Securing Java RMI-based Distributed Applications

Ninghui Li
CERIAS and Department of Computer Sciences
Purdue University
250 N. University Street, West Lafayette, IN 47907
ninghui@cs.purdue.edu

John C. Mitchell
Department of Computer Science
Stanford University
Gates 4B, Stanford, CA 94305-9045
mitchell@cs.stanford.edu

Derrick Tong
Google Inc.
1600 Amphitheatre Parkway, Mountain View CA 94043

Abstract

Both Java RMI and Jini use a proxy-based architecture. In this architecture, a client interacts with a service through a proxy, which is code downloaded from a directory and installed on the client's machine. An attacker who controls the communication channels or the directory may compromise the confidentiality and integrity of the client and of the service. We present a security architecture that protects both clients and services in distributed proxy-based computing. In this architecture, the service registers a signed authentication proxy with the directory. The client, after downloading a signed authentication proxy from the directory, verifies the signature on the proxy, authenticates itself to the service through the proxy, and receives a dedicated session proxy for the service over a secure channel. We also describe a Java-based toolkit that implements the security architecture. This toolkit enables developers to add security to Java RMI-based applications with minimal implementation effort.

1. Introduction

Java RMI (Remote Method Invocation) [11, 8] is a key enabling technology for creating distributed Java-based applications. RMI is widely used to implement critical distributed enterprise information systems. It is a key technology under the popular Enterprise JavaBeans technology [12]; it is also used to create Jini-based applications [1]. RMI's purpose is to make objects in separate Java Virtual Machines (JVMs) look and act like local objects. We call the architecture used by Java RMI the *proxy-based architecture*. In traditional client-server systems, where a client transmits requests to and receives responses from a server, the client needs to know the location of the server and the correct protocol to communicate with the server. In the proxy-based architecture, the provider of a service generates a proxy for the service and registers the proxy with a directory, which may be running on a different Virtual Machine (VM). The proxy contains service-specific code that will run on the client VM and handles all interactions with the service. In Java RMI, the proxy is called a stub, which impersonates a remote Java object (the service object) in the client's JVM and forward calls to the remote JVM where the service object is located. More generally, the proxy may perform more functionalities than simply forwarding method calls. It is also possible that the proxy provides the service entirely by itself, without any remote communication. A client that wants to use the service looks up the proxy in the directory, downloads the proxy into the client's VM, and then communicates locally with the proxy as if it is the remote service. Proxies hide communication details, allowing the client to use a uniform high-level interface to services without regard to the service location, network communication protocol, or other service interaction details.

In addition to being used in RMI, the proxy-based architecture is also used by Jini [1]. Jini is a service-based distributed computing architecture. It abstracts both the devices and software under a service notion and supports dynamic community formulation and dissolution. As more and more critical systems are built using the proxy-based architecture, the security of these systems is a natural concern. The use of proxies introduces security vulnerabilities that need to be dealt with before people can rely on systems built using technologies such as Java RMI. As we discuss in Section 2, an attacker who controls the network communications between the client VM and the directory VM or between the service VM and the proxy VM can compromise the confidentiality and integrity of the client as well as those of the service, as can an attacker who controls the directory.

The use of proxies causes several challenges in providing security mechanisms for proxy-based systems. For example, the goal to provide mutual authentication between a client and a service is complicated by the presence of a proxy. The client and the service communicate through the proxy, which neither the client nor the service fully controls and trusts. The proxy is generated by the service; thus the client cannot trust the proxy more than it trusts the service, even after the proxy has been authenticated. On the other hand, the proxy is executed under the control of the client; thus, the service cannot fully trusts the proxy either. One approach that might seem plausible is to treat the proxy as part of an insecure communication channel connecting the client and the server, and to use standard authentication techniques to provide mutual authentication between them. The difficulty of this approach is that the client communicates only with the proxy, and is aware of only the proxy. In fact, in the extreme case, the proxy provides all services to the client and there may not even exist a remote service object. Some of these difficulties have been noted in the literature [2, 3]. The authors of [2] admitted the difficulty in supporting service authentication by saying "a more satisfying solution (to the problem of service authentication) is yet to be found." While the authors of [3] admitted the difficulty in supporting client authentication by saying "the design (in [3]) supports cleanly only server authentication".

In this paper we present a security architecture for securing distributed applications built using the proxy-based architecture. The security architecture overcomes the challenges described above and achieves several goals: mutual authentication between the client and the service (with choice of the authentication method), protected communication between the proxy and the service, and service access control. In order to reach these ends, the architecture uses a signed authentication proxy. When a client looks for the proxy of a service, it finds the signed authentication proxy. The client verifies that this proxy is signed by the intended service provider. The proxy and the service communicate through TLS/SSL [10] with server authentication, enabling the proxy to authenticate the service and ensuring that the communication between the proxy and the service is secure. The client authenticates itself to the service through a method call to the authentication proxy. After a client is authenticated, a client-specific session of the service that caches the client authentication data is created on the service VM and a proxy of the session is returned to the client. When the client uses the service through this session proxy, the authentication data can be reused. This avoids repeating the authentication effort on every method call to the service.

We also describe a toolkit that we have implemented to help application developers adopt the security architecture in Java RMI-based applications. The public available toolkit[1] uses several facets of Java 2 Security, including the Java Cryptography Extension (JCE), Java Secure Sockets Extension (JSSE), and Java Authentication and Authorization System (JAAS) [5]. The toolkit uses standard Java runtime and Java RMI features, without changing anything "under the hood". This makes it easy to deploy applications built using the toolkit. Applying the toolkit to secure a Java RMI-based applications requires few changes to the application code. On the service side, only the method call that exports a service needs to be changed to a call into the toolkit; the service interface and implementation remain unchanged. On the client side, only the code to look up the service proxy needs to be changed. It is straightforward to add security into an existing application by making the above modest changes to the application code.

The remainder of the paper is organized as follows. In Section 2, we analyze the threats posed to proxy-based distributed systems and state the goals of a security architecture. Section 3 presents an architecture which accomplishes these goals, and Section 4 describes a toolkit that implements this architecture. We discuss related work in Section 5 and conclude in Section 6.

2. Vulnerability Analysis and Security Goals

We now analyze the vulnerabilities of the proxy-based architecture and present the desirable security goals. For concreteness, we use Java RMI in the presentation; however, the same vulnerabilities also exist in other proxy-based distributed systems such as Jini-based systems. The sequence of interactions among the service, the client, and the proxy is shown in Figure 1 and described below. In these interactions, three virtual machines (VMs) are involved: the client VM, the service VM, and the directory VM.

1. The service makes a call to the RMI runtime to export the service. In this process a proxy for the service is created in the service VM.
2. The service makes a remote call to a directory (e.g., rmiregistry) to register its proxy. After this call, the directory VM has the proxy object.
3. The client looks up the service and downloads the proxy from the directory. After this step, the client VM has the proxy.
4. Each time the client needs to use the service, it calls the proxy locally as if it is the remote service.
5. The proxy communicates with the service to serve the client's request.

1 http://theory.stanford.edu/people/jcm/software/secureRMI.html

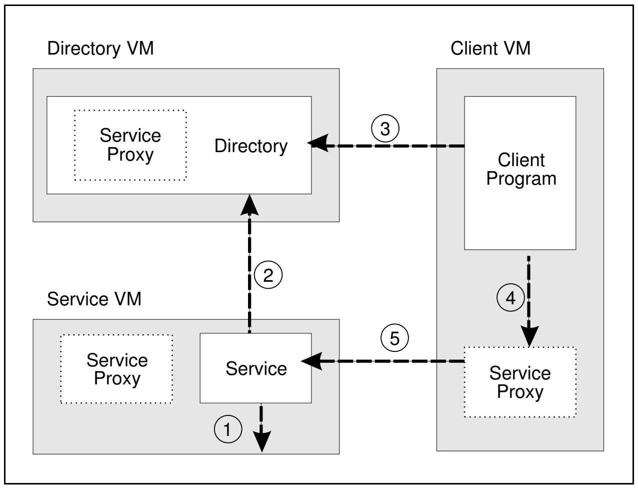

Figure 1. A normal client/service interaction

After a client has located and installed a proxy, the client may reuse the proxy to make further requests by repeating steps 4 and 5 for as long as the proxy is considered valid by the service.

2.1. Vulnerabilities

In our vulnerability analysis for proxy-based systems, we consider three parties: the user, the service provider, and the attacker. We assume that the user has full control over the client VM and that the service provider has full control over the service VM. An attacker, however, may control the network and the directory VM. In other words, we consider the directory as untrusted; a compromised directory should not compromise the security of the client or the service.

The risks to the client are:

Client confidentiality The client may transmit sensitive information to the proxy when trying to use the service. For example, if the service provides access to the user's investment accounts, the user wants to make sure that his instruction is sent only to the service provider, but not to anyone else.

Client integrity The user's VM and local environment may be damaged by the proxy.

Client confidentiality may be compromised by the attacker through the following two ways. One, the attacker can eavesdrop on the communications between the service and the proxy. Two, the attacker may get the client to install and run a bogus proxy, either by compromising the directory or by actively attacking the communication channel between the directory VM and the client VM or the channel between the directory VM and the service VM. In this attack, client integrity may also be compromised.

The risks to the service are:

Service confidentiality The service may send sensitive information to the client through the proxy.

Service integrity The service needs to perform certain functionalities on behalf of its client, and an attacker who successfully impersonates a client may result in damages.

Service confidentiality and integrity may be compromised by an attacker who controls the communication between the proxy and the service, by an attacker who can get the client to accept a bogus proxy, and by an attacker who can impersonate a client.

In this paper we do not consider risks related to availability, as a denial-of-service attack can always be carried out by an attacker who controls the communication channels and/or the directory.

2.2. Security Goals

The security architecture is designed to prevent the attacker from compromising client or service confidentiality and integrity as described above. Security is achieved through meeting the following goals:

1. **Service authentication** When the client communicates with the proxy of a service, it should have sufficient confidence that it is communicating with the service it intends to.
2. **Secure communication channel** Both the client and the service should have confidence that the communication between them is protected from other entities.
3. **Client authentication and authorization** Services should be able to authenticate the requesting clients and authorize them accordingly.

Goals 1 and 2 ensure that any data sent to the proxy are received only by the service; thereby achieving client confidentiality. Goals 2 and 3 achieve service confidentiality and integrity. While Goal 1 helps in achieving client integrity, it is not sufficient by itself. The proxy's access to the client's VM must be limited. Even after the client authenticates a service proxy, the user should not trust the proxy completely and should still restrict the proxy's access to local environment. In this paper we do not address this because it is best achieved with a virtual machine permission architecture as in Java 2 Security [4].

3. An Architecture for Securing Proxy-Based Applications

In this section, we present an architecture that achieves the three goals set in Section 2.2. We first discuss techniques that achieve each goal individually, and then present the complete architecture that combines these techniques. We discuss also alternative techniques and why we found them to be less than satisfactory. Although some elements in our design are specific to Java RMI-based systems, we believe that the main concepts apply to other proxy-based systems as well.

3.1. Using signed proxies to authenticate services

We now discuss how to achieve service authentication. Balfanz et al. [2] proposed to use TLS/SSL with server authentication. In order to ensure that the TLS/SSL code is implemented correctly, they disallow dynamic downloading of stub code. They observed that “this, however, disables a whole set of features for distributed Java programs, so that a more satisfying solution is yet to be found.”

We observe that, in the proxy-based architecture, the client is in direct contact only with the proxy and may be oblivious to the location or even the existence of the remote service (as the service may be provided by the proxy). Therefore, the client cannot authenticate the service directly. From the client’s point of view, the proxy is the only thing that represents the service; thus, the client should authenticate the proxy to be convinced that the proxy indeed represents the intended service. If the proxy needs to communicate remotely with the service, it should authenticate the service to be convinced that it is indeed communicating with the original service.

In summary, the service is authenticated in two steps: the service is authenticated by the proxy and the proxy is authenticated by the client. In our architecture, the proxy is digitally signed so that it can be authenticated by the client, and the proxy authenticates the service by using TLS/SSL with server authentication.

In this approach, it is ultimately up to the service provider to ensure that its proxy authenticates its service correctly. This is a consequence of the fact that the proxy is provided by the service. The implication is that the client has to trust the service provider to have correctly implemented the proxy. This trust is justifiable for the following two reasons. First, it is part of the service’s responsibility to implement its service correctly and free of security holes. As the proxy is part of the service, it is also part of the service’s responsibility to implement the proxy correctly. Second, the user is already trusting the service provider with any sensitive data she wishes to provide to the service. If the user trusts a service to handle the sensitive information properly, the user might as well trust that the proxy implemented by the same provider also handles this information properly.

We now discuss the details of implementing signed proxies in Java. The standard Java security mechanism provides support for signed JAR files. The Java Archive (JAR) file format enables one to bundle multiple files into a single archive file. Typically a JAR file will contain the class files and auxiliary resources. In our architecture, the code of the proxy is stored in a signed JAR file. When the signed JAR file is downloaded to the client VM, the client can verify that the signature on the jar file is valid and that the signing key represents the intended service. However, just verifying the signature on the code is not sufficient — both the proxy code and the proxy data must be authenticated. The proxy data may contain, for example, the service’s public key so that the proxy can use TLS/SSL to communicate with and authenticate the service.

In our architecture, we use what we call encapsulated objects to authenticate the data part of a proxy. This is used in addition to signed JAR files. A *encapsulated object* contains the object to be encapsulated together with a description object and a digital signature over the object and the description object. A description object is needed when a public key is used to sign multiple proxies, in which case a client needs to know not just who signed a proxy, but also what the proxy is supposed to do. A description object may be an URI string or any other object identifying the service associated with the proxy, known to both the client and the service. The client authenticates the proxy by verifying that the code is signed by a key adequately trusted, the proxy data is encapsulated by a key adequately trusted, and the description object is appropriate.

3.2. Using TLS/SSL to secure communication

We now discuss how to achieve Goal 2: securing the communication channel between the client and the service. This channel consists of two parts: the one between the client and the proxy, and the one between the proxy and the service. The former is local to the client’s VM, and is assumed to be protected from the attacker. The latter needs protection. In our architecture, this is achieved by having the service provider make sure that the proxy communicates with the service through TLS/SSL with server authentication. This ensures that the proxy only talks with the service and that the communication is protected against attackers. Observe that again it is up to the service provider to ensure that its proxy communicates over a secure channel. Since JDK 1.2, it is possible to use custom sockets in Java RMI; thus it is straightforward to use TLS/SSL with server authentication in RMI proxies using versions of JDK later than JDK 1.2.

3.3. Using authentication proxy to authenticate clients

Another security goal we would like to achieve is client authentication, i.e., to enable the service to authenticate the identity of the client. We first discuss three approaches that we considered and rejected.

The first such approach is to require each service to provide its own authentication interface. For example, this can be done by requiring the service object to implement an additional authentication interface. The downside of this approach is that the implementation of the service object needs to be changed. This increases the amount of work application developers need to do in order to use the security architecture. Also, as security code tends to be error-prone, rather than having each application implementing its own authentication mechanism, it is more desirable to have the authentication mechanism implemented and verified once and reused by multiple applications.

Another approach is to use TLS/SSL with client authentication. Indeed, this has been proposed in previous work on securing RMI-based systems [2]. One obvious difficulty of this approach is that it is the proxy who actually communicates with the service, yet the proxy does not have the client's private key, which is needed for TLS/SSL with client authentication. We reject two possible solutions to address this difficulty. The first one has the client take over the proxy's communications with the service, as done in [2]. This typically requires changing the Java RMI runtime, because the proxy might open a socket even before the client makes the first call to it. Changing the Java RMI runtime makes deployment much more difficult and increases the risk of non-interoperability. Furthermore, as proxies are meant to hide communication details from the client, requiring the client to take over the communications between the proxy and the service is against the spirit of the proxy-based architecture. The second inappropriate solution is to give the client's private key to the proxy. This is a serious violation of security principles. Even though the client has verified that the proxy is a valid representation of the service, it can only trust the proxy as much as it trusts the service itself. As the client would never authenticate to a service by giving its private key to the service, giving the private key to the proxy is unacceptable. Even though the proxy is executed on the client machine, it is typically very difficult (if not impossible) to be sure that the proxy will not leak the private key through open or covert channels [6].

The third approach that we rejected is to have the service authenticate the proxy (through TLS/SSL with client authentication) and the proxy authenticate the client for the service. This is problematic because the proxy is executed under the client's control and may be altered by the client or an intermediate attacker; thus the service cannot trust the proxy to authenticate the client.

In our security architecture, we use an authenticator that provides a generic interface for client authentication. Rather than registering a proxy for a service with the directory, the service first registers a proxy for the authenticator, called an *authentication proxy.* When the client retrieves the authentication proxy, it authenticates to this proxy through a method call that is forwarded to the service VM. If the client is authenticated and is allowed to use the service, then an actual service proxy is returned to the client. This way, client authentication is independent of the communication protocol and does not require the client to expose its private key.

3.4. Using dedicated session proxy to efficiently control client access

Finally, we would like to enable the service to perform access control based on the results of client authentication. Many clients may be allowed to use a service; however, different clients could have different permissions. For example, some clients may not be allowed to call certain methods of a service. In order to authorize different clients differently, the service needs to know which client called it. One way to do such finer-grained access control is to have authentication information passed as an argument in every method call. This design requires changing the signatures of all the remote methods to accommodate security, which is cumbersome. Another drawback is that authentication would then be repeated for each method call. This could be a serious performance problem, as the authentication process could involve such time-consuming computations such as verifying multiple public key signatures.

In order to do client-specific access control without changing Java runtime, we use *dedicated session proxies*. When a client is authenticated, a dedicated session is created and a proxy for it is returned to the client. The dedicated session maintains the information about the specific client that has been authenticated. This dedicated session proxy can be compared to a "ticket" in single sign-on architectures such as Kerberos [9]. Possession of a dedicated session proxy provides the client access to the service for as long as the proxy is considered valid by the service (i.e. the lifetime of a ticket).

3.5. A complete interaction sequence

Our architecture for securing proxy-based distributed systems combines the four techniques discussed above: signed proxy, secure communication, authentication proxy, and dedicated session proxy. We now describe a sequence of interactions when using the architecture with public key authentication for clients. These sequences are shown in Figure 2.

1. The service creates a proxy to an authenticator; we call this proxy the authentication proxy. The service then creates a signed authentication proxy, which is a encapsulated object that contains this authentication proxy and a description and is digitally signed.

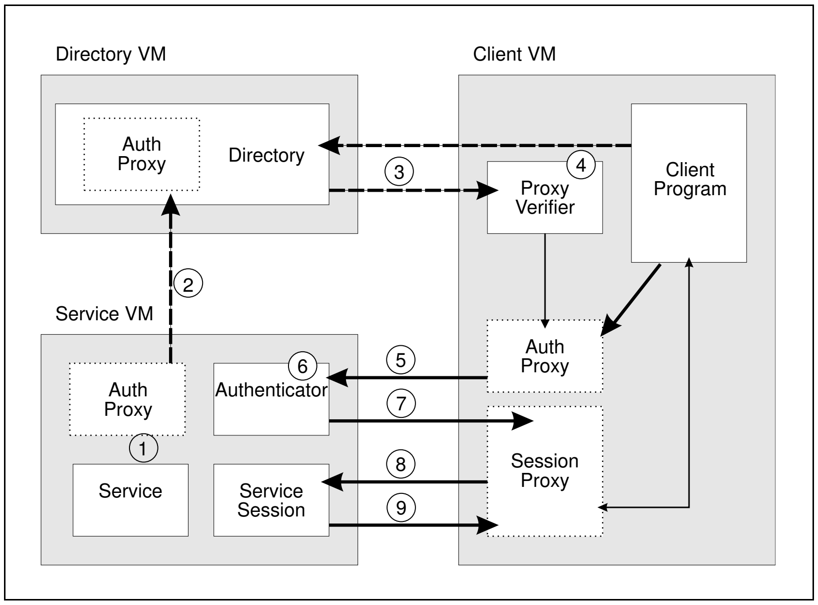

Figure 2. A secure client/service interaction

2. The service registers the signed authentication proxy with a directory.
3. The client looks up the service and downloads the signed authentication proxy from the directory.
4. The client verifies that the signature on the signed authentication proxy is correct and that the description is as expected.
5. The client authenticates itself by presenting its public credentials (i.e., public key certificates) to the authentication proxy, which communicates the request to the service over a secure channel.
6. The authenticator analyzes the request and decides whether to grant access to the client.
7. If access is granted, the service creates a service session that is dedicated to this client. It then returns a proxy to the service session, called the dedicated session proxy. This proxy is encrypted with the client's public key. The client, after receiving the encrypted proxy, decrypts it and gets the dedicated session proxy.
8. The client then makes its service requests through the session proxy, which communicates the requests to the service VM via a dedicated secure channel.
9. By receiving the request through this dedicated channel, the service can associate it with the appropriate client. The service processes the request accordingly and returns the result.

The client may then reuse the proxy to make further requests by repeating steps 8 and 9 for as long as the proxy is considered valid by the service.

When the functionality of a service is implemented by a proxy without interaction to the service, then there is no need to do client authentication and access control, as every client can download the proxy and use the service. In this case, a signed proxy can be used to enable the client to authenticate the proxy.

4. A Secure RMI Toolkit

We have implemented a toolkit that enables developers of RMI-based applications to easily adopt the security architecture described in Section 3. Our goals in designing and implementing this toolkit are to make security as easy to add and as flexible to use as possible. As a base line requirement, the toolkit should not require modifications to the standard Java runtime or the standard JDK. We also want to support flexible authentication and authorization. Application developers should be able to implement new authentication and authorization methods and use them with this toolkit. In addition to meeting the above goals, the toolkit also provides support for the Java Authentication and Authorization System (JAAS) to allow existing JAAS-enabled code to leverage Java's authorization framework in a distributed environment.

The toolkit consists of a set of classes implementing the security architecture. To use this toolkit, it should be statically loaded both at the service side and at the client side. This toolkit is assumed to be trusted, just like other statically loaded libraries.

4.1. Usage of the Toolkit

On the service side, the toolkit handles the creation of authenticators, authentication proxies, signed authentication proxies, dedicated service sessions, and dedicated session proxies. To use the toolkit on the service side, the service program first creates an instance of the server toolkit, providing it with its trust store (trusted root CA's), its public key (for SSL server authentication), and its private key (to sign the proxies). After that the service program, instead of calling the standard Java RMI runtime to export the service, calls the toolkit to do so. The toolkit calls standard Java RMI runtime to export an authenticator for the service, and returns a proxy for the authenticator to the service program. The service program receives an authentication proxy from the toolkit and then registers it with a directory as usual. The toolkit handles all remaining authentication details. From within the service code itself, authorization decisions may be made automatically by the service's JAAS policies, as the toolkit marks each incoming service request with the client's identity (`javax.security.auth.Subject`). Alternatively, the service code may query the toolkit to determine the identity of the calling client and base its authorization decisions appropriately.

On the client side, the toolkit handles the verification of the authentication proxy, the authentication to the service, and the retrieval of the dedicated session proxy. The client program creates an instance of the client toolkit, providing it with its public key (for client authentication) and its private key (to decrypt service proxies). Next, the client uses the toolkit to look up the desired service. Along with the location of the directory and the name of the service, the client provides the expected description of the service, as well as the policies specifying the acceptable credentials of the service provider. The toolkit then handles the retrieval and verification of the signed authentication proxy, as well as the client authentication process to obtain the desired service proxy (i.e., the dedicated session proxy).

4.2. Implementation of the Toolkit

In this section, we describe the implementation of some of the major components of the toolkit. In particular, we describe some interesting techniques that we use to achieve the toolkit's flexibility and functionality.

Signed authentication proxy On the service side, the toolkit is called to export a given service. To do this, the toolkit creates an `Authenticator`, which handles the process of client authentication. The `Authenticator` object resides on the service VM and keeps a reference to the service it protects. The toolkit exports the `Authenticator`, during which process a proxy to the `Authenticator`, which we'll call the `AuthenticatorProxy`, is created. Next, this `AuthenticatorProxy` must be signed. To do so, it is first wrapped in a `java.rmi.MarshalledObject`, which serializes the proxy in such a way that the appropriate code can be located and downloaded by the RMI runtime. The `MarshalledObject` is then wrapped by a `DescribedObject`, which simply attaches a description to the object it wraps. Finally, this `DescribedObject` is wrapped in a `java.security.SignedObject`, which computes a signature over the object it wraps.

We now have a signed authentication proxy object that functions as the first object received by the client in our security architecture. However, the toolkit would do poorly to return this object to the service program as it is, for two reasons. First, when exporting a service through the toolkit, the programmer will expect an object that implements the remote interfaces that the service itself implements, as is the case in using standard Java runtime. If, however, the toolkit returns a `SignedObject` instead, this interface contract would be violated, and much of both the client and service implementations would need to be modified. Second, one would have trouble using this object in Jini. Registering a `SignedObject` with the Jini directory would not be in accordance with Jini's directory lookup process, which relies upon the interface of the proxy. The Jini lookup algorithm would no longer be able to locate the service because the proxy representing the service (e.g. the `SignedObject`) does not correctly implement the service interface.

We now explain how we resolve this problem. The security architecture requires the `SignedObject` that contains the authentication proxy, while Jini requires an object implementing the original service interface. Thus, the solution lies in returning an object that meets both of these requirements—a `SignedObject` that implements the original service interface. The difficulty lies in the fact that the toolkit does not know the service interface a priori—how, then, can the toolkit provide a general solution that works with arbitrary service interfaces? Our solution lies in a technique called dynamic class generation. JDK 1.3 introduced the `java.lang.reflect.Proxy` class, which dynamically generates classes implementing an arbitrary list of interfaces. We use the `Proxy` class to create what we call a typed wrapper, which is essentially an object that wraps another object while implementing an arbitrary set of interfaces. The typed wrapper returned by the toolkit's export method implements all of the remote interfaces of the given service, along with the special TypedWrapper interface, which contains a single method `$getWrappedObject()`. The behavior of the resulting object is to return the signed authentication proxy in response to a call to `$getWrappedObject()`, and throw an `UnsupportedMethodException` when any other method is called. (Figure 3) We thus have an object that implements the interfaces expected by existing service code, conforms to the Jini lookup process, and fulfills the requirements of our security architecture.

Authentication Modules Different applications may need to use different authentication methods. In the toolkit,

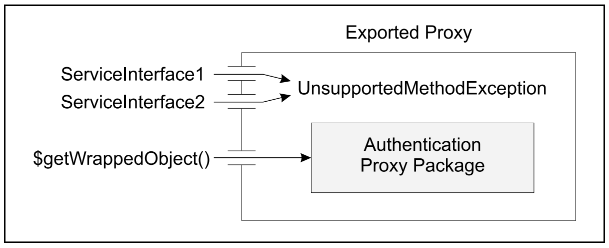

Figure 3. A typed wrapper for a service implementing two interfaces

we clearly could not implement all possible authentication methods. Instead, we implement the toolkit so that the client authentication process is performed using a set of pluggable, stackable *authentication modules*. These modules are loaded from a configuration file that the client can modify by specifying its own verification modules. Application developers can write their own authentication modules and load them through the configuration file.

Each authentication module implements the `AuthenticationModule` interface, which consists of three methods: `prepareForAuthentication`, `unpack`, and `authenticate`. The first two methods are called on the client side, and the last method is called on the service side. The authentication task begins on the client side, where the `prepareForAuthentication` method of each module is called to prepare any data that will be needed by the same module on the service side; such data may include, e.g., the client's public credentials. Once all such data has been collected, it is sent as an argument to a method call to the `AuthenticatorProxy`. When the data is received by the Authenticator, it is processed by the `authenticate` method of the authentication modules. Each module analyzes the data sent by the `prepareForAuthentication` method of the same model and decides whether to grant access. If the module decides to grant access, it packages the service proxy in some way and passes it to the next module. Finally, the resulting object is returned to the client, where the `unpack` method of the authentication modules are then called one by one to recover the service proxy.

The architecture is "stackable", and the order of the modules on the service side is the reverse order of the modules on the client side. This is best shown by an example. The toolkit currently implements three modules: a trust management module, a dedicated session module, and a cryptography module.

Dedicated session proxy The toolkit needs to create a dedicated session proxy after the client is authenticated. The purpose of this proxy is two-fold: (1) single sign-on (described in Section 3.4) and (2) service-level authorization. We now detail how the dedicated session enables both JAAS-based and non-JAAS-based authorization. We need to give the methods in the service object access to the client identity, without changing the signatures of these methods. Without using a service session dedicated to a client, this is very difficult to achieve without changing RMI runtime. As RMI is designed to abstract proxy communication details from both the client and the service, there is no simple way to tell where a remote call originates. In the `java.rmi.server.RemoteServer` class, the `getClientHost()` method comes close to providing the necessary functionality. However, this is inadequate, since we cannot assume that each remote host corresponds to a single client. Furthermore, we cannot rely on any low-level assumptions about RMI proxy connections and threads, since no such association is specified in the RMI standard and thus such implementation specifics are subject to change.

The creation of a service session dedicated to a single client makes it possible to maintain the client identity. This identity then enable the use of access control based on JAAS policies. It can also be retrieved by the service code for access control based on other policies implemented in the service code. To add the client identity information, we introduce a level of indirection, once again using the `java.lang.reflect.Proxy` class. Instead of exporting the service itself to create the dedicated session proxy, we create a wrapper to the service and export that wrapper object instead. This wrapper object implements all of the service's `Remote` interfaces, and also stores a `javax.security.auth.Subject` object representing the client associated with this dedicated session. The behavior of the dedicated session is described in Figure 4 and explained below.

1. The dedicated service session receives a remote call from the client, forwarded by the dedicated session proxy.
2. The session stores the `Subject` of the client in a `ThreadLocal` variable, making it accessible to any code called thereafter.
3. The session then calls `Subject.doAs(...)` to forward the method call to the service.
4. The service processes the method call, making authorization decisions using the `Subject` stored in the `ThreadLocal` variable, or using built-in JAAS policies made possible from the `Subject.doAs(...)` call.
5. After the service returns (or throws an exception), the session removes the client's `Subject` from the `ThreadLocal` variable.

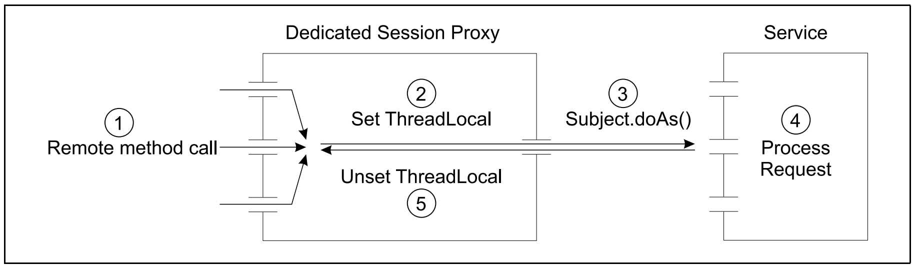

Figure 4. The Dedicated Session

Wrapping each incoming service method call with a call to `Subject.doAs(...)` allows the service application to use JAAS-based authorization. Using the toolkit, services that use JAAS stack inspection for authorization can now do this for remote method calls without modifying any existing code. Storing the client's `Subject` in a `ThreadLocal` variable allows the toolkit user to perform non-JAAS authorization. The toolkit exports a static method that returns the `Subject` stored by the service session making the current call (or `null` if the current call is not a remote call). Thus, authorization decisions in the service code can be made by calling this method and examining the credentials of the calling client.

4.3. An Example of Using the Toolkit

To verify the practical usability of the architecture and the toolkit, we applied the toolkit to a user-to-user instant messaging system we wrote before starting the current project. We found that only a small amount of code change is required to apply the toolkit to the application, thereby validating our design of the architecture and toolkit. Applying the toolkit involved the following two steps. The first step is to add method calls to the toolkit to instantiate toolkit objects. These objects are initialized with keystores containing the credentials of trusted authorities and other entities. The second step is to replace the usual export and lookup of the services by the secure versions of them provided by the toolkit. Figure 5 shows the code changes before and after using the toolkit.

5. Related Work

Balfanz et al. [2] described the design and implementation of a security infrastructure for a distributed Java application. Their focus was on exploring a variation of SPKI/SDSI as an access control language in the application. Issues such as verifying the authenticity of the service proxy were not considered. They used SSL/TLS for client and service authentication and discovered some difficulties of this approach such as those discussed in Section 3. Eronen and Nikander's [3] work on Jini security is closer in spirit to the present paper. They considered verification of proxies. However, instead of verifying a signature on the serialized value of a proxy, they verify a signature on a message digest of the proxy object computed by the proxy object itself. We are able to directly verify a serialized object because of the `java.lang.reflect.Proxy` class in JDK 1.3, which might not have existed at that time of the previous work. Eronen and Nikander also had difficulty using TLS/SSL for client authentication and concluded that "the design supports cleanly only server authentication". The two papers above addressed the issue of trust-management policy languages for authorization. We view trust management, and the decision to trust a key or not, as an orthogonal issue to the security architecture for proxy-based distributed systems and beyond the scope of this paper.

The approach of using an authenticator for client authentication and a dedicated proxy for client authorization was discovered independently by Marques in [7]. (Our architecture was first presented in January 2002 at DARPA Dynamic Coalitions PI meeting.) The work by Marques is limited in its scope as it does not deal with proxy authentication and supports only password for client authentication.

The Davis project [13] was an effort by the Jini technology project team at Sun Microsystems to provide the new programming models and infrastructure needed for a Jini security architecture. The Davis project specification have been incorporated into the v2.0 release of the Jini Technology Starter Kit. The approach taken in the Davis project is to revamp RMI to provide security. The revised RMI programming model has not been incorporated into standard Java yet. The approach to revamp RMI to provide security as well as other functionalities was also taken in two rejected Java Specification Requests (JSR's), namely JSR 76 and 78. We believe that an architecture that adds security without changing the underlying RMI mechanism has its virtue. As there already exist many business applications that use RMI, it is desirable to provide security to these applications without rewriting them using the new RMI programming model. Furthermore, we believe that our security architecture can be applied to other, similar, proxy-based systems, such as those based on Jini.

```
[SERVICE SIDE]
Before:
 Directory exported = (Directory) UnicastRemoteObject.exportObject(directory);
 Naming.bind("//host/directory", exported);
After:
 ....  // create the server toolkit
 Directory exported = (Directory)
   serverToolkit.exportSignedAuthenticatingObject(directory,"Description of directory");
 Naming.bind("//host/directory", exported);

[CLIENT SIDE]
Before:
 Directory dir = (Directory) Naming.lookup("//host/directory");
After:
 ...   // create the client toolkit
 Directory encapsulated = (Directory) Naming.lookup("//host/directory");
 // Verifies the signature and description of the encapsulated proxy is correct
 directoryCertificate = ... // the certificate of the desired signer of the directory
 Directory dir = (Directory)
   clientToolkit.unwrap(encapsulated, directoryCertificate, "Description of directory");
```

Figure 5. Example of code changes before and after using the toolkit

6. Conclusions

More and more critical systems are built using the proxy-based architecture, which is used in Java RMI and Jini. In proxy-based distributed applications, client and server confidentiality and integrity may be compromised by adversaries controlling the communication channels. We presented a security architecture to counter these threats by addressing the unique challenges presented in proxy-based distributed systems. The architecture provides mutual authentication between the client and the service, secure communication channels, and efficient client access control. We also described the design and techniques that we have used to implement the security architecture in the form of a Java-based toolkit, which allows security to be added to Java RMI-based applications with minimal implementation effort.

Acknowledgement This work is supported by DARPA through SPAWAR contracts N66001-00-C-8015. It is also supported by DOD MURI “Semantics Consistency in Information Exchange” as ONR Grant N00014-97-1-0505 and by DOD University Research Initiative (URI) program administered by the Office of Naval Research under Grant N00014-01-1-0795. We thank the anonymous reviewers for their helpful comments.

References

[1] Ken Arnold, editor. *The Jini(TM) Specifications*. Addison-Wesley, 2000.

[2] Dirk Balfanz, Drew Dean, and Mike Spreitzer. A security infrastructure for distributed Java applications. In *Proceedings of 2000 IEEE Symposium on Security and Privacy*, pages 15–26, May 2000.

[3] Pasi Eronen and Pekka Nikander. Decentralized Jini security. In *Proceedings of the Network and Distributed System Security Symposium*, February 2001.

[4] Li Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, June 1999.

[5] Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers. User authentication and authorization in the Java platform. In *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC’99)*, pages 285–290, December 1999.

[6] Butler W. Lampson. A note on the confinement problem. *Communication of the ACM*, 16(10):613–615, October 1973.

[7] Paulo Marques. Building secure java rmi servers. *Dr. Dobb’s Journal*, November 2002.

[8] Rickard Öberg. *Mastering RMI*. John Wiley & Sons, Inc., 2001.

[9] B. Clifford Neuman and Theodore Ts’o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, pages 33–38, September 1994.

[10] Eric Rescorla. *SSL, TLS: Designing, and Building Secure Systems*. Addison-Wesley, 2001.

[11] Sun. Core Java: Java Remote Method Invocation (Java RMI). `http://java.sun.com/products/jdk/rmi/`.

[12] Sun. J2EE: Enterprise JavaBeans Technology. `http://java.sun.com/products/ejb/`.

[13] The Davis Project Team. The Davis project home page. http://davis.jini.org/.