ten thousand API operations per day, compared to the 460 slice-creation events that happen over many days. Therefore, we can say that the users generally allocate once and then use the allocated resources many times, during which certificates are retrieved most often.

8.5.2 Certificate Retrieval Times

We measure the effect of DHT on certificate retrieval times and find that it helps reduce time under high loads. We perform experiments that run normal and high loads and observe their certificate retrieval times for both DHT enabled and disabled experiments.

The certificate retrieval process in a DHT enabled emulation is such that first, DHT is queried for the certificate. Since it is a key-based search, it does not require IP resolution. If no result returns for some time threshold, the DHT query times out and an authoritative query is made. In this case, if the IP address of the destination of an authoritative query is not known, it will have to be retrieved first, with another set of certificate queries. For the DHT disabled configuration, the process starts directly by authoritative query. We measure the overall time this process takes for each certificate and study results in both DHT enabled and disabled configurations. We select the DHT timeout threshold to be 0.7 seconds since we observe that 90% percent of successful DHT queries are completed within this time.

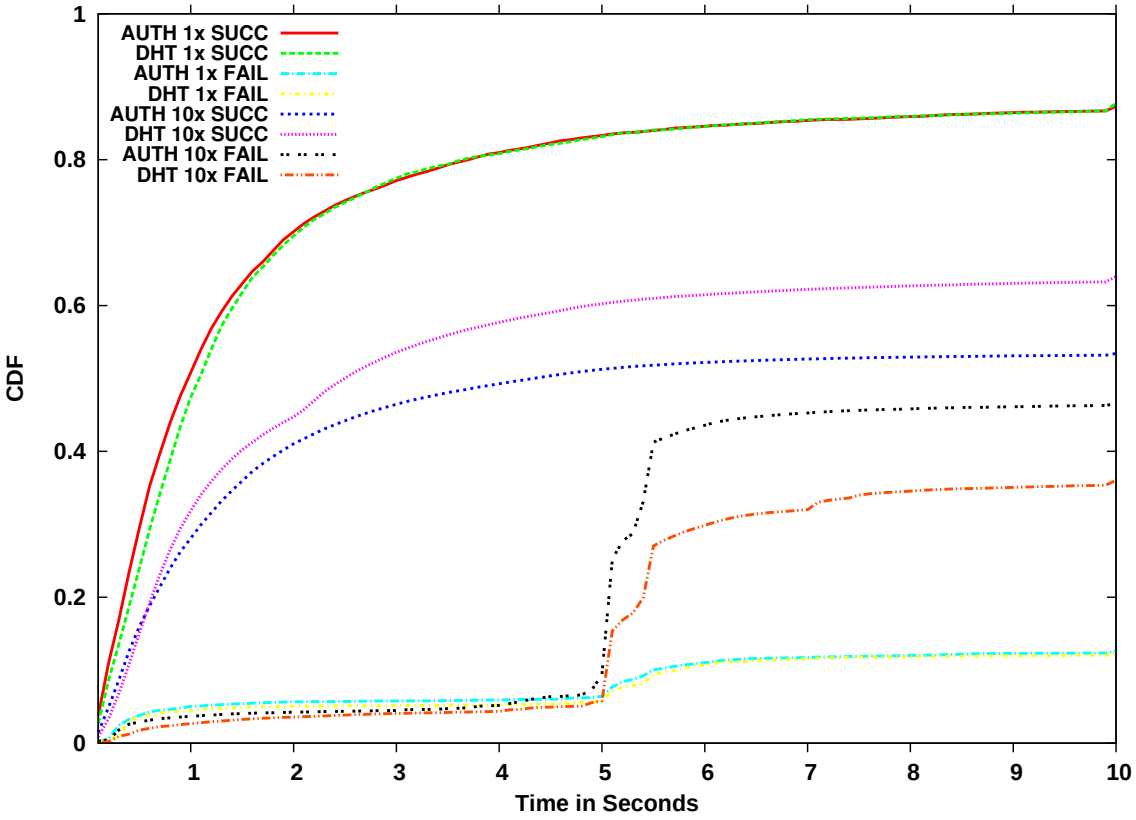


Figure 8.5: CDF of certificate retrieval times for normal and high loads and DHT enabled and disabled configurations

Figure 8.5 shows the CDF of certificate retrieval times for normal load and 10x load, with DHT and without DHT. The normal-load experiment emulates the SSH and API events at the same rate they appear in logs, whereas the 10x load emulates them with 10 times the normal user base.

The upper two lines in the graph show that, with normal load, DHT does not improve retrieval times, although it was shown before to reduce load on the servers. This is because some time is lost in failed DHT queries, and authoritative queries execute fast enough since they are not very overwhelmed. On the other hand, the lower two lines show that, as the system load increases, DHT improves the retrieval times and also the success rate. Because, the authority servers become very overwhelmed and it takes more time to process the authoritative queries, DHT can help by spreading the load, thereby improving retrieval times.

The graph shows that there is about 15% failure rate in certificate retrieval under normal load. Also, lower lines show that failures happen around 5 seconds, which is our authoritative query timeout value. The reason of the high rate of failures is mainly the experimentation node set. This set of experiments used around 550 machines, some of which are not in very good condition and can go down or lack machine and link resources during the experimentation period. In experiments of Section 8.3 and 8.4, on the other hand, we used fewer nodes and select those in the best condition; therefore, query failures in that case were around 1%. Failed retrievals do not invalidate our analysis on the effect of DHT on retrieval times. We expect such failures to be eliminated in real deployments by per-organization management teams.

In summary, the two graphs show that DHT can be useful for reducing load on organizational servers, improve certificate retrieval times, and increase their success rate. The rate of the operation requests in the system can affect the overall benefits obtained from using the DHT. In addition, we note that the existence of common certificates is key to efficiency benefits of DHT. In other words, if the same certificates are needed by many principals, it is possible to replicate and cache them in DHT to improve efficiency.

Chapter 9

Conclusions

In this thesis we investigated the access control and resource allocation problems in computational federations, such as federated testbeds and cloud computing federations, and introduced our federation framework that makes it easy to build federations of varying complexities.

Our work aims to be the foundation of future federations. The key challenge is the growing number of participant organizations in a federation, establishing and maintaining trust relationships in an environment where each participant can potentially allocate resources from any other. Today's federations have small number of participant organizations and their architectures do not address the scalability problem.

Unlike in a traditional market where buyer and seller interaction lasts until payment occurs for a particular price, computational resource exchange is an ongoing relationship where resources are leased for some period, during which both the tenant (user) and resource owner need to trust each other, and allocations should adhere to a set of conditions imposed by the owner. As a result, computational resource exchange depends not only on price and payment, but also on trust and policy. Resource owners accept only trusted tenants and vice versa, where trust arises from in-person agreements, usage history, etc., and this trust may have to be strengthened, repaired or rebuilt from time to time through in-person checks and sanctions. We assume maintaining such trust is not trivial and can be time consuming since it involves humans.

In Chapter 2 we introduced a trust model analysis of today's federations and technologies and showed that they have simple trust structures. These structures such as circle of trust or bridge model may not be scalable because they require too many trust relationships to be established

by all or some participants of the federation. Our federation framework takes the approach of allowing any type of trust structures in a federation, supporting simple models such as the circle of trust, as well as complex ones such as the distributed trust model in federations.

The primary contribution of this thesis is to show that arbitrarily complex indirect trust relationships can be a practical way to scale federations. This can be realized by a system that synthesizes trust management, policy languages and resource discovery systems. These three areas were studied separately in the past, but we show that they are related, and constitute different layers of a more general system that we call the federation framework.

From Chapters 3 to 8, we have explained the design, implementation and evaluation of the federation framework, which is the only system to our knowledge that combines trust management, policy languages and resource discovery in a single system. Individual pieces of this framework, CERTDIST, FPL and CODAL, have unique features and provide contributions in the areas of trust management, policy languages and resource discovery systems, respectively. Chapter 2 showed that none of the previous systems in these three areas have features capable of building our federation framework.

Expressiveness, ease of use and extensibility are primary goals of the federation framework. Chapter 3 explained the key actors and abstractions that allow us to achieve these goals. Security architects use the lowest layer, CERTDIST, to create expressive FPL security and allocation policies, which in turn are used by system admins to write contracts easily. Users interact with CODAL, which underneath uses FPL contracts and policies to discover and authorize for resources.

Chapter 4 explained CERTDIST in detail. CERTDIST can express programmable credentials, and perform any of the compliance checking methods seen in previous work, thanks to its Prolog engine. Also, CERTDIST's DHT certificate retrieval allows distributed proofs that were not possible before. As a result, CERTDIST serves as the underlying mechanism for FPL to be able to express both security and allocation policies.

In Chapter 5 we argued that a trust management system is not very usable without a domain specific language, and vice versa. We introduced FPL as the policy language for federations, layered on top of CERTDIST. FPL exposes the expressiveness of CERTDIST with easy-to-use primitives, such as contracts, which are crucial for federations to scale easily.

Similarly, CODAL is built on top of FPL, because resource discovery must be policy aware, as explained in Chapter 6. CODAL uses FPL contracts to discover organizations, uses the se-

curity policies to authorize operations on resources, and finally the allocation policies to acquire sets of resources. CODAL applies a partial matching algorithm to satisfy a user request from multiple domains, and addresses the problem of resource information being private to individual organizations.

One future work for federation framework is to enrich the FPL language with constructs relevant for cloud computing federations, such as pricing. Pricing can fit into our federation model by expressing prices as part of contracts, allowing both the resource owners and the brokers to charge for allocations. Specifically, this could be done by using an additional dimension in contract constraints, that denotes the allocation cost. Granter of a contract can express various formulas that compute the allocation cost, similar to the way they express the allocation constraints. When a user requests an RSPEC, contracts can determine not only the type, quantity and location of resources, but also the cost. Depending on this cost, allocation can be allowed to go through or not. CODAL should also take into consideration the pricing constructs, where the CODAL's mediator can compute the overall cost of a requested allocation by adding the resource and brokering costs, and CODAL's querier can be improved to minimize the cost of allocations, in a similar way that it processes the group constraints. The actual money transfer can occur during the allocation or at a later time. Misbehaviors in transactions should be unlikely, since parties are accountable to each other through contracts, and transactions can be proven by signed credentials, and allocations are recorded by CODAL's accounting structures.

Another future work is to optimize CODAL's querier by utilizing parallel queries. CODAL performs queries at many peers sequentially in order to discover resources, which can take time. Although time consuming, more queries can be useful to find cheaper or better resources in terms of various metrics such as latency and bandwidth. To address this trade-off, new strategies involving parallel queries can be developed to optimize allocation time, cost or other metrics. Reducing the querier runtime can be important for possibly supporting transactions, where allocations are reserved in various peers for a short time, until all of RSPEC is satisfied by unit and group constraints, and finally are executed at the same time. That said, transaction support could be another important future direction depending on the need for commercial use cases.

Once our framework is adopted, one future direction will be to analyze commonly issued contracts between organizations and their relationship to the types of contracts used in other research areas including economics. Federation framework should make it very easy to create

contracts by providing GUI tools, providing commonly used contract types and letting admins to possibly just fill out the details. Analysis of these agreements will give insights into what contracts allow efficient resource utilization and also are healthy for federation to grow.

Similarly, the analysis of new policy requirements and extending FPL language with new primitives is a future direction. As mentioned before, pricing is one such construct. As new contract types will be needed, new FPL primitives will be introduced to realize them, which can contribute to the policy languages research in general. Similar to the tools for admins to easily create contracts, our framework should also provide tools that help security architects construct correct and efficient FPL primitives. Such tools and checkers are additional future work that will allow easy evolution of our framework with new policies.

In our view, there will be two drivers behind the adoption of our federation framework: the need for fine-grained allocation policies and indirect trust structures. Currently in GENI, resource requesters and owners trust a single GENI clearinghouse that forms a bridge trust model. Since everybody trusts the clearinghouse, there is not much incentive for owners to express complex contracts; rather, they express simple policies, such as, "allocate 4 machines to clearinghouse". This can make sense in an academic environment, where trust breaches will be minimal and clearinghouse vetting for all participants can be enough, as long as the number of participants is not very high. On the other hand, not all cloud computing institutions may be willing to operate under a single intermediary. The number of these institutions can also be a reason for adopting indirect trust, since it is hard to establish and maintain trust relationships. Also, profit-based institutions have an incentive to maximally utilize the available hardware, which can result in more fine grained allocation conditions, involving varying payments, locations and durations. Overall, we believe our federation framework will be a solid foundation and a reference point while building new federations, as well as expanding and merging existing ones.

Bibliography

[1] D. Abramson, R. Buyya, and J. Giddy. A computational economy for grid computing and its implementation in the nimrod-G resource broker. *Future Generation Comp. Syst*, 18(8):1061–1074, 2002.

[2] J. Albrecht, D. Patterson, and A. Vahdat. Scalable wide-area resource discovery, July 27 2004.

[3] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, Á. Frohner, A. Gianoli, K. Lörentey, and F. Spataro. VOMS, an authorization system for virtual organizations. In F. F. Rivera, M. Bubak, A. G. Tato, and R. Doallo, editors, *European Across Grids Conference*, volume 2970 of *Lecture Notes in Computer Science*, pages 33–40. Springer, 2003.

[4] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[5] N. Andrade, W. Cirne, F. V. Brasileiro, and P. Roisenberg. Ourgrid: An approach to easily assemble grids with equitable resource sharing. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *JSSPP*, volume 2862 of *Lecture Notes in Computer Science*, pages 61–86. Springer, 2003.

[6] Attribute Based Access Control(ABAC). `http://groups.geni.net/geni/wiki/TIEDABACModel`.

[7] A. Barmouta and R. Buyya. Gridbank: A grid accounting services architecture (GASA) for distributed systems sharing and integration. *CoRR*, cs.DC/0210002, 2002.

[8] J. Basney, W. Nejdl, D. Olmedilla, V. Welch, and M. Winslett. Negotiating trust on the grid. In *Semantic Grid: The Convergence of Technologies*, number 05271 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005.

[9] A. C. Bavier, N. Feamster, M. Huang, L. L. Peterson, and J. Rexford. In VINI veritas: realistic and controlled network experimentation. In L. Rizzo, T. E. Anderson, and N. McKeown, editors, *SIGCOMM*, pages 3–14. ACM, 2006.

[10] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar. GENI: a federated testbed for innovative network experiments. *Computer Networks (Amsterdam, Netherlands: 1999)*, 61:5–23, Mar. 2014.

[11] S. Bhatia, A. C. Bavier, L. L. Peterson, and S. Sevinc. sfatables: A firewall-like policy engine for federated systems. In *ICDCS*, pages 467–476. IEEE Computer Society, 2011.

[12] P. Bhoj, S. Singhal, and S. Chutani. SLA management in federated environments. *Computer Networks*, 35(1):5–24, 2001.

[13] I. Bird. Computing for the large hadron collider. 61:99–118, 2011.

[14] Blaze, Feigenbaum, and Keromytis. Keynote: Trust management for public-key infrastructures. In *IWSP: International Workshop on Security Protocols, LNCS*, 1998.

[15] Blaze, Feigenbaum, and Strauss. Compliance checking in the policymaker trust management system. In *FC: International Conference on Financial Cryptography*. LNCS, Springer-Verlag, 1998.

[16] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1996. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[17] M. Brinn, N. Bastin, A. Bavier, M. Berman, J. Chase, and R. Ricci. Trust as the foundation of resource exchange in GENI. In *Proceedings of the 10th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, June 2015.

[18] J. Bustos-Jimenez, C. Varas, and J. Piquer. Sub-contracts: delegating contracts for resource discovery. In *Grid Middleware and Services*, pages 95–104. Springer, 2008.

[19] R. Buyya and S. Vazhkudai. Compute power market: Towards a market-oriented grid. In *CCGRID*, pages 574–581. IEEE Computer Society, 2001.

[20] D. W. Chadwick and A. Otenko. The PERMIS X.509 role based privilege management infrastructure. *Future Generation Computer Systems*, 19(2):277–289, 2003.

[21] CoTop: A Slice-Based Top for PlanetLab. `http://codeen.cs.princeton.edu/cotop/`.

[22] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science*, 1459, 1998.

[23] Earth System Grid. `https://www.earthsystemgrid.org`.

[24] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computers and Security*, pages 83–91. ACM Press, 1998.

[25] J. Frey, T. Tannenbaum, M. Livny, I. T. Foster, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[26] Y. Fu, J. S. Chase, B. N. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (19th SOSP'03)*, pages 133–148, Bolton Landing, NY, USA, Oct. 2003. ACM SIGOPS.

[27] F. Gagliardi, B. Jones, M. Reale, and S. Burke. European DataGrid project: Experiences of deploying a large scale testbed for E-science applications. *Lecture Notes in Computer Science*, 2459, 2002.

[28] B. N. Grosof, Y. Labrou, and H. Y. Chan. A declarative approach to business rules in contracts: courteous logic programs in XML. In *EC*, pages 68–77, 1999.

[29] B. N. Grosof and T. C. Poon. Sweetdeal: representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In G. Hencsey, B. White, Y.-F. R. Chen, L. Kovács, and S. Lawrence, editors, *WWW*, pages 340–349. ACM, 2003.

[30] Interoperable Global Trust Federations. `https://igtf.org`.

[31] InCommon Federated Identity and Access Management. `http://www.incommonfederation.org/`.

[32] T. Jim. SD3: a trust management system with certified evaluation. In *Symposium on Security and Privacy (SSP '01)*, pages 106–115, Washington - Brussels - Tokyo, May 2001. IEEE.

[33] H. Jin, X. Shi, W. Qiang, and D. Zou. DRIC: Dependable grid computing framework. *IEICE Transactions*, 89-D(2):612–623, 2006.

[34] T. Kurze, M. Klems, D. Bermbach, A. Lenk, S. Tai, and M. Kunze. Cloud federation. In *Proceedings of the 2nd International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING 2011)*. IARIA, September 2011. Best Paper Award.

[35] B. Lang, I. T. Foster, F. Siebenlist, R. Ananthakrishnan, and T. Freeman. A flexible attribute based access control method for grid computing. *J. Grid Comput*, 7(2):169–180, 2009.

[36] Li, Grosof, and Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACMTISS: ACM Transactions on Information and System Security*, 6, 2003.

[37] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *2002 IEEE Symposium on Security and Privacy (SSP '02)*, pages 114–130, Washington - Brussels - Tokyo, May 2002. IEEE.

[38] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, 2003.

[39] J. Linn. Trust models and management in public-key infrastructures. Technical report, RSA Data Security, Inc., pub-RSA:adr, Nov. 2000.

[40] M. Lorch and D. G. Kafura. The PRIMA grid authorization system. *J. Grid Comput*, 2(3):279–298, 2004.

[41] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *IPTPS*, pages 53–65, 2002.

[42] W. Nejdl, D. Olmedilla, and M. Winslett. Peertrust: Automated trust negotiation for peers on the semantic web, 2004.

[43] M. Noureddine and R. Bashroush. An authentication model towards cloud federation in the enterprise. *Journal of Systems and Software*, 86(9):2269–2275, 2013.

[44] J. Novotny, S. Tuecke, and V. Welch. An online credential repository for the grid: Myproxy. In *HPDC*, page 104. IEEE Computer Society, 2001.

[45] Oasis Trust Models Guidelines. `http://www.oasis-open.org`.

[46] OnApp: A complete IaaS platform for service providers. http://onapp.com/.

[47] OpenID. http://openid.net/.

[48] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier. A federated multi-cloud paaS infrastructure. In *Proc. 2012 IEEE Fifth International Conference on Cloud Computing (5th IEEE CLOUD'12)*, pages 392–399, Honolulu, HI, USA, June 2012. IEEE Computer Society.

[49] K. Park and V. S. Pai. Comon: a mostly-scalable monitoring system for planetlab. *Operating Systems Review*, 40(1):65–74, 2006.

[50] L. Pearlman, V. Welch, I. T. Foster, C. Kesselman, and S. Tuecke. A community authorization service for group collaboration. *CoRR*, cs.DC/0306053, 2003.

[51] L. Peterson, S. Sevinc, J. Lepreau, R. Ricci, J. Wroclawski, T. Faber, S. Schwab, and S. Baker. Slice-Based Facility Architecture. In *Ad Hoc Design Document*, August 2008.

[52] L. L. Peterson and T. Roscoe. The design principles of planetlab. *Operating Systems Review*, 40(1):11–16, 2006.

[53] ProtoGENI: Prototype implementation and deployment of GENI. http://www.protogeni.net/.

[54] D. M. Reeves, M. P. Wellman, and B. N. Grosof. Automated negotiation from declarative contract descriptions. *Computational Intelligence*, 18(4):482–500, 2002.

[55] Research Federations. https://refeds.org.

[56] A. Sahai, A. Durante, and V. Machiraju. Towards automated SLA management for web services. Technical Report HPL-2001-310R1, Hewlett Packard Laboratories, Apr. 16 2001.

[57] Security Assertion Markup Language (SAML) v2.0. http://www.oasis-open.org/specs/#samlv2.0.

[58] S. Sevinc. A path to evolve to federation of testbeds. In T. Korakis, H. Li, P. Tran-Gia, and H.-S. Park, editors, *TRIDENTCOM*, volume 90 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 126–141. Springer, 2011.

[59] S. Sevinc, L. L. Peterson, T. Jim, and M. F. Fernández. An emulation of GENI access control. In T. V. Benzel, J. Mirkovic, and A. Stavrou, editors, *CSET*. USENIX Association, 2009.

[60] Shibboleth federated identity management system. `http://shibboleth.internet2.edu/`.

[61] M. Thompson, A. Essiari, K. Keahey, V. Welch, S. Lang, and B. Liu. Fine-grained authorization for job and resource management using akenti and the globus toolkit, June 20 2003. Comment: CHEP03, La Jolla, Mar 24-27, TUB2006, Grid Security, 7 pages, 5 figures.

[62] D. C. Verma, S. Sahu, S. B. Calo, M. Beigi, and I. Chang. A policy service for GRID computing. In M. Parashar, editor, *Grid Computing – (3rd GRID'02)*, volume 2536 of *Lecture Notes in Computer Science (LNCS)*, pages 243–255. Springer-Verlag (New York), Baltimore, MD, USA, Nov. 2002.

[63] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation, pages 255-270*. Boston, MA, Dec. 2002, 2003.

[64] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[65] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. Analyzing market-based resource allocation strategies for the computational grid. *IJHPCA*, 15(3):258–281, 2001.

[66] J. Wu, C. Leangsuksun, V. Rampure, and H. Ong. Policy-based access control framework for grid computing. In *Proc. Sixth IEEE International Symposium on Cluster Computing and the Grid (6th CCGRID'06)*, pages 391–394, Singapore, May 2006. IEEE Computer Society (Los Alamitos, CA).

[67] X. Yang, B. I. Nasser, M. Surridge, and S. E. Middleton. A business-oriented cloud federation model for real-time applications. *Future Generation Computer Systems (FGCS)*, 28(8):1158–1167, Oct. 2012.

[68] Q. Yin, A. Schüpbach, J. Cappos, A. Baumann, and T. Roscoe. Rhizoma: A runtime for self-deploying, self-managing overlays. In J. Bacon and B. F. Cooper, editors, *Middleware*, volume 5896 of *Lecture Notes in Computer Science*, pages 184–204. Springer, 2009.

[69] X. Zhang, Q. Li, J.-P. Seifert, and M. Xu. Flexible authorization with decentralized access control model for grid computing. In *HASE*, pages 156–165. IEEE Computer Society, 2007.