Federated Role-based Access Control in Exertion-oriented Programming

Satish Vellanki and Michael Sobolewski
Computer Science, Texas Tech University
SORCER Research Group
satish.vellanki@ttu.edu, sobol@cs.ttu.edu

Abstract

Federated computing environments expose lots of resources in order to serve their clients, which include system services, domain-specific services, and distributed file systems. A flexible and coordinated mechanism to control access to these resources is proposed which allows participants to form themselves into collaborative groups and secure access is granted to group members. Then, the participants can make resources available to a named group and manage locally the members in the group with required permissions across multiple domains. We explain how the proposed approach focused on user's local namespace is used in exertion-oriented programming and in particular in a SORCER federated file system where members of a group or delegated services can securely fetch any file replica that is available to a named group from any byte store service.

1. Introduction

The SORCER environment provides a way of creating service-oriented programs and executes them in a metacomputing environment. The service-oriented paradigm is a distributed computing concept wherein objects across the network play their predefined roles as service providers. Service requestors can access these providers by passing messages called service exertions. An exertion defines how the service providers federate among themselves to provide the requestor with required service collaboration. These services form an instruction-set of virtual metacomputer. The federated environment requires an access control mechanism to protect the resources of the metacomputer from unauthorized activities. This calls for a scalable authorization mechanism that scales along with the grid of resources while allowing the users to collaborate with each other.

In group-based security in a federated file system , possible ways of constructing a group manager service are discussed with federated environments in view. In this paper we try to investigate ways to improve upon the concept by avoiding a global Certificate Authority (CA) while at the same time enabling users to share resources with people from any administration domain without a global authority. The rest of the paper is divided into the following sections: Section 2 describes background and literature review, Section 3 gives introduction to SORCER, Section 4 describes service messaging with exertions, Section 5 talks about authentication and authorization with exertions, Section 6 looks into a role-based framework for SORCER and Section 7 presents a deployment of the framework in a federated file system.

2. Background and literature review

Access control comprises of authentication, authorization, and auditing. Authentication is the process of verifying the identity of a user, service or device. Authorization is the process of determining the access level of the authenticated identity on any requested resource. For example, allowing an authenticated user to read a file. Auditing allows us to review all authentication and authorization requests to determine system accountability and any gaps in security. For example, analysis of users logged sessions on a computer and updated resources. In this paper we concentrate on how to develop reliable authentication and authorization in federated environments across multiple administration domains.

2.1 Access control techniques

Access controls systems follow one of the following three approaches:
1. A mandatory access control system where any user who created an entity will not have all the rights on the entity. He/she may share it with other users but they cannot assume full control on this entity. The security policy on the entity is determined by the properties attached to it. This kind of access control is usually required in defense environments where any resource is dealt with utmost security.
2. In a discretionary access control system , the owner of a resource specifies who will be allowed the access to it and what kind of access is allowed. This is the most popular technique. Many existing file systems follow this access control technique. In a federated environment many users can collaborate to get a particular work done. Setting permissions for each user and following them through time is practically not

possible. Rather users are grouped according to some criteria and the access is managed per group. Most existing systems do not allow a new group to be created by normal users. The administrator has to be involved in creating a new group, adding/removing users to/from it and in managing it. File systems in UNIX and Windows operating systems employ this approach.

3. In a role-based access control system the permissions are defined for each group by a security authority on each resource. These roles usually do no change through the life of the system. The collection of roles is predefined. Each role is associated with a set of permissions. Any access to resource is granted if the requesting user belongs to any of the roles that allow access to this resource. Role-based access control is easier to manage and permissions can be granted and revoked any time.

Federated environments require the ability of a discretionary access control system while keeping the ease of use of a role-based access control system. Rather than predefining the roles, a user should be allowed to create roles according to his/her wish. The user can then set the permissions a role defines on entities he/she owns. In this case these roles become local to the current user. A role or the permission of a role on any entity created by one user cannot be modified by another user.

Such a complex access control system that works with multiple domains and users usually makes the use of public key cryptography. Public key cryptography or asymmetric cryptography makes use a pair of cryptographic keys – public and private keys, in such a way that given the public key, the private key cannot be usually determined. Any content encrypted using one of these keys can only be decrypted using the other key. This adds lot of security to previously insecure communications and allows for unique authentication of the owner of the private key. The public key can be published in a common directory while the private key must be stored in a secure place. Any content sent to the owner of the key pair is encrypted with the owner's public key so that only the owner can decrypt it with the private key and read it. The owner can use the private key to sign messages that can be verified by any other person or system using the matching public key. This gives the possibility of having digital certificates that can be verified. While public key cryptography provides so many uses the greatest problem with it is determining the public key of an entity with who you wish to have a secure communication. The Public Key Infrastructure (PKI) has solved this problem, which is an arrangement to bind a public key to a user identity by a Certificate issuing Authority (CA). The user identity or the distinguished name should be globally unique in PKI. The CA verifies that the identity really belongs to the user in question before issuing a certificate. PKI enables the secure communication between two parties that have no prior knowledge of each other. Hence as per definition PKI can provide authentication of the key pair owner while it cannot represent any form authorization. Another public-key certificate standard – Simple Public Key Infrastructure/Simple Distributed Security Infrastructure (SPKI/SDSI) makes it possible to represent authorization grants using digital certificates without a need for global CA.

2.2 Simple Public Key Infrastructure

SPKI is a merger of two separate designs – SPKI and SDSI . SDSI allows defining groups and group membership certificates. SPKI concentrates on providing authorization certificates. Thus SPKI standard defines two certificate formats – name certificates and authorization certificates. The name certificates bind a public key to a name in the local namespace of the issuing authority. Any user who possesses a key-pair can issue name certificates, which makes the user a certificate authority as in PKI. This is not possible in PKI where only few defined authorities can issue these certificates . The authorization certificate defines an authorization grant by the issuer of the certificate. It is possible to allow delegation of authorization grant in these certificates.

SPKI by reducing the dependence on a central certificate authority allows the system to scale to any number of users from multiple domains. In fact SPKI designers believed that a central certificate authority serves no real purpose . A user can share his resources with any other user in the system provided he knows the public key of that user. He can add any user to his list of local users by importing their public keys. He can also give a friendly local name that he intends to use for this user. Authorization grants can be made using the local names or the recommended way of directly using public key in the certificates. SPKI/SDSI is defined in RFC specifications 2692 and 2693.

SPKI allows the authorization grant to be delegated by the grantee to others. The granter can decide whether to delegate or not when issuing the certificate. SPKI also defines *threshold subjects* where the authorization is granted when a minimum of k out of n granters concur to allow access to a resource.

3. SORCER

SORCER (Service Oriented Computing EnviRonment) is a federated service-to-service (S2S) metacomputing environment that treats service providers as network objects with well-defined semantics of a federated service object-oriented architecture. It is based on Jini semantics of services in the network and Jini

programming model with explicit leases, distributed events, transactions, and discovery/join protocols. While Jini focuses on service management in a networked environment, SORCER focuses on exertion-oriented programming and the execution environment for exertions . SORCER uses Jini discovery/join protocols to implement its *exertion-oriented architecture* (EOA) using *federated method invocation* , but hides all the low-level programming details of the Jini programming model.

In EOA, a service provider is an object that accepts remote messages from service requestors to execute collaboration. These messages are called service exertions and describe *service (collaboration) data, operations* and collaboration's *control strategy.* An *exertion task* (or simply a *task*) is an elementary service request, a kind of elementary federated instruction executed by a single service provider or a small-scale federation for the same service data. A composite exertion called an *exertion job* (or simply a *job*) is defined hierarchically in terms of tasks and other jobs, a kind of federated procedure executed by a large-scale federation. The executing exertion is dynamically bound to all required and currently available service providers on the network. This collection of providers identified in runtime is called an *exertion federation*. The federation provides the implementation for the collaboration as specified by its exertion. When the federation is formed, each exertion's operation has its corresponding method (code) available on the network. Thus, the network *exerts* the collaboration with the help of the dynamically formed service federation. In other words, we send the request onto the network implicitly, not to a particular service provider explicitly.

The overlay network of service providers is called the *service grid* and an exertion federation is in fact a *virtual metacomputer*. The metainstruction set of the metacomputer consists of all operations offered by all service providers in the grid. Thus, an exertion-oriented (EO) program is composed of *metainstructions* with its own *control strategy* and a *service context* representing the metaprogram data. The service context describes the collaboration data that tasks and jobs work on. Each provider guards the resources specified in service context with the help of two providers `Authenticator` and `Authorizer` described in Section 6. Each service provider offers services to other service peers on the object-oriented overlay network. These services are exposed *indirectly* by operations in well-known public remote interfaces and considered to be elementary (tasks) or compound (jobs) activities in EOA. Indirectly means here, that you cannot invoke any operation defined in provider's interface directly. These operations can be specified in the requestor's exertion only, and the exertion is passed by itself on to the relevant service

provider via the top-level `Servicer` interface implemented by all service providers called *servicers*— service peers. Thus all service providers in EOA implement the `service(Exertion, Transaction):Exertion` operation of the `Servicer` interface. When the servicer accepts its received exertion, then the exertion's operations can be invoked by the servicer itself, if the requestor is authorized to do so. Servicers do not have mutual associations prior to the execution of an exertion; they come together dynamically (federate) for a collaboration as defined by its exertion. In EOA requestors do not have to lookup for any network provider at all, they can submit an exertion, onto the network by calling `Exertion.exert(Transaction):Exertion` on the exertion. The `exert` operation will create a required federation that will run the collaboration as specified in the EO program and return the resulting exertion back to the exerting requestor. Since an exertion encapsulates everything needed (data, operations, and control strategy) for the collaboration, all results of the execution can be found in the returned exertion's service contexts.

Domain specific servicers within the federation, or task peers (*taskers*), execute task exertions. *Rendezvous* peers (jobbers and spacers) coordinate execution of job exertions. Providers of the `Taker`, `Jobber,` and `Spacer` type are three of SORCER main infrastructure servicers-see Figure 1. In view of the P2P architecture defined by the `Servicer` interface, a job can be sent to any servicer. A peer that is not a `Jobber` type is responsible for forwarding the job to one of available *rendezvous* peers in the SORCER environment and returning results to the requestor.
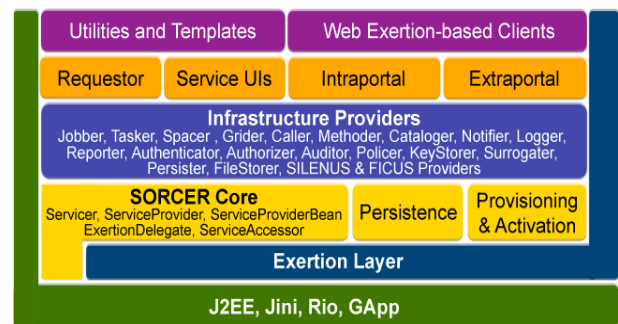

Figure 1 The SORCER layered functional architecture.

Thus implicitly, any peer can handle any job or task. Once the exertion execution is complete, the federation dissolves and the providers disperse to seek other collaborations to join. Also, SORCER supports a traditional approach to grid computing similar to those found, for example in Condor . Here, instead of exertions being executed by services providing business logic for invoked exertions, the business logic comes

from the service requestor's executable codes that seek compute resources on the network.

Grid-based services in the SORCER environment include `Grider` services collaborating with `Jobber` and `Spacer` services for traditional grid job submission. `Caller` and `Methoder` services are used for task execution. `Caller`s execute conventional programs via a system call as described in the service context of submitted task. `Methoder`s can download required Java code (task method) from requestors to process any submitted context accordingly with the code downloaded. In either case, the business logic comes from requestors; it is a conventional executable code invoked by `Caller`s with the standard `Caller`'s service context, or mobile Java code executed by `Methoder`s with a matching service context provided by the requestor.

4. Service Messaging and Exertions

In object-oriented terminology, a message is the single means of passing control to an object. If the object responds to the message, it has an operation and its implementation (method) for that message. Because object data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name (identifier) of the receiving object, the name of operation to be invoked, and its parameters. In the unreliable network of objects; the receiving object might not be present or can go away at any time. Thus, we should postpone receiving object identification as late as possible. Grouping related messages per one request for the same data set makes a lot of sense due to network invocation latency and common errors in handling. These observations lead us to service-oriented messages called exertions. An exertion encapsulates multiple *service signatures* that define operations, a *service context* that defines data, and a *control strategy* that defines how signature operations flow in collaboration. Different types of control exertions (`IfExertion`, `ForExertion`, and `WhileExertion`) can be used to define flow of control that can also be configured additionally with adequate signature attributes .

An exertion can be invoked by calling exertion's `exert` operation:

`Exertion.exert(Transaction)    :Exertion`, where a parameter of the `Transaction` type is required when the transactional semantics is needed for all participating nested exertions within the parent one, otherwise can be `null`. Thus, EO programming allows us to submit an exertion onto the network and to perform executions of exertion's signatures on various service providers indirectly, but where does the service-to-service communication come into play? How do these services communicate with one another if they are all different? Top-level communication between services, or the sending of service requests (exertions), is done through the use of the generic `Servicer` interface and the operation `service` that all SORCER services are required to provide— `Servicer.service(Exertion,    Transaction)`. This top-level service operation takes an exertion as an argument and gives back an exertion as the return value. How this operation is used in the federated method invocation framework is described in detail in .

So why are exertions used rather than directly calling on a provider's method and passing service contexts? There are two basic answers to this. First, passing exertions helps to aid with the network-centric messaging. A service requestor can send an exertion out onto the network—`Exertion.exert()`—and any servicer can pick it up. The servicer can then look at the interface and `PROCESS` operation requested within the exertion, and if it doesn't implement the desired interface or provide the desired operation, it can continue forwarding it to another provider who can service it. Second, passing exertions helps with fault detection and recovery, and security. Each exertion has its own completion state associated with it to specify who is invoking it, if it has yet to run, has already completed, or has failed. Since full exertions are both passed and returned, the requestor can view the failed exertion composition to see what method was being called as well as what was used in the service context input nodes that may have caused the problem. Since exertions provide all the information needed to execute a task including its control strategy, a requestor would be able to pause a job between tasks, analyze it and make needed updates. To figure out where to resume a job, a rendezvous service would simply have to look at the task's completion states and resume the first one that wasn't completed yet.

5. Authentication and Authorization with Exertions

Polymorphism let us encapsulate a request then establish the signature of operation to call and vary the effect of calling the underlying operation by varying its implementation. The Command design pattern establishes an operation signature in a generic interface and defines various implementations of the interface. In Federated Method Invocation (FMI), the three interfaces are defined with the following three commands: `Exertion.exert(Transaction):Exertion`—join the federation; `Servicer.service(Exertion, Transaction):Exertion`—request a service in the federation from the top-level `Servicer` obtained for the activated exertion;

`Exerter.exert(Exertion, Transaction):Exertion`—execute the argument exertion by the target provider in the federation.

These three commands define the Triple Command pattern that makes EO programming possible via various implementations of the three interfaces: Exertion, Servicer, and Exerter. The FMI approach allows for:

- the P2P environment via the `Servicer` interface,
- extensive modularization of programming P2P collaborations by the `Exertion` type,
- the execution of exertions by providers of the `Exerter` type, and
- vast common synergistic extensibility from the triple design pattern.

Thus, requestors can exert simple (tasks) and structured metaprograms (jobs with control exertions) with or without transactional semantics as defined in) above.

The Triple Command pattern in SORCER works as follows:

An exertion is invoked by calling `Exertion.exert(Transaction)`. The `Exertion.exert` operation implemented in `ServiceExertion` uses `ServicerAccessor` to locate in runtime the provider matching the exertion's `PROCESS` signature. If a `Subject` in the exertion is not set, the requestor has to authenticate with the `Authenticator` service. After the successful authentication the `Subject` instance is created and the exertion can be passed onto the network.

If the matching provider is found, then on its access proxy the `Servicer.service(Exertion, Transaction)` method is invoked. The matching provider first verifies if the requestor is authenticated; otherwise authenticate it with `Authenticator`. Then the provider consults the `Authorizer` service if the exertion's `Subject` is authorized to execute the operation defined by the exertion's `PROCESS` signature.

When the requestor is authenticated and authorized by the provider to invoke the method defined by the exertion's `PROCESS` signature, then the provider calls its own `exert` operation: `Exerter.exert(Exertion, Transaction)`.

`Exerter.exert` method calls `exert` either of `ServiceTasker`, `ServiceJobber`, or `ServiceSpacer` depending on the type of the exertion (`Task` or `Job`) and its control strategy. Permissions to execute the remaining signatures of `APPEND`, `PREPROCESS`, and `POSTPROCESS` type are checked with the `Authorizer` service for the executing `Subject`. If all of them are authorized, then the provider calls all the `APPEND`, next `PREPROCESS` methods, next the `PROCESS` method, and finally all the `POSTPROCESS` methods.

Individual service providers either `Taskers` or rendezvous peers, implement their own `service(Exertion, Transaction)` method according to their service semantics and control strategy,

however all of them federate with available `Authenticator` and `Authorizer` providers in a uniform way using Java Authentication and Authorization Service (JAAS) as described later in Section 6.

6. A Role-based Framework

In order to make the process of authentication and authorization easier in the federated environment, the framework is divided into two major modules to handle cohesive functionality separately – one for authentication and another for authorization. These two modules are implemented as individual service providers in the SORCER environment. Both these services utilize the common infrastructure of SORCER and operate on the key store module. Ideally the key store has to be built as a separate service provider in the near future. Multiple instances of both the services can be run for scalability. These services can communicate with other instance of the same service type in order to synchronize the access control lists, name and authorization SPKI certificates. Please note that digital certificates do not require a secure storage space but need to be verified when using them.

6.1 Architecture

The described Role-based Access Control Framework (RACF) uses JAAS but in the federated environment with distributed services. The authentication service acts as a login backend while the requestor handles the JAAS login callbacks. The authenticator utilizes any configured legacy authentication service to authenticate the users and assigns the JAAS subject with some public credentials. The authorizer gets this subject through the resource providing services. The authorizer maintains the access control lists in form of SPKI authorization certificates.

6.2 Authentication Service

The requestors should be authenticated with the authentication service before they access any resource providing service. Requestors can be authenticated against any existing user databases. In our approach any legacy authentication module supported by JAAS can be used as backend.

How the authentication service works is described below. The service requestor gets the user name and hashed password from the user and sends it across to the authenticator. The authenticator service authenticates the user using the backend legacy authentication service. Upon successful authentication it generates a name certificate using the public key of the user. If the user is

authenticating with the RACF system for the first time then a public/private key pair is generated for this user upon successful authentication with the legacy authentication service. This key-pair resides in any available, secure keystore. The authentication service then utilizes this key pair to generate a name certificate for the user. This name certificate is used as an authentication token since it is signed by the authentication service.

If the user is a returning user, his public key is simply fetched from the keystore and the name certificate is signed with the private key of the authentication service after he is authenticated with the legacy authentication service. All instances of authentication service use the same private key. The authentication service has to keep this private key secure, either by storing it in the key store provider or managing it by itself. The name certificate is then sent to the requestor. The requestor can use this name certificate along with the requests it makes for any resource to prove its identity. Any authorization service verifies the signature of the authentication service before providing any resource. A validity specification on the name certificate can be used to specify a time frame only within which the token is valid. After this timeout the requestor has to renew this token for further usage.

If the requestor has multiple accounts with the legacy authentication services then it is possible to have a single identity for this requestor in SORCER. For example, if a user has a UNIX account at the Computer Science Department and also a Windows account at the university then he/she can choose to have a single identity in SORCER. This is possible since we use a public-private key pair for the user, using which we identify the user after authentication. This allows him/her to access his/her resources using any legacy authentication service. It will not hinder the user from accessing his/her resources when one of the legacy authentication services cannot be used due to network problems or what so ever issues. Also users from any domain can be authenticated and issued a key pair, there by breaking the domain barrier.

All requestors to be identified by its name can use the same name certificate created by the authentication service. When they wish to include the requestor in any role they can look for the requestor's name certificate to be included. Requestors can issue authorization grants without using name certificates as well, if they wish to, by using the public key as subject in their authorization certificates instead of their local names.

6.3 Authorization Service
The authorization service keeps up the access control

lists and verifies any request to access resources. The Authorizer itself does not guard the resources, but only provides a way to verify if the subject in question has the permission to access the resource. The resource provider itself by any means should keep the actual resource secure. The authorization service also requires the keystore module. The keystore can be run as an independent service in the federated environment and multiple instances of it can be run for scalability. All keystore modules will have to synchronize the keys available in order for the authorization system to work.

When a request for a resource arrives at an actual service provider, such as SILENUS, it calls for the authorization service to verify the user identity and determine if the user is allowed to access the requested object. The authorization service verifies the user identity by simply verifying the signature of the name certificate sent across by the resource provider. This name certificate is supposed to be signed by the authentication service, whose public key the authorizer is aware of. The request is denied if the verification fails. Once this signature is verified, the authorization service proceeds to determine the access control on the requested object.

Access control objects are stored as authorization certificates in the keystore. These authorization certificates indicate the issuer, the subject, a tag, a bit field indicating if the subject can delegate this authorization grant and a validity specification. The tag specifies the object on which the authorization is granted and what type of access is allowed. The format of tag is not defined precisely in the certificate format, it could be any object suited for the application. As a case in point, the SILENUS file system puts the file name and access type (primarily read and/or write) in a way that authorization service can understand. In order to speed up the process of access checking, the authorization certificates are stored as 5-tuples, if the storage area is secure. When that is not possible they can be stored as certificates and the individual fields can be determined when they are read.

The authorization service requests for the related authorization certificates from the keystore and runs a resolution algorithm to determine if the access is allowed. The algorithm checks if it can find a chain of delegated authorizations from the resource owner to the requestor under question on the requested object. If it can resolve the chain the access is allowed, if the chain resolution fails the access will be denied. The Java implementation of SPKI/SDSI is utilized for this purpose. This library defines ways to create name certificates, authorization certificates, tags and a keystore to store SPKI certificates along with many necessary mechanisms.
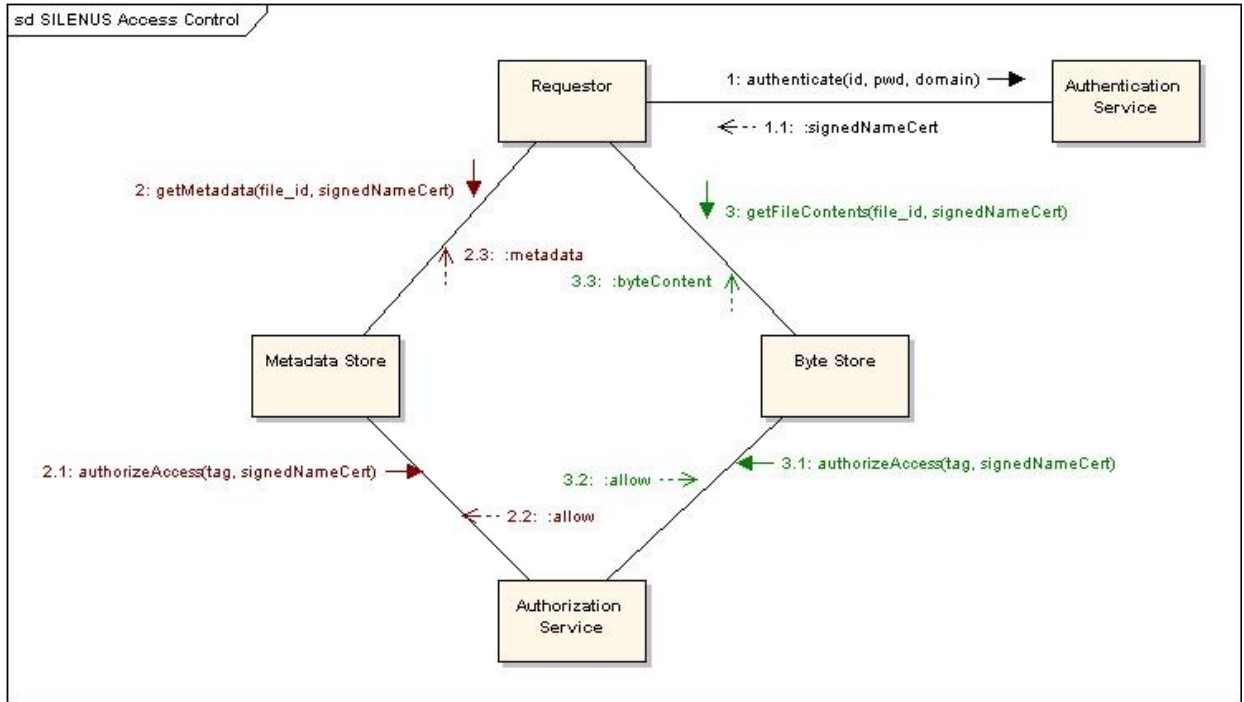
Figure 2. SILENUS utilizing the role-based access control framework

6.4 Group formation

Any user, when they wish to share their resources with other users, has to create a group and then allow this group to access the resources owned by the owner. The group is local to the current user and is not applicable to the entire system; it is valid only in the user's local namespace. This group may include only one user in which case it will be like a local name for the subject. For example the user Alice can create a group named "friends" and add Bob and Carol to it. This is done by issuing name certificates with Bob's public key and Carol's public key as subject respectively. Alice can then issue an authorization certificate that specifies the authorization in its tag field and the local name "friends" in the subject. The tag contains the resource objects id and the access control specification.

A name certificate is represented using the notation,

```
Issuer localname -> Subject
```

And authorization certificates are represented as

```
Issuer tag -> Subject
```

So our example will be

```
Alice friends -> Bob
Alice friends -> Carol
Alice (+read document.txt) -> friends
```

The tag indicates permission to read document.txt and the ability to delegate this permission to others by "friends".

The subject need not be a public key always; it could be a list of names too. Let's say Dave wants to let Alice's friends read his files too. He may issue a certificate

```
Dave (read mydoc.txt) -> Alice friends
```

This allows Alice's friends to be able to read Dave's mydoc.txt. The local name of Alice, "friends" has been used by Dave here.

7. Deployment in SILENUS

The role-based access control framework has been deployed in the SORCER federated environment and validated successfully in the SILENUS file system with a role-based file browser. The framework is built in Java using the JSDSI library . SILENUS provides a federated file system for SORCER. The system itself is made up of multiple service providers that collaborate with each other to provide a service-oriented file system. The most important ones are the metadata store and byte store providers . As the names indicate the metadata store persists the metainformation of the files such as file name, size, and mime type. It also saves a unique identifier for each file. When a byte store is contacted with this unique id, the file contents can be obtained. In order to provide access control to the SILENUS file system both metadata store and byte store have to be secured and have to utilize the access control framework. The SILENUS file system instead of exposing the internal modules follows a façade design pattern where a façade service acts as the SILENUS entry service provider. The façade provides service UI, accepts requests from requestors, and forwards them to the appropriate service provider. The interaction of

SILENUS with RACF has been depicted in Figure 2. For brevity some SORCER components are omitted.

The SILENUS façade manages all proxies for the underlying services. It acts as an entry point service for SILENUS service providers with multiple façade in the system at any time. All login requests are sent to the authentication service. The metadata store and the byte store depend on the authorization service to control access to their resources. When a requestor makes a request to access a file's content it sends the file information and signed public key that are obtained from authentication service to metadata store. The metadata store then requests the authorization service to determine if this access is allowed. Only when the authorization service signals go ahead the metadata store entertains the request. The byte store also works in a similar way with the authorization service.

The SILENUS façade provides an interactive user interface to access files without exposing the user to any complex access control behavior. In fact user-friendliness has been one of the major requirements for SILENUS.

The role-based federated access control framework has been utilized in a similar way for exertion-oriented programming by other SORCER services to provide a scalable and reliable authentication and authorization services so that resource providers do not have to handle it themselves in an ad-hock manner.

8. Conclusions

An access control mechanism is needed in federated environments where conventional existing solutions do not scale well. Most existing access control solutions are tightly coupled with the service provider or a part of a service provider and as such are not meant for the entire federated environment. We propose a new federated implementation of the JAAS framework. The proposed solution scales well with increasing resources and service providers. Along with providing a federated access control framework we also have concentrated on user collaboration where users can share resources with other users irrespective of the administration domain they come from. SPKI certificates are used to create local namespace there by avoiding global naming conventions and central certificate authorities. SPKI also provides the facility to delegate authorization grants across exertion-based federations. Users and requestors can create roles that exist only in that user's namespace and can assign permissions to these roles thereby avoiding the involvement of an administrator for day-to-day operations of users, which is highly required in a self-sustaining environment like SORCER. A successful validation of the presented framework was deployed in the SILENUS federated file system along with the same federated JAAS-based approach for all SORCER requestors and providers.

9. References

[1] Sobolewski M., Federated Method Invocation with Exertions, Proceedings of the 2007 IMCSIT Conference, PTI Press, ISSN 1896-7094, pp. 765-778, (2007). Available at: http://sorcer.cs.ttu.edu/publications/papers/96.pdf

[2] Sobolewski, M., SORCER: Computing and Metacomputing Intergrid, 10th International Conference on Enterprise Information Systems, Barcelona, Spain (2008). Available at: http://sorcer.cs.ttu.edu/publications/papers/2008/iceis-intergrid-08.pdf

[3] Thain D., Tannenbaum T., Livny M. Condor and the Grid. In Fran Berman, Anthony J.G. Hey, and Geoffrey Fox, editors, Grid Computing: Making The Global Infrastructure a Reality. John Wiley (2003).

[4] Berger, M., Sobolewski, M., Group-based Security in a Federated File System, 2nd Annual Symposium on Information Assurance, Albany NY, June 6-7, 2007, pp. 56-63 (2007).

[5] Berger, M., Sobolewski, M. 2007, Lessons Learned from the SILENUS Federated File System, Complex Systems Concurrent Engineering, Loureiro, G. and Curran, R. (Eds.), Springer Verlag, ISBN: 978-1-84628-975-0, pp. 431-440 (2007)

[6] Java Cryptography Architecture, Key Management http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyManagement

[7] Jini architecture specification, Version 2.1. Available at: http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html Accessed on: March 1, 2008.

[8] Morrie Gasser, "Building A Secure Computer System", pages 45-58

[9] Ravi S. Sandhu, Edward J. Coynek, Hal L. Feinsteink and Charles E. Youmank, Role-Based Access Control Models. IEEE Computer 29(2): 38-47, IEEE Press, 1996

[10] Joel Weise, "Public Key Infrastructure Overview", Sun BluePrints, SUN Microsystems, http://www.sun.com/blueprints/0801/publickey.pdf

[11] Carl M. Ellison, Bill Franz, Butler Lampson, Ron Rivest, Brian M. Thomas, Tatu Ylonen. SPKI Requirements, SPKI Certificate Theory, Simple public key certificate, SPKI Example. Internet draft. October 1998. http://world.std.com/~cme/spki.txt

[12] Carl Ellison, "Establishing Identity Without Certification Authorities"

[13] JSDSI: A Java Implementation of Tools for SDSI Certificate Management. http://jsdsi.sf.net/

[14] Rivest and Lampson, SDSI A Simple Distributed Security Infrastructure

[15] Garms J., and Somerfield D., Professional Java Security Wrox Press, (2001)