Adding Discretionary Access to Remote Method Invocation

Fernando Magno Quintão Pereira

August 14, 2005

Abstract

This paper describes the implementation of an object oriented middleware that allows the application developer to regulate the use of individual remote methods by means of access control lists. Such platform has been implemented as an instance of Arcademis, a framework for middleware development. The objective of this case study is twofold. Firstly, to demonstrate how frameworks and design patterns can be synergistically combined in order to facilitate the implementation of distributed software. Secondly, to point similarities between the architecture of object oriented middleware, such as Java RMI, and distributed authentication systems, such as Kerberos, in order to argue that discretionary access control can be added to the commercial middleware platforms as a natural extension of the remote method invocation paradigm.

1 Introduction

Remote method invocation is an extremely useful abstraction technique because it hides the application developer from the idiosyncrasies typical from the distributed environment, such as data representation, and object location. In these systems, the programmer has the illusion that all the objects of his interest are located in the same address space, although they may actually be distributed across different processes, or even different computers.

Since the pioneering concept of distributed objects introduced by Modula 3 [1], several different platforms have emerged in order to provide application developers with increasingly more expressive abstraction capabilities. Three systems, however, constitute a standard in this area [4]: CORBA [10], Java RMI [19], and Microsoft .NET Remoting [9]. Despite presenting several differences, these platforms follow the same general model according to which client objects can remotely invoke methods of server objects, also called service providers. In order to allow the location of service providers, these middleware use centralized directories of names, whose location is well know.

The previously cited middleware systems provide security mechanisms as optional features; thus, security policies must be customized, or even implemented, by the software engineer at the application level. This is a complex and error prone task, due to the inherent difficulties that permeate the computational security area. Therefore, in order to implement more robust and efficient security protocols, this paper defends that fundamental services, such as authentication and confidentiality, should be guaranteed by the core of the middleware system, instead of being provided by the application developer. Relying on this objective, we describe the experience acquired during the development of `SaMi` [1], a remote method invocation platform in which every network object is bound to an access control list.

`SaMi` has been implemented as an instance of Arcademis, a framework for middleware development [12]. A framework is a set of cooperating classes and interfaces constituting a semi-complete application, which can be customized by programmers [6]. Libraries and frameworks are similar concepts, because both are formed by sets of components; however, while in a framework these components cooperate to describe

[1] `SaMi` is a word coined from the initials of *Secure Access in Method Invocation*

the skeleton of an application, in a library they are relatively autonomous. Microsoft Foundation Classes, and Java Swing are examples of frameworks. The use of frameworks in middleware development has been advocated in recent years [11, 17, 18]. Examples of platforms built from collections of reusable components are ACE TAO [14] and Open ORB [3].

Arcademis has been used in the instantiation of middleware platforms targeting different sets of requirements. Examples of Arcademis' instances are RME [12] and Aries [13]. The later platform allows the application developer to specify different tactics during remote method invocation. The former provides the CLDC configuration of Java 2 Micro Edition with a remote method invocation system. Although the framework defines primitives for protecting distributed data, none of these middleware make use of them. `SaMi`, on the other hand, makes extensive use of such primitives in order to provide certification and authentication to the application developer. Moreover, `SaMi` is the only instance of Arcademis that makes use of computational reflection, which allows the user to customize security polices during execution time.

During the development of `SaMi`, we perceived similarities between distributed authentication systems and remote method invocation. In order to outline such similarities, Section 2 presents a general overview of remote method invocation systems, and Section 3 summarizes basic principles of a simple authentication protocol. In order to demonstrate how the Arcademis framework can facilitate the development of middleware platforms, Section 4 outlines some aspects of its architectures, and gives further details about `SaMi`'s implementation. Section 5 discusses the adopted authentication protocol. Section 6 introduces an example application that can benefit from the functionalities provided by `SaMi`. Related works are presented in Section 7. Finally, Section 8 concludes this document.

2 Overview on Remote Method Invocation

There exist different types of middleware platforms, for instance, message oriented, or tuple space oriented; however, the object oriented paradigm seems to be the dominating model in the software industry. According to this model, a client object uses intermediate components in order to invoke methods on remote objects. Two of these components are the stub, that exist on the client side of a distributed application, and the skeleton, that is located on the server side. The stub acts as a local proxy for the remote object, and its function is to forward to the server all the remote calls made by the client. The skeleton represents the invoking client to the remote object, acting as an adapter. It receives messages containing information about remote invocations and determines what method of the server should be executed. Although application developers have the illusion that the methods are being locally processed, each remote call is actually transmitted by the stub to the skeleton and then to the implementation of the remote object. The results of remote invocations are transmitted across the opposite path.

Another essential part of a remote method invocation system is the lookup service. Object oriented middleware can be described as service-oriented architectures. In this model there exist three different actors: service providers, service requesters and discovery agencies. Service providers are represented by remote objects, whereas requesters are represented by clients in general. The third actor, also called lookup service, or name service, allows clients to find the objects in which they are interested. In general, service providers register themselves using a `publish` operation, while clients look for distributed objects by means of a `find` operation. A schematic view of a remote call is shown in Figure 1.

3 A Glance at Authentication Systems

The distributed environment presents several characteristics that make reliable communication a challenging activity. Some examples of such characteristics are sharing of resources, unknown perimeter, anonymity of users and the existence of many potential points of attack. In a large network, where new

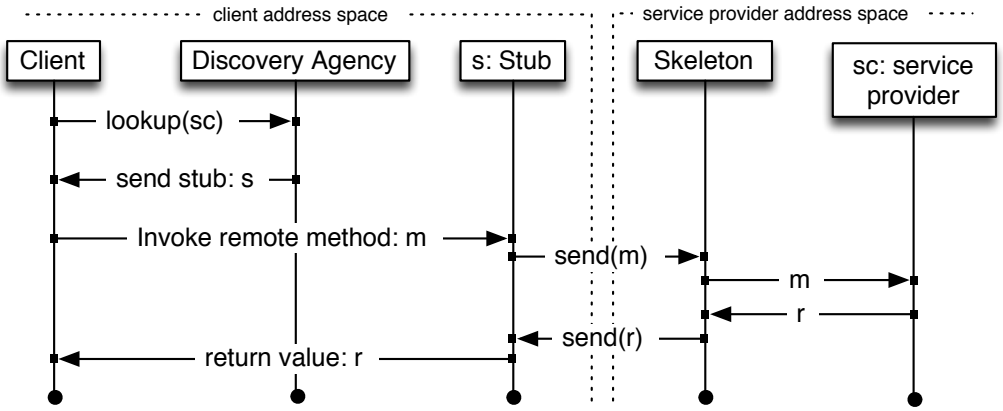


Figure 1: Interactions between stub and skeleton in a remote method invocation system.

users appear all the time, authentication is a central issue. The main objective of an authentication system is to assure the identity of a remote user.

Cryptography is one of the basic mechanisms of authentication. Cryptographic algorithms are used to assure the secrecy of information, and to guarantee the authenticity of its provider. The cryptographic protocols can rely on private or public keys. The later strategy is adopted in `SaMi`. In a classical public key system, each user has two keys: a secret key (D), and a publicly available key (E). As an assumption, messages encoded with the private key can only be recovered with its public counterpart, and vice-versa.

One of the main problems of this cryptographic scheme is how to publish the public key, and how to guarantee the identity of the publisher. A solution normally adopted is to delegate this task to a central authority. The role of such entity is to issue certificates that testify the authenticity of the owner of a key. Every time a specific user needs to proof its identity, he asks the central authority for a certificate, and sends it to the interested party. Figure 2 gives a general overview of this protocol. Further details are given in Section 5.

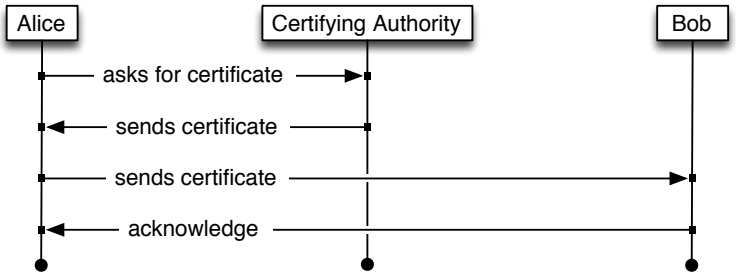
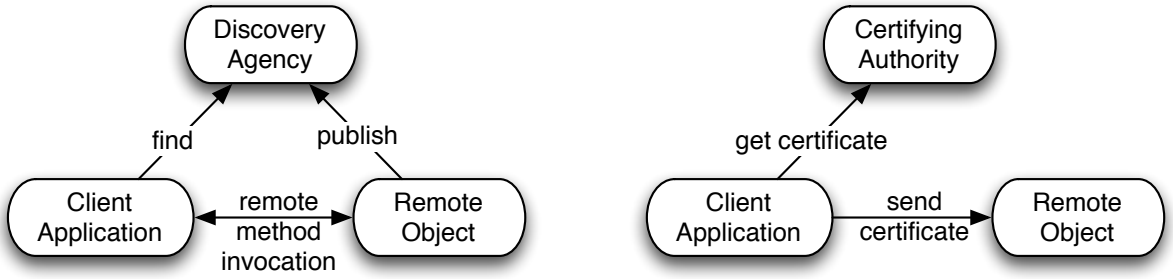


Figure 2: Authentication via certifying authority: Alice tries to proof her identity to Bob.

It is possible to trace an analogy between the authentication system just described, and the service-oriented architecture presented in Section 2. In both models, there exists the concept of a central entity that can be consulted by marginal users. In the authentication system, such entity is the certifying authority. In the service-oriented paradigm, it is the discovery agency. Therefore, in order to expand the security capabilities of an object oriented middleware, it is natural to ascribe to the discovery agency the capacity of issuing certificates that clients can use to proof their identity to service providers. Relying on this design, the discovery agency used in the implementation of `SaMi` is both an authentication authority,



(a) **(b)**

Figure 3: A comparison between the (a) service oriented architecture, and (b) a authentication system.

and a directory of names. Every time the client executes a search operation, it obtains, when the query is successful, the stub referencing the remote object, and a certificate that it can use to prove its identity to the service provider.

4 The Arcademis/SaMi Architecture

SaMi has been build as an extension of RME [12], which is an instance of Arcademis targeting the CLDC configuration of J2ME. A distributed system built on top of Arcademis is structured on three abstraction levels. The first of these levels is composed of the framework components. Essentially these are abstract classes and interfaces, although Arcademis also provides concrete components that can be used without further extensions. The second level is represented by the concrete middleware platform, obtained as an instance of Arcademis. The framework defers to this level decisions such as the communication protocol and the serialization strategy that will be adopted. Finally, the third programming level comprises all the components that provide services to end users. These components constitute what is normally called a distributed application. SaMi constitutes the second level of an Arcademis based system.

Each instance of Arcademis has a central component called ORB. This element is implemented as a *Singleton*, a design pattern that limits the maximum number of instances of a given class to exactly one [5]. The ORB can also be characterized as a set of *Object Factories*. An Object Factory is another design pattern that is used to create instances of objects. The main advantage of this pattern is to make it easier to change a component's implementation without interfering in other modules of the system. For example, in Arcademis, all communication channels are created by an Object Factory. In order to modify the transport protocol used by the middleware, for instance, from TCP to UDP, it is sufficient to change the channel factory bound to the ORB. Because the factory preserves the channel interface, the other components of the platform need not to be changed.

The first action that must be performed by every application based on Arcademis is to configure the ORB, which means to define the set of factories that will be used during the middleware execution. It is the configuration of the ORB that will define all the behaviors of the middleware system. The framework does not requires that an implementation for all of its factories be provided by its instances. Indeed, the programming philosophy adopted in Arcademis advocates a minimal use of components, that is, each middleware platform must use only the elements that will be necessary for its correct execution. For instance, while the framework defines 17 object factories, the server implementation of RME uses 14, and the implementation of SaMi uses 15. The client version of RME has been designed for a cell phone, and cannot handle incoming connections. Because of this simple design, its implementation uses only 9

factories.

Besides stubs, skeletons, and lookup service, discussed in Section 2, Arcademis defines several other components that collaborate to outline the middleware architecture and to support customizations. The most important of these elements are represented in Figure 4. In this figure, the arrows indicate the flow of data during a remote call. The `invoker` is responsible for emitting remote calls, whereas its server counterpart, the `dispatcher`, is in charge of receiving and passing them to the skeleton. The `Scheduler` is used whenever necessary to order remote calls according to their priorities. The network layer, in Arcademis, is represented by a set of components that constitute the transport protocol, serialization protocol and middleware protocol. Connections are established by two components: the `Connector` and the `acceptor`. Request *senders* and *receivers* determine the reliability level the middleware provide to distributed applications. Finally, the `Activator` determines how an object is made ready for receiving remote calls.

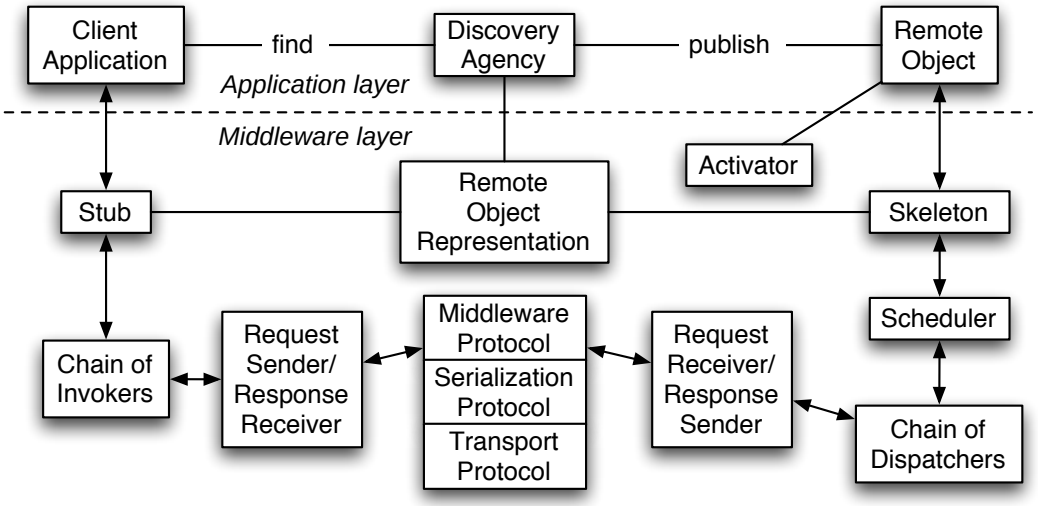


Figure 4: Representation of the main components of Arcademis. Double arrows indicate the flow of data during a remote method invocation.

Arcademis makes extensive use of design patterns. For instance, *object factorY* and *singleton* [5] are used in the ORB implementation. The *acceptor-connector* pattern [15] is used to handle remote connections. Stubs are implemented as *proxies*, and skeletons as *adapters* [5]. The service providers have been designed according to the *Active object* [16] pattern, and messages are implemented according to the *Command* pattern [5]. However, *object decorator* [5] is the most used pattern in the Arcademis architecture. A decorator allows to aggregate extra functionalities to an object while preserving its type. The same object can be encapsulated by different decorator, each one further enhancing the set of available functionalities. In Arcademis, Invokers, Dispatchers and Channels support decorators. Therefore, it is more appropriate to talk call such components chains of objects. A chain is determined by the basic component and all its decorators.

The component called dispatcher has particular importance in `SaMi` design. In Arcademis, the implementation of this component determines, for example, if remote calls are passed directly to the skeleton or to other components, such as a scheduler. In `SaMi`, every dispatcher is augmented with an access control list, which is implemented by means of a Dispatcher decorator: the `SecureDispatcher`. This decorator contains operations to verify certificates, and to manipulate the list of access privileges.

Several of the Arcademis/RME components were reused in the implementation of `SaMi`. Arcademis

defines 79 basic components, and RME defines 56. Most of RME classes are implementations of decorators and messages that comprise the middleware protocol. `SaMi` redefines 33 of the Arcademis classes, and uses 14 other classes to implement cryptographic routines. All the RME components related to serialization of objects, and transmission of data have been reused in `SaMi` without modifications.

4.1 The Reflective Interface

The access control lists are implemented at the middleware layer; therefore, they are not visible in the application level. However, the distributed scenario is a dynamic environment, and sometimes the knowledge necessary to configure the security parameters of a service provider are only available at runtime. `SaMi` allows such dynamic modifications by means of *Reification*. To reify a software means to expose to the user details of its implementation. In `SaMi`, reification is used to allow the application developer to change the access control list of a remote object. A component called *Reificator* allows the application developer to inspect and alter internal properties of the middleware. For instance, the reificator can scan the chain of dispatchers looking for an object of type `arcademis.security.SecureDispatcher`. The search uses Java computational reflection to discover the dynamic type of the dispatchers. The reification interface provides the following list of operations to the application developer to customize security parameters:

- `getRemoteMethodList(RemoteObject o)`: this method allows the application developer to get a list of all the remote methods implemented by a service provider (o). Each method has an index, which is used to change the method's access policy.
- `setLookupServicePublicKey(Key k, RemoteObject o)`: this method gives to the dispatcher of the remote object access to the discovery agency's public key. The dispatcher needs the public key in order to authenticate the certificates it receives.
- `grantPermission(int m, string c, RemoteObject o)`: this operation grants to the client c access to the method of index m.
- `revokeAllPermissions(int m, RemoteObject o)`: this method revokes all the permissions associated to method m.
- `revokeAllPermissions(RemoteObject o)`: this method revokes all the entries in the access control list of the remote object.
- `grantFreeAccess(RemoteObject o)`: this method makes all the remote methods of the service provider publicly available.
- `grantFreeAccess(String c, RemoteObject o)`: this method grants the client free access to all the remote methods of the service provider.
- `grantFreeAccess(int m, RemoteObject o)`: this method makes the operation of index m publicly available.

5 The Cryptographic Protocol

In `SaMi`, each remote method is protected by an access control list. In order to use a remote method, a client needs a certificate, which is generated by cryptographic algorithms. The implementation of `SaMi` uses the gnu cryptographic framework [8] to encrypt messages. The encrypting algorithm is RSA, with keys of 1024 bits. In the implementation of `SaMi`, a service provider does not authenticate itself to the

client. That is, while the service provider can be sure about the identity of the client, reciprocity does not hold: the client is not able to assure the server's identity. This approach was chosen because it simplifies the implementation of the system. If necessary, it is possible to extend `SaMi`'s protocol to allow server authentication.

In order to be able to obtain certificates, the client must register its public key in the discovery agency. This operation is called *key's debut*. Once the client announces its key, it can receive certificates from the discovery agency. The general representation of a certificate is $C = \{name \parallel E_c \parallel T\}D_{NS}$. Certificates are encrypted with the private key of the name server (D_{NS}), and contain three basic fields: the client's name ($name$), the client's public key (E_c), and a time stamp T. Every time a client executes a lookup operation, in addition to the stub of the remote object, it receives also a new certificate. The lookup operation is represented in Figure 5.

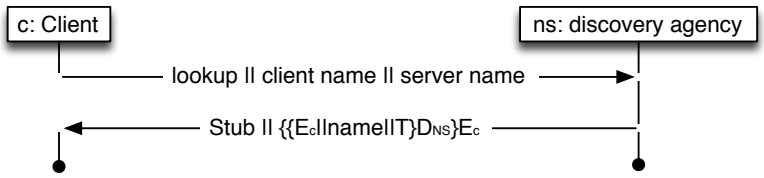


Figure 5: The lookup operation.

`SaMi`'s certificates are valid for at most one day. The discovery agency updates its key in regular periods of 24 hours. The time stamp embedded in every certificate informs the service provider until what moment that certificate should be accepted. This policy makes difficult to break the encryption scheme by brute force attacks. However, it may deny access to an authentic client, if she is not using a updated certificate. Another problem with this approach is the necessity of synchronized clocks. Synchronization in a distributed environment is a hard problem, and, in order to mitigate it, the discovery agency provides a clock service that can be remotely invoked.

In addition of updating its own key, the name server also revokes the registration of clients after 24 hours. This obliges clients to be constantly renewing their cryptographic keys. The update of a key is not a trivial operation, because the discovery agency must confirm the client's identity before revoking the old key and registering the new one. Such confirmation is made by means of callbacks. Given two remote objects, A and B, a callback happens if A invokes method m_B on B, and in m_B's code, B invokes a method m_A on A. The second call, i.e. $B \rightarrow A.m_A$ is called a callback. When a client has to update its key, it passes to the discovery agency a remote object, the *validator*, which is used by the name server to confirm the client's identity. The validator is a remote object, and the name server uses it to check if the client really wants to update its key. In the updating message, the client sends a random number R_A to the server. The server generates a new random number (R_B), and sends both, encrypted with the client's public key to the client. Only the actual client can recover these numbers. After deciphering the validation message, the client sends to the server $R_B - 1$, encrypted with the server's public key. This verification step avoids the risk of man-in-the-middle and replay attacks. The updating protocol is depicted in Figure 6.

Like the key's update, a remote method invocation is also performed in four steps. This protocol allows the server to confirm the identity of the client, although it does not permit the opposite: the client cannot confirm the server's identity. Service providers have access to the public key of the discovery agencies in which they are bound. Upon receiving a remote call, the service provider extracts the contents of the certificate, using the discovery agency's public key, and, by consulting the access control list of the invoked method, it determines if the client has permission to execute it. If the permission is not granted, the server reports an exception to the client; otherwise, it starts a confirmation process similar

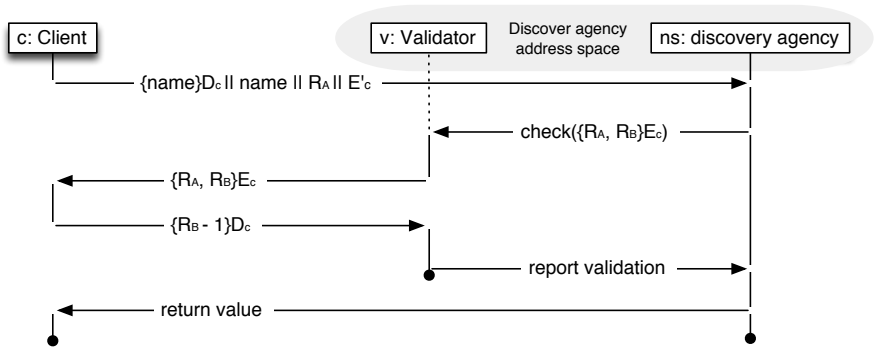


Figure 6: Updating of client's key.

to that used in the update of keys. However, in this case, the validator is not necessary, because the confirmation protocol has been implemented at the network level, by the request sender and request receiver (Figure 4). Because the directory of public keys is not available to the network components, the protocol of key update demands a second connection between discovery agency and clients.

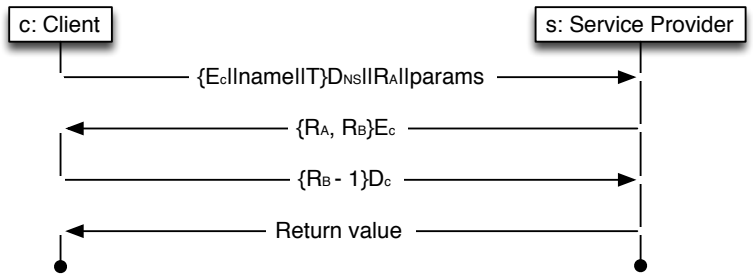


Figure 7: The remote invocation protocol.

The protocol just described presents a number of properties that allow it to withstand some attacks in the distributed environment:

- *Private keys are never communicated on the network.* In order to emphasize this property, the object that implements private keys can not be serialized. That is, they cannot be changed into a sequence of bytes, and recovered later.

- *Protection against spoofing.* Every service provider uses the public key of the discovery agency to assure the authenticity of certificates. Therefore, only the central lookup service can grant clients access to protected remote services.

- *Limited period of validity.* Certificates contain time stamps that can be verified by the service provider. This aims to neutralize brute force attacks on intercepted messages. In addition, clients are obliged to change their keys in regular time intervals.

- *Client confirmation to prevent replay attacks.* After gaining access to the service provider, the client must pass thru an extra confirmation step in which encrypted random numbers are exchanged. This avoids that a third entity intercepts messages from the client and use them to cause any action in the service provider.

6 An Example Application

In this section we will discuss a hypothetical software application that can take benefit from the security mechanisms provided by `SaMi`. This software is intended to support a review committee, in which members can post, update, and write evaluations about articles. Every article is represented by a single distributed object, which provides four different methods: `getAbstract()`, `getPDF()`, `sendReview()`, and `update()`. The first operation simply returns a string of characters containing the paper's abstract. The second gives the caller a full PDF version of the article. The third method attaches a review to the paper, and the last operation updates the complete text. Figure 8 shows the implementation of this interface. The syntax provided by `SaMi` is similar to the one adopted in Java RMI. Remote interfaces must extend the `Remote` type, and remote methods must throw `ArcademisException`.

```
import java.arcademis.*;

public interface Paper
extends Remote {

  public String getAbstract()
  throws ArcademisException;

  public File getPDF()
  throws ArcademisException;

  public void sendReview(String s)
  throws ArcademisException;

  public void update(File pdf)
  throws ArcademisException;
}
```

Figure 8: The interface for an object that represents an article electronically available.

Although all the remote methods are declared in the same interface, they target different actors. In order to better illustrate this fact, assume that all the papers are part of the proceedings of a conference. With this assumption in mind, only the paper's author should be allowed to update his article. Furthermore, just members of the program committee should be able to write reviews about the text, and only people registered in the conference should be allowed to get the full PDF version of the article. Assume also that, according to the conference's policy, any person is able to get the abstract of a publication.

It is not simple to implement this set of constraints in traditional Java RMI, CORBA, or .NET Remoting. This application demands an access control list in which different sets of methods could be assigned to different actors by the remote object creator. Upon creation, the access control list of the distributed object is configured, and the object's name is registered in the discovery agency. If a client attempts to execute a method whose access it has not acquired, it will receive an exception instead of the expected return value. The access control list of a remote object must be configured by the service provider. In Figure 9 there is an example code in which different privileges are defined. The user "Rhyme" can access all the remote methods, whereas the user "Ann" can only get the abstract and PDF versions of the paper.

In Figure 9, `ConcretePaper` is an implementation of the interface shown in Figure 8. The code in lines 8, 9, and 10 defines the middleware configuration that will be used. To configure an instance of Arcademis means to define the set of factories associated to the ORB (Section 4). In line 14 the remote object is being bound to the name `‘‘P23’’` in the discovery agency. When no host name is given, the lookup interface seeks for a discovery agency in the local host. Client applications will have to look for `‘‘P23’’` in order to invoke the remote methods of object `p`. In lines 17, 18, and 19 the public key used by the discovery agency is sent to the dispatcher assigned to `p`, so it can verify certificates. The remaining lines define the access policy adopted by the service provider. Methods are discriminated by integer indexes. In `SaMi` remote methods are indexed, and in order to know the mapping between indexes and methods, the Reificator provides the operation `getRemoteMethodList`.

In order to illustrate the impact of the security protocol in the communication middleware, Table 1 compares the performance of `SaMi`, Java RMI, and RME (described in Section 1). `SaMi` is tested with two different cryptographic algorithms. The first is based on a simple XOR mask, and its running time

```
1. import rme.SNService.*;
2. import rme.security.*;

4. import arcademis.security.Key;

6. public class Server {
7.  public static void main(String a[]) {
8.   SecureConfigurator c =
9.   new SecureConfigurator();
10.  c.configure();

12.  ConcretePaper p=new ConcretePaper();
13.  p.loadArticle("article.tex");
14.  SecureLookupService.bind("P23", p);
15.  p.activate();

17.  String h = "localhost";
18.  Key k=SecureLookupService.getPublicKey(h);
19.  Reificator.setLookupServicePublicKey(k, p);

21.  // no one access this remote object
22.  Reificator.revokeAllPermissions(p);

24.  // Rhyme can access all the methods.
25.  Reificator.grantFreeAccess("Rhyme", p);

27.  // each one can access the method getAbstract
28.  Reificator.grantFreeAccess(0, p);

29.  // Fernando can access getAbstract, getPDF
30.  Reificator.grantPermission(1, "Ann", p);

32.  // Sandy can access almost everything
33.  Reificator.grantPermission(1, "Sandy", p);
34.  Reificator.grantPermission(2, "Sandy", p);
35. }
36.}
```

Figure 9: The server code, where the access list is configured.

reflects the impact of the extra messages in the communication protocol. The second algorithm is based on the RSA public key system. The increased running time is due to the cryptographic computations. Table 1 also shows the size of messages exchanged between clients, server and discovery agencies during service registration, service lookup and remote call. The running time has been obtained by invoking the methods `getAbstract` and `getPDF` defined in Figure 8 1000 times. The message size refers to a single invocation. In this experiment, service provider and client are been executed in two PowerPC 1.25GHz with 1GH DDR SDRAM, connected by a 10Mb/s LAN. The four sizes of a remote method calls do not include the size of the return value.

	Running time (s)	Size of Registration message (byte)	Size of lookup message (byte)	Size of Method call message (byte)
Java RMI	1.066	a	a	a
RME	1.097	593	592	277
`SaMi` XOR	1.669	664	4701	1299
`SaMi` RSA	15.170	664	4701	1299

Table 1: Comparative performance of Java RMI, RME and `SaMi`

7 Security on Object Oriented Middleware

Commercial middleware systems normally provide primitives for implementing security directives; however, these are not part of their core architecture. In order to illustrate such fact, this section address security strategies adopted in .NET Remoting, CORBA, and Java RMI. First considering .NET Remoting, this platform has no built-in support for user authentication or integrity protection [2]. Instead, it provides a range of technologies that can be used at the application level to enhance security. As a result, it requires a non-trivial amount of programming work, often resulting in complicated systems. However, it is expected that a future release of the .NET framework will incorporate cryptographic algorithms in its TCP channel [2].

The security mechanisms found in Java RMI are an extension of those provided in the standard Java

library. By implementing instances of the `SecurityManager` class, the application developer can define which classes can be downloaded, and which remote addresses can be reached. These functionalities, however, are not part of the default services provided by Java RMI. In contrast, JINI, another member of the Java distributed family, provides several services aiming to guarantee server/client authentication, integrity and confidentiality. JINI also is a service-oriented architecture, and its implementation uses Java RMI. In JINI the client must acquire the proxy from the service provider in order to get access to a specific service. This model gives JINI great flexibility when necessary to define security policies. For instance, in JINI the application developer can define a set of security requirements, which must be always satisfied, and a set of preferences that should be attended when possible. Moreover, JINI permits to assign separate access privileges to each proxy. One of the disadvantages of JINI's security model is due to its complexity: the application developer is forced to deal with a number of interfaces and he often must configure non-trivial security parameters.

The CORBA specification defines a very complete set of security directives, but its implementations are not obliged to incorporate them. For instance, IONA ORBIX, a commercial implementation of CORBA, gives the application developer some options to incorporate security. It makes available cryptographic algorithms such as RSA and Diffie Hellman, and their design foresees the use of the Kerberos [7] authentication system. But these tools are not part of the default implementation of IONA, and must be configured by the application developer.

In general, the implementation of security mechanisms decrease the performance of distributed systems. For example, cryptographic algorithms demand memory and processing time. Therefore, one of the advantages of making the security features optional extensions of the middleware is the greater flexibility: if the protection layer is not necessary, it is simply not used. However, it is possible to present reasons that justify the design of platforms such as `SaMi`, which have security mechanisms implemented at the middleware level, instead of the application level.

Firstly, as pointed in Section 3, object oriented middleware offer an infrastructure on which security can be naturally incorporated. Secondly, the implementation of a security layer is a complex and error prone task, and its frequent delegation to the application developer can result in vulnerable distributed applications. Furthermore, if the middleware already implements protective functionalities, the software engineer can focus on the details of the application logic. Finally, by implementing security at the middleware level, it is possible to improve the utilization of resources, such as bandwidth. For example, instead of sending public keys in separate messages, they can be transported together with the result of a lookup query.

8 Validation and Conclusion

This paper described `SaMi`, a remote method invocation system that implements security policies by means of a discretionary access control system. `SaMi` associates to every remote object an access control list, that describes, for each of its methods, the users that can invoke it. The basic security requirement guaranteed by `SaMi` is: *in order to invoke a remote method, the client needs the permission of the service provider that implements it.*

The mechanism used to accomplish the proposed policy is based on public key cryptography. In order to invoke a method, the client needs a certificate. The discovery agency that clients use to locate remote objects is the central authority responsible for issuing certificates. The adopted cryptographic protocol makes computationally unfeasible to forge certificates, and avoids replay and man-in-the-middle attacks. It is important to notice that, although this protocol authenticates clients to service providers, the opposite path is not guaranteed: clients do no receive any assurance about the identity of the remote object. However, due to the modular nature of the system, such extension is simple.

The proposed security model is flexible and general. An recurrent critic made on access control list based systems is the small number of different access privileges they provide. For example, in the UNIX operating system, the only operations allowed on files are reading, writing and executing. However, in `SaMi`, every remote method constitutes a separate entry in the access control list, and its access police can be configured independently of other methods.

An important point of the proposed model is the natural matching between a remote method invocation middleware and a distributed certification system. In both scenarios there is the concept of a central authority globally accessible. In the middleware's case, it is the name server, or discovery agency, wheres in the certification platform it is the certifying authority.

Other interesting features present in `SaMi` architecture are computational reflection and call backs. The first mechanism allows service providers to change their security parameters during runtime. The second is employed in the implementation of the protocol used by clients to update the keys that they have registered in the discovery agency. Finally, the implementation of `SaMi` is available at `http://compilers.cs.ucla.edu/fernando/projects/`

References

[1] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *14th Symposium on Operating Systems Principles (SOSP)*, pages 217–230. Software–Practice and Experience, 1993.

[2] Keith Brown. *The .NET Developer's Guide to Windows Security*. Addison Wesley, 2005.

[3] Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109 – 126, 2002.

[4] Frantisek Plasil and Michael Stal. An Architectural View of Distributed Objects and Components in CORBA, Java RMI and COM/DCOM. *Software - Concepts and Tools*, 19(1):14–28, 1998.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, October 1994.

[6] Ralph E. Johnson. Components, frameworks, patterns. In *SIGSOFT Symposium on Software Reusability*. ACM, 1997.

[7] J Kohl and C Neumann. Rfc 1510: The kerberos network authentication service v5, 1993.

[8] Casey Marshall and Raif S Naffah. *Programming with GNU Crypto*. Free Software Foundation, 2003.

[9] Piet Obermeyer and Jonathan Hawkins. Microsoft .net remoting: A technical overview. Technical Report 013, Microsoft Corporation, July 2001.

[10] OMG. CORBA IIOP 2.3.1 Specification. Technical Report 99-10-07, OMG, 1999.

[11] Nikos Parlavantzas, Geoff Coulson, and Gordon Blair. Applying component frameworks to develop flexible middleware. In *Workshop on Reflective Middleware*. ACM Press, 2000.

[12] Fernando M Q Pereira, Marco T O Valente, Roberto S Bigonha, and Mariza A S Bigonha. Arcademis: a framework for object oriented communication middleware development, 2005. Accepted for publication.

[13] Fernando M Q Pereira, Marco T O Valente, Wagner S Pires, Roberto S Bigonha, and Mariza A S Bigonha. Tactics for Remote Method Invocation. *J-JUCS*, 10(7):824–842, 2004.

[14] Doublas Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible and Maintainable ORB Middleware. *IEEE Communications Magazine – Special Issue on Design Patterns*, 37(4), 1999.

[15] Douglas Schmidt. *Acceptor-Connector: Design Patterns for Initializing Communication Services*, chapter 12, pages 191 – 206. Addison-Wesley, 1997.

[16] Douglas Schmidt and Greg Lavender. Active object – an object behavioral pattern for concurrent programming. In *Second Pattern Languages of Programs conference*, September 1995.

[17] Douglas C Schmidt and Frank Buschmann. Patterns, frameworks, and middleware:their synergistic relationships. In *25th International Conference on Software Engineering*. IEEE, 2003.

[18] Ashish Singhai, Aamod Sane, and Roy H. Campbell. Quarterware for middleware. In *18th International Conference on Distributed Computing Systems (ICDCS)*, pages 192 – 201. IEEE, 1998.

[19] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232. USENIX, 1996.