Deployment-Based Security for Grid Applications

Isabelle Attali, Denis Caromel, and Arnaud Contes

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis,
BP 93, 06902 Sophia Antipolis Cedex - France
First.Last@inria.fr

Abstract. Increasing complexity of distributed applications and commodity of resources through grids are making harder the task of deploying those applications. There is a clear need for a versatile deployment of distributed applications. In the same time, a security architecture must be able to cope with large variations in application deployment: from intra-domain to multiple domains, going over private, to virtually-private, to public networks. As a consequence, the security should not be tied up in the application code, but rather easily configurable in a flexible and abstract manner. To cope with those issues, we propose a high-level and declarative security framework for object-oriented Grid applications. This article presents the transparent deployment-based security we have developed. In a rather abstract manner, it allows to set security policies on various security entities (domain, runtime, nodes, objects) in a way that is compatible with security needs according to a given deployment.

1 Introduction

This paper aims at introducing our security framework designed for distributed applications and optimized for grid infrastructure. We focus here on the transparent and deployment-based security part. It allows the secure deployment and the secure execution of security unaware distributed applications. Writing security policies for an application deployed within at least two different administrative domains is a challenge and leads to ad-hoc security solutions. Grid computation makes this challenge harder to overcome. The dynamic nature of grid resources enforces the use of a security framework that can be easily adapted to dynamically acquired resources. From one execution to the other, acquired resources could change due to concurrent access to the grid by external applications or node failures. Regarding this aspect, security features should be easily configurable and adaptable. There is also a strong need of security from users, but writing security-enabled program is often difficult and painful. Rather than letting programmers write security-related code and handle security concepts, we advocate that a middleware should provide an easy way to use security features. Our approach goes further and introduces a transparent security model for distributed applications. Our security framework focuses on authentication of users and processes; it supports user-to-process, process-to-user, process-to-process authentication. We provide authentication solutions that allow users, user's processes, and the resources used by those processes to verify each other's identity and to communicate securely. We assume that processes are running on a trusted environment, our framework is not intended to protect objects from malicious runtimes but applications from each other and network attacks.

The security model discussed in this paper is implemented on *ProActive* [1, 2]. Section 2 introduces *ProActive* and its descriptor-based deployment model. Section 3 introduces the security model associated to the deployment model. Section 4 presents the declarative language we created to store and express security rules. Then, section 5 gives some implementation details and section 6 presents some benchmarks. Section 7 compares with related work. Finally, section 8 concludes and presents future extensions.

2 The ProActive middleware

ProActive is a LGPL Java library for concurrent, distributed and mobile computing. With a reduced set of simple primitives, *ProActive* provides a comprehensive API allowing to simplify the programming of applications that are distributed on Local Area Network, on clusters or on grids.

2.1 Base model

A distributed application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Given a standard Java object, *ProActive* allows to transparently add behaviours to this java object. These transparent behaviours are, for example, location transparency, activity transparency or mobility. Each active object has one distinguished element, the *root*, which is the only entry point to the active object. It also contains a connected graph of standard java objects called *passive objects*. References are only possible onto root object and not onto passive objects. There is no sharing of passive objects. Each active object has its own thread of control.

2.2 Descriptor-based Mapping and Deployment

Another extra service provided by *ProActive* is the capability to *remotely create remotely accessible objects*. For that reason, there is a need to identify *Runtimes*, and to provide them some services. A Runtime is a remotely accessible java object which offers a set of services needed by *ProActive* to access remote Java Virtual Machine. Offered services are the creation of local nodes, the creation of another VM (local or remote) and the creation of a local active object within an existing local node. At any time, a runtime hosts one or several nodes. A *Node* is an object defined in *ProActive* which gathers several active objects in a logical entity. Remote objects are identified by an URL, for example `rmi://lo.inria.fr/Node1` identifies a node. However, the first step towards seamless deployment is to abstract away from hardware and software details as node or runtime URLs. To abstract away the underlying execution plate-form, and to allow a *source-independent deployment*, *ProActive* provides the following elements:

- *Virtual Nodes*: an abstraction of the distributed resources,
- *XML Descriptors*: a way to define the logical entities the application needs to run, the computing resources available and the mapping of those logical entities onto those hardware resources (i.e. real machines, using actual creation, registry, and lookup protocols).

Virtual Nodes (VNs) are identified as a name (a simple string) in the program source. They are defined and configured in the XML descriptor file. After activation, a VN is mapped to one or to a set of actual *ProActive* Nodes. Of course, distributed entities (active objects) are created on Nodes, not on Virtual Nodes. From application point of view, there is no notion of local or remote objects or remote runtimes, nor how the underlying architecture has been set up. As a consequence, applications cannot know if some security risks exist when an interaction is performed. The rest of the article presents our security model and how it ensures a secure seamless deployment and a secure application execution.

3 Generic Security Infrastructure Model

ProActive is used to deploy distributed applications as if they were deployed onto a big virtual computer. First, the underlying deployment mechanism starts or acquires remote or local execution places (runtimes or nodes). Then, once the deployment is over, the real program can be started. All these steps involve many different security features such as remote creation of node, of activities, method calls, etc. The first item, prior to seeing how the security infrastructure works, we have to solve is how to identify participating entities.

3.1 Security Entity Model

A Security Entity is an object which contains a wrapped object onto which it enforces a security policy (see figure 1). The wrapped object is seen as a black box. The security entity does not interact with its wrapped object internal code to handle security features. This allows to secure all kind of java objects even if the object has been loaded using Java Native Interface (JNI). The security entity is able to intercept incoming and outgoing calls thanks to the use of the proxy pattern. For each intercepted call, it performs needed security checks onto that call. Each Security Entity is uniquely identified by an EntityID which is, actually, a PKI certificate. Access control, communication privacy and integrity and specific *ProActive* features like migration or group communication are defined in Access Control Lists (ACLs) using our security language (see section 4). It is worth mentioning that, as no assumption is done onto the wrapped object, it can also be a security entity. This is an important point that allows an hierarchical organization of security entities.

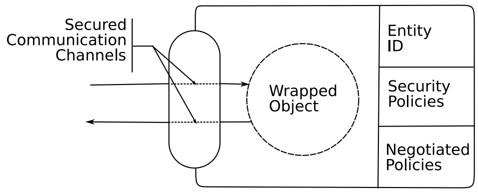

Fig. 1. A Security Entity

3.2 Hierarchical Security Entities

A *ProActive* application is composed of several objects: runtimes, nodes and active objects. These objects are remotely accessible and must be secured. The *ProActive* infrastructure implies an ordered structure (see figure 2): runtimes contain nodes which contain active objects. This hierarchy is used to structure and to define our security hierarchy. Each part of this structure belongs to a specific security level. When a Security Entity is involved in an interaction with

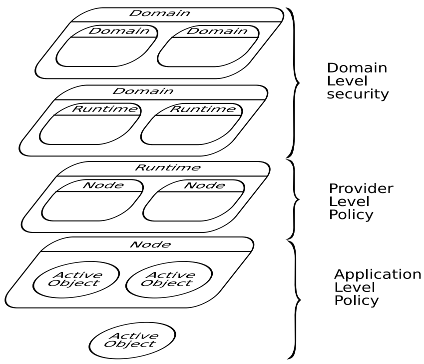

Fig. 2. Hierarchical Security Entities

an other, prior to performing the interaction, the Security Entity checks if this interaction is authorized by checking its owns security policy and by retrieving security rules from its wrapping security entities.

3.3 Domain Level Security

Domains are a standard way to structure (virtual) organizations involved in a Grid infrastructure. A Security Domain represents a distinct scope, within which common security needs are exhibited and common security rules observed. As a Security Domain is a Security Entity, it is possible to define fine grain and declarative security policies at the level of *Domains*. It allows organizations (companies, research labs, ...) to specify general security policies onto their resources. Domains are intended to be set by grid administrators.

3.4 Provider Level Security

Runtimes represent available Grid resources. They are started by grid users who act here as resource providers. A resource *Provider* is an individual, a research institute or an organization offering some resources under a certain security policy to a restricted set of users. A resource provider can set up a runtime with its own policy where clients will be able to perform computation. *ProActive* runtimes have been extended in order to become a Security Entities.

3.5 Application Level Security

Virtual Nodes are application abstractions, and nodes are only a run-time entity resulting from the deployment. A first decisive feature allows to define application-level security on those application-level abstractions. Application Security policies can contain references on virtual nodes used within the application source code. One can express which security attributes a communication between two Virtual Nodes requires. At execution, that security will be imposed on the Nodes resulting from the mapping of Virtual Nodes to runtimes.
Security policies are stored within an XML file, like the deployment descriptor. When a user want to start an application, he writes the deployment descriptor according to the deployment scenario he needs. He writes the security policies file and references this security policies file in the descriptor file.

4 Declarative Grid Security Language

The general syntax to provide security rules is the following[1]:

```
Entities -> Entities: Interaction # [SecurityAttributes]
```

An *entity* is a security entity onto which one want to define a policy rule. As an interaction always involves two entities, the first *Entities* part describes the entity which starts the interaction, the second one, the entity which receives the interaction. In order to express fine security rules, the *entities* part can be a set of entities. So, it is possible to specify that : (1) if the active object which starts the interaction belongs to Virtual Node A and if this Virtual Node is located within Domain D; (2) and the targeted active object belongs to Virtual Node B, then the communication must ciphered.
Interaction is a list of actions (communication, migration, creation of runtimes, nodes, active objects). Finally, security attributes specify how, if authorized, those interactions have to be achieved in terms of authentication, integrity and confidentiality. Each attribute can be required, optional or denied. This helps the

[1] for simplicity reasons, XML syntax has been removed

security mechanism to compute a result security policy according to the security attributes computation algorithm (see section 5).
Our language also accepts joker rules. They are important to cover all cases, to specify default behaviors, and to provide a conservative security strategy for un-forecasted deployments.
In order to provide a flavor of the system, we consider the following example.

```
Domain[inria.fr] -> Domain[ll.cnrs.fr] : Q,P # [+A,+I,?C]
```

The rule specifies that between the domain *inria.fr* (identified by a specific certificate) and the parallel machine *ll.cnrs.fr*, all communications (reQuests, and rePlies) are authorized, they are done with *authentication* and *integrity*, *confidentiality* being accepted but not required.

5 Adaptative Security Policies

As we are in a distributed world, without a global administrator to handle all security policies, a given interaction could involve many security policies. The following protocol is used by security entities to compute a final security rule for a given interaction using all matching security policies. Indeed, the final policies being used are both dynamically computed, and possibly change during computation time.

1. The object acting as a client performs a method call onto the object acting as server.
   (a) The security entity intercepts the method call. The security mechanism contacts the callee security entity and requests callee entity location informations (callee security entity + encapsulating entities).
   (b) The client security entity collects all location informations. Once it has all location informations about caller and callee, it can retrieve all security rules matching to the interaction between callee and caller on the caller side.
   (c) The caller's security entity computes all matching rules to obtain a result rule and sends it to the callee security entity.
2. Callee security entity receives the requested policy rule by the caller security entity. The callee security entity :
   (a) retrieves all rules that match the requested interaction, computes them to find the result rule.
   (b) compares the rule requested by caller security entity with the locally computed rule. If they do not match, the interaction stops and the callee's security entity returns an exception to caller's security entity. Otherwise, an object matching that session is created and will be used to perform all security related actions needed for this specific interaction. If needed, the caller generates a session key.
   (c) gets back to the callee security informations (Session ID, cyphered session key, computed rule).

3. The caller security entity checks that the returned policy rule matches its local policy, then creates a local session object. The method call is given to the Session object. The Session object performs security actions requested by the exchanged policy rule and gives the secured method call to the underlying transport layer.

6 Benchmarks

To perform benchmarks, we choose the *Jacobi iterations*, an algorithm to solve a linear matrix equation. It performs local computations and communications to exchange data. Benchmarks are executed on 4 Pentium IV@3.2Ghz, 1Go(DDR), 512 Kb L2 cache, Linux (2.4.22) computers interconnected by a 100Mb/s network. Java VM is Sun JVM 1.4.2. These computers are simultaneously used by their users. When activated, security policy requires that all interactions (communications and also deployment) are ciphered and authenticated. The left figure represents the average duration, in milliseconds, of a Jacobi iteration depending on the data. The right figure presents the duration of matrix initialization (data transfer) at the computation initialization step depending on the data. Benchmarks show that as soon as the size of computed data begins to grow the ratio of the overhead induced by the security strongly decreases, from 1.8 to 1.15.

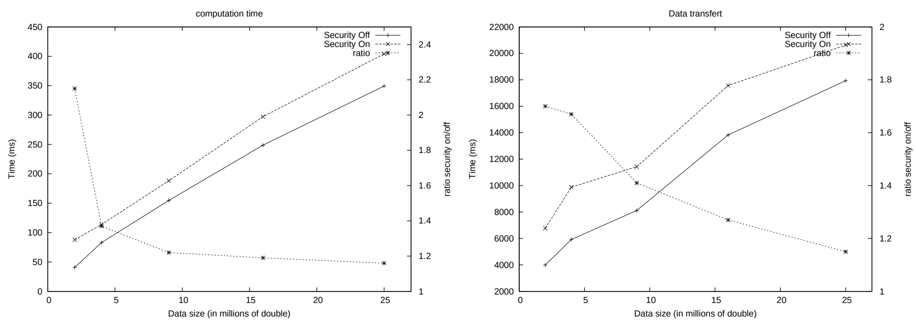

Fig. 3. Security Overhead

7 Related work

The .NET [3] framework provides security features allowing protection of a host against malicious code. Security system is based on user- and code-identity using public/private keys, hash functions, digital signatures and security policies. There are four policy levels Enterprise, Machine, User and Application Domain. *ProActive* security levels are not restricted. Note that *ProActive* does not use code identity. Legion [4] encapsulates functionalities as objects and provide mechanisms for their location, migration, etc. Legion objects interact via remote method invocations and the main security objectives are authenticating the communicating parties, protecting traffic, enforcing access control, delegating rights and enforcing site-specific security concerns. Unlike *ProActive*, Legion does not support a hierarchical management of security policies. The Globus system relies on the Globus Security Infrastructure (GSI) [5] that supports, integrates and unifies popular security models. It supports an inter-domain and intra-domain security interoperability. The notion of virtual organization is defined as a set of individuals and/or institutions sharing resources and services under a set of rules and policies governing the extent and conditions for that sharing. *ProActive* is able to interact with GSI, it also goes further by proposing a dynamic and adaptive security which takes account computation mobility.

8 Conclusion

We have proposed a decentralized, declarative security mechanism for distributed systems that features interoperability with local policies, dynamically negotiated security policies, and multi-users systems. The use of a logical representation of the application (Virtual Nodes) allows to have a security policy adaptable to the deployment, a crucial feature for Grid applications. The security mechanism allows to express all security-related configurations within domain policy files, and application deployment descriptor, outside the source code of the application.
As a very short term perspective, the implementation of group communication security is in the process of being finalized; a single session key being used for all group members. This work is an attempt to contribute to the construction of flexible solutions that are very much needed for Grid deployment.

References

1. Caromel, D., Klauser, W., Vayssière, J.: Towards Seamless Computing and Metacomputing in Java. Concurrency Practice and Experience **10** (1998) 1043–1061
2. Attali, I., Caromel, D., Contes, A.: Hierarchical and declarative security for grid applications. In: High Performance Computing - HiPC 2003, 10th International Conference, Hyderabad, India, December 17-20, 2003, Proceedings. Volume 2913., Springer Verlag, Lecture Notes in Computer Science, LNCS (2003) 363–372
3. Wesley, A., ed.: .NET Framework Security. Addison Wesley Professional (2002)
4. Grimshaw, A., et al., W.W.: The Legion Vision of a World-wide Virtual Computer. Communications of the ACM **40** (1997)
5. Foster, I.T., Kesselman, C., Tsudik, G., Tuecke, S.: A Security Architecture for Computational Grids. In: ACM Conference on Computer and Communications Security. (1998) 83–92