Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Methodology and Design Alternatives for Federation and Interoperability

D3.5.15

Due date of deliverable: February 28^{th}, 2010
Actual submission date: March 24^{th}, 2010

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: WP3.5
Task number: T3.5.4

Responsible institution: STFC
Editor & and editor's address: Alvaro Arenas
STFC
Didcot, Oxfordshire
OX11 0QX, UK

Version 1.0 / Last edited by STFC / March 22^{nd}, 2010

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	$\surd$
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.0	25/01/10	Alvaro Arenas	STFC	first draft
0.1	19/02/10	Erica Yang	STFC	added DToken delegation
0.2	22/02/10	Erica Yang	STFC	updated DToken delegation
0.3	25/02/10	Yvon Jégou	INRIA	added SSO section
0.3	25/02/10	Pushpinder Chouhan	STFC	added interoperability section
0.3	20/02/10	Andrew McDermott	STFC	added information about the credential conversion service
0.3	01/03/10	Benjamin Aziz	STFC	added the sections on security policy challenges and meeting the challenges
0.9	15/03/10	Alvaro Arenas	STFC	Final revision, added summary, introduction and conclusion sections
1.0	22/03/10	Alvaro Arenas, Benjamin Aziz and Pushpinder Chouhan	STFC	Applied reviewers' corrections and comments

Reviewers:
Björn Kolbeck (ZIB), Barry McLarnon (SAP)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.5.4	Federation and Interoperability	STFC*, INRIA

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Contents

Glossary

AEM	Application Execution Management
CDA	Credential Distribution Authority
CA	Certification Authority
GUID	Global User Identifier
GVID	Global VO IDentifier
OGSA	Open Grid Services Architecture
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
PKI	Public Key Infrastructure
RCA	Resource Certification Authority
SAML	Security Assertion Markup Language
VOPS	Virtual Organization Policy Service
VOLife	Virtual Organization Lifecycle Service
VOMS	EGEE Virtual Organization Membership Service
XACML	eXtensible Access Control Markup Language
XtreemFS	XtreemOS File System
XVOMS	XtreemOS Virtual Organization Management Service

List of Figures

Executive Summary

This deliverable presents work developed within XtreemOS in two areas: federation and interoperability. Federation is achieved by implementing single sign-on and delegation across a XtreemOS infrastructure. Interoperability between XtreemOS and the gLite middleware is analysed, focusing on security issues such as certificate management, security policies and Virtual Organisation (VO) management.

Single sign-on in XtreemOS is based on a credential store trusted by all other operating system services, where the user can upload his public certificates and through which all grid requests from user space to XtreemOS services are forwarded. This credential store initiates a key challenge to check that the user owns this certificate and applies the certificate chain validation when the certificate is uploaded. The certificates remains in the credential store until the user closes his session or until it is not valid anymore. This approach can be also used to achieve delegation. The software stack developed for the integration of Single-sign-on and delegation in XtreemOS contains a new operating system service (`SSO-XOS`), a client library for trusted nodes (`libSSO`), a client library for untrusted nodes (`libuntrustedSSO`), an SSH-XOS subsystem exploited by `libuntrustedSSO` (`ssh-xos-sso-client`) and another SSH-XOS subsystem for credential delegation (`ssh-xos-sso-delegation`).

The deliverable also presents DToken, a complementary approach to the standard delegation in XtreemOS, which allows one to achieve lightweight and traceable delegation. DToken is lightweight because it reduces system vulnerability and enhances system manageability by eliminating the use of freshly generated key pairs in a distributed setting, which subsequently removes the major performance bottleneck suffered by classical Grid delegation solutions. DToken is also traceable because the principalŠs identity in a delegation chain is preserved by cryptographically verifiable mechanisms. Implementation of the DToken solution uses the OpenSSL toolkit. The overall cost of creating a DToken, the major cost of the DToken delegation architecture, is roughly 1/3, 1/5, and 1/10 of that of creating a proxy certificate when the certificate key size is 512, 1024, and 2048 bits, respectively. These results demonstrate that our proposal provides significant performance gain over the proxy certificate approach.

The deliverable also reports experiences on achieving interoperability between XtreemOS and the gLite middleware. In relation to certificate management, both XtreemOS and gLite certificates are based on the standard X.509 certificates, but differ in the optional fields and extensions. To deal with these differences, it is presented a credential conversion service that acts as a gateway converting gLite certificates provided by the UK National Grid Service into valid XtreemOS certificates. In relation to security policies, main interoperability challenges con-

cern with the policy language, ontology issues and enforcement of policies. Both XtreemOS and gLite could include policies written in the XACML language, facilitating their interoperability. In relation to VO management, the deliverable analyses difference in membership management, by comparing EGEE VOMS with XtreemOS X-VOMS, and information security of VO management. Finally, interoperability is exemplified in a case study about a virtual marketplace of computational resources across gLite and XtreemOS Grids.

1 Introduction

This deliverable presents work developed in two areas: federation and interoperability. Federation is achieved by implementing single sign-on and delegation across a XtreemOS infrastructure. Interoperability between XtreemOS and the gLite middleware is analysed, focusing on security issues.

Single sign-on in XtreemOS is based on a credential store trusted by all other operating system services, where the user can upload his public certificates and through which all grid requests from user space to XtreemOS services are forwarded. This credential store initiates a key challenge to check that the user owns this certificate and applies the certificate chain validation when the certificate is uploaded. The certificate remains in the credential store until the user closes his session or until it is not valid anymore. This approach can be also used to achieve delegation.

The deliverable also presents DToken, a complementary approach to the standard delegation in XtreemOS, which allows one to achieve lightweight and traceable delegation. DToken is lightweight because it reduces system vulnerability and enhances system manageability by eliminating the use of freshly generated key pairs in a distributed setting, which subsequently removes the major performance bottleneck suffered by classical Grid delegation solutions. DToken is also traceable because the principal's identity in a delegation chain is preserved by cryptographically verifiable mechanisms.

Finally, the deliverable reports experiences on achieving interoperability between XtreemOS and the gLite middleware. It is analysed interoperability challenges in relation to certificate management, security policies and Virtual Organisation management. Interoperability is exemplified in a case study about a virtual marketplace of computational resources.

The structure of the document is the following. Chapter 2 presents single sign-on and delegation in XtreemOS. Then, Chapter 3 describes how to achieve traceable delegations by using the DToken approach. Chapter 4 discusses interoperability between XtreemOS and gLite. Finally, Chapter 5 concludes the document and highlights future work.

2 Single Sign-On in XtreemOS

2.1 Introduction

XtreemOS users are managed by Virtual Organizations and are not registered on the grid nodes. User credentials on an XtreemOS grid are stored inside X.509 identity and attribute certificates distributed by their Virtual Organizations. These credentials are exploited by XtreemOS core and resource nodes configured to handle requests received on behalf VOs. In the security architecture of XtreemOS, users get their credentials inside X.509 certificates from their Credential Distribution Authority (CDA) running on behalf of their VOs. As these certificates are stored in user space, they need to be validated each time the user transfers them to some XtreemOS service. X.509 certificate validation involves private key challenge to make sure that the user owns this certificate and certificate chain validation to check that it has been delivered by a recognized certificate authority.

Having to type a password for each request to the grid is really painful, not very secure and makes scripting difficult. Single-sign-on systems avoid the needs to re-type a password each time. Various single-sign-on mechanisms have been proposed so far through agents, through secure authenticated channels or using proxy certificates. Using a single-sign-on agent, all key challenges are handle by a some service (agent) acting on behalf of the user. A typical example is the SSH agent. The user uploads his password to his agent which then replies to all challenge requests. SSH also provide an even more efficient single-sign-on mechanism through the control master which implements a secure channel between the user and a node. This secure channel is established during the first connection to the node and reused for later requests. This technique avoid the rather expensive key challenge step but is limited to connections between a single user to a single node. X.509 proxy certificates used in many grid systems (Globus, Glite) provide single-sign-on when their private keys are not protected by a password. Password-less certificates or proxies are considered secure enough as long as their validity period is short (a few hours).

Delegation happens when a user (or a service) transfers part of its rights (credentials) to some other service. This mechanism allows the *delegatee* to act on behalf of a *delegator*. For instance, using some form of delegation, an XtreemOS user would delegate his rights to access the grid to his jobs running on resource nodes. Single-sign-on is a restricted form of delegation. Proxy certificates are used in Globus-based middleware to provide delegation: a new password-less key pair is generated by the *delegatee*, the *delegator* generates a new proxy-certificate integrating the proxy public key and signs it using its certificate or proxy private key.

2.2 XtreemOS SSO system

A certificate needs to be checked each time it is moved from the user space to the operating system space in XtreemOS: it is necessary to check that the user owns this certificate and that it has been signed by a recognized authority. The basic idea for XtreemOS is to provide some credential store trusted by all other operating system services where the user can upload his public certificates and through which all grid requests from user space to XtreemOS services are forwarded. This credential store initiates a key challenge to check that the user owns this certificate and applies the certificate chain validation when the certificate is uploaded. The certificates remains in the credential store until the user closes his session or until it is not valid anymore.

The user processes interact with the SSO service through the client-SSO library. This library provides entry points for certificate management and for all grid requests.

2.2.1 User authentication on the SSO system

The SSO system is an operating system service running with operating system credentials and listening for requests on a socket. To authenticate users, this service records the local UID/GID of the local process uploading credentials and then restricts access to these credentials to processes running with the same Unix credentials (the same UID/GID). In order to provide this simple and efficient scheme, the listening socket is a UNIX socket, which allows to check peer credentials. Only local processes running on the same node as the SSO service can communicate with this service. The fact that this service does not run with user credentials eliminates the risk that a user could modify the credentials through `ptrace` or through the `\proc` file-system.

2.2.2 User isolation in the SSO system

The SSO system proposed for XtreemOS can be run in private as well as shared mode. When started by a local user, the system runs in private mode and accepts only requests from this user. Moreover it is possible, in this case, to restrict access rights to the listening socket to this single user. However, also it is working on behalf a single user, private SSO services still run with operating system credentials.

When started by root (by `init.d` for instance), the SSO service runs in shared mode: all users can upload credentials and submit grid requests through the listening socket. Shared SSO services select the credentials to be associated to a request from the UID/GID of the calling process.

2.2.3 Grid requests submission

The client-SSO library provides an API for all XtreemOS grid requests. When the client-side handles a grid request, job creation for instance, it opens the SSO socket (whether it contacts a shared or a private SSO service is configured in the user `sso.cfg` configuration file), transfers the request data to the service and waits for the result. The SSO service piggy-backs the user credentials in this requests and forwards it to the destination service (DIXI bus in general). The destination service trusts the SSO service: both run with operating system credentials and can authenticate themselves using their service certificates. Single-sign-on is provided by the fact that key challenge is necessary only when the credentials are uploaded. Although the certificate chain can be validated when the credentials are uploaded, extra verification might be necessary each time they are used: validity period, non-revocation, etc.

2.2.4 Delegation using SSO services

The user credentials associated to a job can be automatically uploaded (or uploaded only when requested through some extension to the JSDL) in a local SSO service on each resource node by the Execution Managers of XtreemOS. However, as the credentials received by the Execution Managers are currently limited to a single X.509 token, and as the DIXI software bus currently provides no means to upload credentials, some other means need to be provided for delegated credential management. Two solutions have been evaluated to transfer credentials between SSO services: connection through some `INET` socket and XOS-SSH subsystem.

It is possible to provide some `INET` socket bound to a local port for each SSO service. SSO service can then transfer credentials directly through these sockets. Security is provided using `SSL/TLS` and authentication using the service certificates of the SSO services. The major limitations of this solution concern security and configuration. The presence of the `INET` socket makes DOS attacks on SSO services possible. Configuration is made more complex, mainly for private SSO services as the port number where a delegatee service is listening needs to be known by the delegator.

Through a dedicated SSH-XOS subsystem, a SSO service can connect to the resource node using its XtreemOS X.509 service certificate and upload user credentials on a shared service or start a private service. The main advantage of this solution is that as SSH-XOS is already configured in XtreemOS on all resource nodes, there is no need to allocate any new port for communication.

2.2.5 Support for untrusted clients

Untrusted XtreemOS clients such as mobile devices or user laptops cannot be trusted when users can modify their configuration. These devices cannot run trusted SSO services. The solution proposed for this case is to allow users to communicate with SSO services running on trusted nodes trough SSH-XOS with their XtreemOS certificate. The untrusted-client-SSO library provided on these devices export the same API as the client-SSO library. The difference is that this new library communicates with the SSO service through a new SSH-XOS subsystem. The subsystem executed on the trusted node authenticates the user with his XtreemOS certificates, contacts the local shared SSO service (or starts a private one) and can then manage credentials or make grid requests. It is possible to configure the SSH subsystem used for authentication in the SSO service. As this untrusted-client-SSO library provides the same API as the trusted node client-SSO library, mobile device applications get the same capabilities to access the grid as if they were running on trusted nodes. A list of nodes where the subsystem uploads the user credentials can be defined in a local user configuration file. Single-sign-on for the SSH-XOS subsystem is provided by a SSH agent and/or using the SSH control master. The control master option should give better performance as it avoids the authentication phase for each request.

2.3 Extra functionalities

2.3.1 User private sockets

In order to better isolate user credentials, a shared SSO service can provide private UNIX socket to its users. Access to these private sockets is limited to the user through file-system access rights. Using private sockets increases security and isolation and should provide better resistance to DOS attacks as only the associated user can open his private socket.

2.3.2 Compatibility with with XtreemOS 2.0 and 2.1 releases

The XtreemOS SSO system is to be introduced in release 3.0 of the operating system. In order to maintain compatibility with the previous releases, the SSO service will automatically upload the user certificate present in user grid requests. When the SSO service receives a Grid request containing some X.509 certificate from a user, it checks if this certificate has not been already validated. If the certificate is unknown in the user credential list, the validation procedure is applied: key challenge and chain verification. If the SSO service has already received this certificate and if this certificate is still valid, the key challenge step is skipped. It

seems to be also possible to skip certificate chain validation in some cases (non-revocation checks as well as validity period verification are mandatory). If the grid request contains no certificate, the SSO service will piggy-back the currently selected certificate in the request sent to the destination service.

2.3.3 Credential types and credential history

The basic SSO system described in section 2.2 considers one credential type only: X509 certificate. The current implementation of SSO considers various credential types. For each new credential type, SSO must be extended to integrate their validation procedure. SSO can also be configured to record the history of user credentials: how, when and on which node they were validated, usage logs, etc. The history is considered as a special credential type and is replicated with the other credentials during delegation.

2.3.4 Credential delegation

In order to support credential delegation between users for highly dynamic VOs (extension from INRIA), the SSO system allows users to delegate their credentials to other users of the system. In a user delegation request, a user specifies to which user (name or UID) he delegates his credentials and with which capabilities. Using capabilities, it is possible to restrict the delegation rights to monitoring, for instance. Credential delegation is in its early development.

2.3.5 Distributed SSO service

The current implementation of SSO provides two functionalities on trusted nodes: credential store and Single-sign-on. Everywhere a user process need to make grid requests, the service needs to be replicated. For the future, we plan to evaluate the possibility to transform the SSO credential store into a distributed service. Using a distributed credential store, credential uploaded on some node can be exploited on other nodes. It should be possible to implement a distributed credential store on some kind of overlay. And, if the overlay securely manages credential deletion, certificate revocation can be efficiently implemented: it should be enough to delete the certificate from the credential store.

2.4 SSO software

The software stack developed for the integration of Single-sign-on and delegation in XtreemOS contains a new operating system service (`SSO-XOS`), a client library

for trusted nodes (libSSO), a client library for untrusted nodes (libuntrusted-SSO), an SSH-XOS subsystem exploited by libuntrustedSSO (ssh-xos--sso-client) and another SSH-XOS subsystem for credential delegation (ssh--xos-sso-delegation).

2.4.1 SSO-XOS service

The SSO-XOS service runs as a shared service when started by root and as a private when started by a user. The SSO-XOS binary is set-uid and, in both cases, this service runs with operating system credentials (sso:sso or root:root). When started as a shared service, it reads its configuration file from /etc/xos/-config/sso.cfg by default. Private SSO services read their configuration from $HOME/.xos/sso.cfg by default. By default, the listening sockets are located in /var/run/sso/. The shared socket is /var/run/sso/shared and private sockets are named /var/run/sso/$USER.

2.4.2 libSSO library

The libSSO library allows applications running on trusted nodes to directly access the XtreemOS operating service through a shared or private local SSO service. This library reads its configuration from $HOME/.xos/sso.cfg by default (or /etc/xos/config/sso.cfg if the user does not provide any configuration). It provides the API for managing credentials in the SSO service and for calling XtreemOS services: libSSO replicates the whole API of libXATICA.

2.4.3 libuntrustedSSO library

The libuntrustedSSO library allows applications running on untrusted nodes to directly access the XtreemOS operating service through a shared or private SSO service running on a trusted node. This library reads its configuration from $HOME/.xos/sso.cfg by default or /etc/xos/config/sso.cfg if the user does not provide any configuration). It provides the same API as library libSSO installed on trusted nodes. Through dynamic binding, it should possible to run the same application binaries on trusted as well as on untrusted nodes.

2.4.4 ssh-xos-sso-client SSH-XOS subsystem

The ssh-xos-sso-client SSH-XOS subsystem is used by the libuntrus-tedSSO library to connect to a SSO-XOS service running on a trusted node using the user's XtreemOS certificate. This subsystem is made of a client part executed by the application and a server part executed by the SSHD-XOS daemon.

2.4.5 `ssh-xos-sso-delegation` SSH-XOS subsystem

The `ssh-xos-sso-delegation` SSH-XOS subsystem allows a SSO service to delegate credentials to a SSO service running on another trusted node using its XtreemOS service certificate.

2.5 Discussion

The SSO solution proposed in this section is a pure *operating system* solution to single-sign-on and delegation. In classical operating systems, some credentials are bound to users when they log in, UID/GID for instance in Unix systems. All processes running of behalf of the user have these credentials. When a user process accesses some object, access control is in general based on the process credentials. There is no need to asks the user if he allows his process to access the object. This is a form of delegation managed by the operating system: the operating system ensures the binding of user credentials to his processes. The solution proposed in this document is similar: XtreemOS guarantees the binding of user credentials to all grid requests sent by the user to XtreemOS services.

At the opposite, the single-sign-on and delegation solution provided by X.509 proxy certificates is a pure cryptographic solution and is not based on the underlying operating system. X.509 proxies are stored in user space and all verifications happen in user space. The sole implication of the operating system is in the protection of private keys through file access rights.

The DToken solution proposed in section 3 is basically a cryptographic solution with support from the operating system: at the opposite of the X.509 proxy certificate solution, the keys used for signing DTokens belong to operating system services and not to the users.

3 Delegation in XtreemOS via DToken

This chapter presents DToken[1] , a lightweight and traceable delegation architecture for distributed systems, including a distributed Grid-wide operating system, such as XtreemOS.

Several major techniques have been proposed to address delegation problems in distributed computing environments of various scales, ranging from LAN, WAN, to the Internet. In the Grid world, RFC3820 [1] is the de-facto standard for delegation. This specification introduces a notion - Proxy Certificate as a delegation concept to describe the delegation chain between two or more parties in a distributed system. Proxy certificate, as one of the most popular forms of delegation tokens, is based on public key cryptography. Hence, in our work, we also focus on delegation technique using public key cryptography.

One of the major characteristics of existing public key cryptography based delegation mechanisms is their use of a fresh key pair every step along the delegation chain. This has led to a range of open issues, including a non-negligible performance overhead imposed by using a fresh key pair in proxy credentials; the lack of traceability of the principals in a delegation chain; and the complexity of managing the dynamically created key pairs in the distributed environment. This work focuses on the architectural issues of delegation. We propose a new delegation architecture, called DToken, that takes advantage of the PKI. DToken is lightweight as it eliminates the use of freshly generated key pairs in a distributed setting. DToken is also traceable because the identity of the principals in a delegation chain is preserved by cryptographically verifiable mechanisms. A preliminary evaluation demonstrates that DToken outperforms proxy certificate. We will demonstrate that in a single-level delegation, the cost of creating a DToken, the major cost of delegation, is roughly 1/3, 1/5, and 1/10 of that of creating a proxy certificate when the certificate key size is 512, 1024, and 2048 bits, respectively.

3.1 Introduction

The problem of delegation for computer systems has attracted significant interest in recent years, see, for example [4, 1, 5, 2, 3]. Delegation occurs when a principal (a person, a process, a service or a machine) *authorizes* another principal to act on its behalf. The receiver of a delegation relationship, called a *delegatee*, inherits all (or part of) the rights and responsibilities from the initiator of the relationship, called a *delegator*.

[1]The major parts of this work has been published at The 28th IEEE Symposium on Reliable Distributed Systems by Erica Y. Yang and Brian Matthews as a full paper. The acceptance rate (full paper) for this conference is 22% (23/104).

Delegation of rights is pervasive in computer systems. The typical cases of delegation include: forking a process on behalf of a user logging in to a computer; starting a new process for a user already logged in; running a job or an application; and accessing a file system locally or remotely on behalf of an authenticated user. All these cases can occur within a standalone or a distributed system.

In the 90's, delegation was often studied in the context of a distributed system where critical services are centrally managed, meaning that they are deployed in trustworthy premises and operated by trusted operators within a single organization. However, because resources are distributed across a closed network which is perceived as untrusted, security techniques were introduced into delegation mechanisms to provide assurances such as accountability, non-repudiation and traceability. At around the same time, two representative approaches emerged. The proxy concept was introduced into Kerberos to support restricted proxying by leveraging conventional cryptography [5]. And a notion based on public key cryptography, called delegation certificate, was introduced to enable delegation in the Distributed System Security Architecture (DSSA) [4]. This certificate has a private counterpart, named delegation key, which allows a delegator to authorize the delegation certificate provided by a delegatee to enable further delegation.

In recent years, with the development of Grid middleware (e.g. Globus and Glite) and their deployment in wide-scale Grid infrastructures (e.g. EGEE in Europe, TeraGrid in the U.S., National Grid Service in the U.K.), delegation techniques have been widely employed to enable accountability and (distributed) authorization for coordinated resource sharing and collaboration in large-scale decentralized distributed systems. Such systems differ from centralized distributed systems in two major aspects: a) it is hard to ensure the same level of trust across all services because they are often managed decentrally by different organizations under different standards; b) enforcing conventional identity-based authorization (e.g. via access control lists) is no longer a scaleable approach because there are potentially thousands of machines and users from around the world.

3.1.1 The Problem

Proxy certificates, an RFC standard [1] proposed as part of the Grid Security Infrastructure (GSI), are a widely deployed delegation solution implemented upon OpenSSL and Grid protocols. It differs from the DSSA delegation solution in two major aspects [2]: the former is based on the standard X509 certificate format whilst the latter is not; and the former comes with an open source reference implementation being widely deployed in practice whilst the situation with the latter is unclear.

However, despite their popularity, proxy certificates suffer from three major limitations, as discussed in [2, 1]. First, proxy certificate generation imposes a

non-negligible performance overhead on server side key generation because of the use of a fresh key pair in proxy credentials. In practice, as the authors noted, this also can cause concerns regarding Denial-of-Service attacks. Secondly, proxy certificates are generated by a delegatee (typically a server) and signed by a delegator (typically a user or a predecessor server in a delegation chain). But only the identity of the originating delegator (often the user) is preserved in the current proxy certificate solution, and no information about servers later the delegation chain can be identified. This is referred to as the delegation tracing problem and was noted in rfc 3820 [1] but it was left as future work. Such information can be useful to the principals in the delegation chain to determine whether they would like to reject the chain because some of the participants have been compromised. Finally, the use of a fresh key pair for dynamic delegation also makes it difficult to ensure a consistent level of security for the private keys throughout the entire delegation chain due to the scale and heterogeneity of the distributed system. In theory, both proxy certificate and the DSSA solutions both allow users to "revoke" the delegation, thus invalidating the delegation chain, by deleting the private key from their local computer. But this is based upon the assumption that a compromised server cannot exercise its delegated rights because all the other servers in the chain behave correctly by eliminating their local private key when the delegation is no longer needed. Similar to the authors of [6], we believe it is reasonable to ask how to ensure this assumption in practice. There are two major issues here. One is how to determine when the delegation chain is no longer needed (while the keys are still valid). Neither solution gives hints on how this should be implemented. The other problem is that because servers are managed by different organizations, it is difficult to ensure all the sites offer the same level of security standards.

3.1.2 Contributions

The contributions of this work are summarised as follows.

1. A novel practical public key cryptography based delegation architecture, called *D-Token*, is proposed to tackle the major limitations of existing public key cryptography based delegation approaches (e.g. [4, 1]).
2. The DToken delegation protocols are presented in the context of two typical delegation scenarios when accessing a distributed system: via a (trusted) gateway and via a (untrusted) public computer.
3. We demonstrate, through preliminary experimental evaluation, that our implementation reduces the overall cost of delegation to roughly 1/3, 1/5, and 1/10 of that with an existing popular delegation implementation.

3.2 Context and Definitions

We consider a distributed system S consisting of a set of system components deployed on a set of n machines $M = \{m_1, m_2, ..., m_n\}$. S provides services to its users, such as accessing files stored in a distributed file system or running processes/jobs on a remote machine (or machines). A user U accesses the services via a client machine C. U's long-term credentials (a public key identity certificate and the private key) are stored at a user trusted machine TM, which can be a trusted online credential store, a smart card owned by the user, or a trusted personal computer.

C can be classified into two cases. First, C can be a gateway (or a web portal) into S, implemented as a long-lasting process, running on behalf of the user while he is offline. It is typical that such examples of C concurrently deals with different users' requests for the services provided by S. C needs to demonstrate to the machines in M that it is "authorized" by U to perform tasks. Second, C can be a public computer (e.g. a pool of shared computers in a laboratory). In this case, C cannot be trusted, therefore, the user's long-term credentials cannot be stored at C. To access the services, C still needs to demonstrate to the machines in M that it is "authorized" by U. In either case, a delegation between U and C is required to obtain the authorization. There is an open network N (e.g. the Internet) connecting the machines in S, C, and TM.

Threat Model We consider two types of attackers: insiders and outsiders.

An *insider* openly and correctly execute protocols as specified by a protocol specification under their own identity and credentials. But they cannot be trusted to honor their own behaviour after the protocol is finished. An insider might try to subvert the protocol by, for example, denying its participation of the protocol or replaying messages so as to gain advantages of the system (e.g. access resources repeatedly by replaying messages). However, we assume that an insider keeps the secrecy and integrity of its own credentials (e.g. private key).

The network is untrusted, namely there are outsiders attacking N at their wish. We consider an *outsider* (or outsiders) under the Dolev-Yao model [9], where the outsider can: a) eavesdrop and modify any messages transmitted over N; and b) take part in a protocol using his own identity by sending or receiving messages. But an outsider cannot subvert any cryptography protocols (e.g. digital signature schemes) we employ in the system. In other words, the outsider is not able to perform cryptoanalysis over the cryptography protocols to derive useful information, for example the plaintext from a piece of encrypted text, or the private key used for signing a message.

In general, it is not realistic to assume the trustworthiness of C because it is neither always under the control of the system (for example, in the public com-

puter scenario), nor always under that of U. The rest of the chapter shall focus on the delegation problem introduced by the general case.

Secure Channels Because of the untrustworthiness of N and C, we require the interactions among the machines within S and those between C and S to be secure. By a "secure" channel, we mean the channel is mutually authenticated and encrypted. This secure channel requirement can be satisfied by using the standard SSL protocol.

Credentials Our system is based on PKI. Each machine m_i, where $i = 1, 2, ..., n$, in M has a machine certificate C_{m_i} and a corresponding private key K_{m_i}. Hereafter, by a certificate, we refer to the standard X.509 public key certificate. Similarly, C has its own credentials. When it is a gateway, we denote its certificate as C_G and a corresponding private key K_G. When it is a public computer PC, its credentials are denoted as C_{PC} and K_{PC}. The certificates and keys may be distributed to the machines via an online or offline Certification Authority (CA) or from an online credential repository. The certificate/key distribution problem of how the user and the machines obtain, configure, and use these certificates and keys is described in Deliverables D3.5.13 and D3.5.9.

Concepts A *principal* is an entity that can be granted access to objects or can make statements affecting access control decisions [7]. In its basic form, delegation is a relationship between two principals: a delegator D_1 and a delegatee D_2. We denote a delegation from D_1 to D_2 as $D_1 \rightarrow D_2$. D_2 can further delegate to other principals D_3, D_4, ..., D_n, where $n \in N$ and $n \geq 2$, forming a *chained delegation* [4], also called a *delegation chain* or *cascaded delegation* involving principals D_1, D_2, ..., to D_n (and so on), which is denoted by $D_1 \rightarrow D_2 \rightarrow D_3 \rightarrow ... \rightarrow D_n$. An *originating principal* is the principal from who a delegation chain starts, and the number of non-originating principals involved in a delegation chain is called the *delegation depth*. A *final principal*, also called *end server* [5], is the principal who finally executes the delegated rights. An *intermediate principal* is any principal between the originating and final principals in a delegation chain. Clearly, there can be none, one or more than one intermediate principals in the chain. In the above example, D_1 is the originating principal, D_n final principal, D_2, ..., D_{n-1} the intermediate principals. The delegation depth is $n - 1$.

In a distributed environment, delegation of rights is often "restricted" so that only part of the rights is made available to a delegatee to reduce the risk of a compromised delegatee. Typically, three types of *delegation constraints*, also called *delegation restrictions* can be imposed in a delegation: the delegatee's identity

ID_{De} (to restrict who can act on behalf of the delegator); the delegation duration, including a valid-from timestamp V_{fr} and a valid-to timestamp V_{to} (to restrict how long the delegatee can exercise the delegated rights and how long the delegator can reuse a delegation token); and delegation policies $P_{Dr \to De}$ (to restrict the context where the delegation can take place, e.g. delegation depth, characteristics of the delegatee, the scope of delegated operations).

It is often down to the final principal to respect and enforce the constraints when it finally authorizes the operations based on its local policies and the aggregated delegated rights passed down the delegation chain. The delegated rights should be the intersections of all the rights that have been granted by and within the minimum duration specified by the principals along the chain, resulting in the least privileges of the original delegator being enforced at the final principal.

Typically, a delegator passes the delegatee a security token, called a *delegation token* $DT_{D_r \to D_e}$, also called a *delegation certificate* [4] or a *proxy* [5], to allow it act on its behalf under certain constraints. By a "security token", we mean that it is constructed using digital signature techniques to ensure information integrity and provide non-repudiation property. Of course, if the delegator chooses there may be no constraints. In addition to the delegation constraints, a DT should also contain a timestamp TS specifying when the delegation takes place.

In a chained delegation, each principal passes a DT to its immediate descendant *and* all the previous DTs it has received from its ancestors. Thus, D_1 passes a $DT_{D_1 \to D_2}$ to D_2, D_2 passes $DT_{D_1 \to D_2}$ and $DT_{D_2 \to D_3}$ to D_3, ..., and D_{n-1} passes $DT_{D_1 \to D_2}$, ..., $DT_{D_{n-1} \to D_n}$ to D_n.

In this work, "a mutually authenticated channel" is treated as a synonym of "a SSL or TLS channel" because the latter is a well-established PKI based technique to build the former. In practice, a SSL/TLS channel is also encrypted, although the delegation scheme and architecture presented in this work does not require this feature.

Properties of delegation Depending on the cryptography techniques that it employs, a delegation scheme can have a range of properties, such as traceability, accountability, non-repudiation, and authorization.

A *traceable* delegation is one where each principal in a delegation chain is uniquely identifiable. That is, each principal can find the identity of any of the principals prior to itself in the delegation chain. If a delegation is traceable, we also call the delegation satisfies *traceability* property.

Definition 3.1 (*A traceable delegation TD) A traceable delegation TD is a delegation such that identity of any principal(s) prior to D_i, where $2 \leq i \leq n$, that is ID_{D_1}, ID_{D_2}, ..., $ID_{D_{i-1}}$ are uniquely identifiable through examining the DTs it receives (i.e. $DT_{D_1 \to D_2}$, ..., $DT_{D_{i-1} \to D_i}$).*

A traceable delegation is typically also *accountable* if cryptography measures are in place to allow other principals to *verify* the identity of any principals prior to itself. If a delegation is accountable, we also say that the delegation satisfies the *accountability* property.

Definition 3.2 (*An accountable delegation AD) An accountable delegation AD is a delegation such that identity of any principal(s) prior to D_i, where $2 \leq i \leq n$, that is ID_{D_1}, ID_{D_2}, ..., $ID_{D_{i-1}}$ are not only uniquely identifiable but also cryptographically verifiable through validating the DTs it receives (i.e. $DT_{D_1 \rightarrow D_2}$, ..., $DT_{D_{i-1} \rightarrow D_i}$).*

A TD is not always accountable, but a AD is always traceable. That is, traceability is a necessary but not sufficient condition of accountability. An accountable delegation is also called a non-repudiated delegation.

A *restricted* delegation is one that the delegator can specify delegation policies over the delegation.

Definition 3.3 (*A restricted delegation RD) A restricted delegation RD is a delegation such that any principal D_i, where $1 \leq i \leq n-1$, can impose delegation policy $P_{D_i \rightarrow D_{i+1}}$ upon the delegation.*

However, not all the delegation solutions satisfy all three properties. For instance, the proxy certificate approach does not support traceability or accountability because the identity of the principals along a delegation chain is lost in the delegation process. In the rest of this chapter, we describe how the DToken architecture satisfies all three properties.

3.3 Overview of the Architecture

In its basic form, delegation is defined as a relationship between two parties: a delegator and a delegatee, where the delegator *initialises* the delegation and the delegatee *accepts* it. This is often achieved by two parties jointly forming *a pair of delegation credentials* which consists of a (public) delegation token and a (private) delegation key.

Through the delegation token, the DToken delegation architecture presented in this chapter allows a machine to check whether a user has delegated to one (or more) machine(s) without directly contacting the user or any trusted online third-party service. In a large-scale distributed environment, this is an important feature that allows the system to cope with scalability requirements (e.g. thousands of users who may concurrently access many services). This architecture is similar to that implemented by Kerberos [5], DSSA [4], and GSI [1]. The major differences

are briefly highlighted as follows. Kerberos requires the presence of an online trusted third party to initiate the delegation. Because the use of public key cryptography, this is not a requirement for DSSA, GSI, or our solution. In addition, DSSA and GSI require the generation of public key pairs (public key and private key) at every intermediate principals in the delegation chain on a session basis. The main argument for employing fresh key pairs in a delegation chain in DSSA and GSI is to minimize the risk of a compromised principal [4, 2, 1].

The DToken architecture has several features that differentiates it from the existing solutions. It allows any non-originating principals in a delegation chain to: 1) extract the identity of all the principals prior to itself from the delegation tokens passed down through the chain, thus providing the traceability property of the chain; 2) verify whether any intermediate principals has actually accepted the delegation via a dual digital signature scheme , thus providing non-repudiation or accountability property of the chain; 3) determine whether it accepts (or rejects) any ancestor principals in the chain based on its own local trust policy (e.g. which CAs it trusts) without directly communicating with them; and 4) manage delegations with their existing long-term credentials (i.e. long-term certificate and private key) without the need for managing newly generated key pairs (or short-term certificate and private key pairs).

To the best of our knowledge, all these features are not present in any of the existing public key cryptography based delegation solutions. Because of the absence of these features, all the existing solutions place significant trust assumptions upon principals in a delegation chain. In practice, this can hinder its practical deployment in wide-scale cross-organization deployment environment if the level of trust upon these principals is of a concern.

For example, in the conventional solutions (e.g. [4, 1]), all intermediate principals have to be trusted to eliminate their local private (delegation) key when a delegation is no longer needed. Details are not given on how an intermediate principal determines when a delegation is no longer needed, leaving a window of vulnerability for the delegation between the time which a principal becomes compromised and when the principal deletes the private key. An implicit assumption arising from this is that as long as the delegation chain is valid, all the intermediate principals should remain trusted (i.e. the dynamically generated private delegation key and the permanent private key of an intermediate principal are not compromised). As pointed out by Broadfoot and Lowe [6], a) this trust assumption is a very restricted one when employing this delegation architecture in a practical environment, such as the Internet; and b) without this assumption, the DSSA delegation architecture is rendered insecure. Because we do not employ freshly generated key pairs, our proposal does not suffer from this problem.

Second, all the intermediate principals also have to be trusted to maintain a *persistent* association between the newly generated key pair and the *dynamically*

generated process at an intermediate principal. In DSSA [4], a principal is a universal abstraction of different types of principals, such as user, workstation, process, system, or server. But, in practice, there is a fundamental difference between these principals in the way they are named (identified) and certified (i.e. issuing of public key identity certificate). Unlike the other principals, a process is a dynamic entity generated by a computer system which does not have a permanent global identifier (or name). Therefore, unlike a user or a machine, which can be issued a certified identity certificate, it is impossible to associate a process (which doesn't have a persistent global identity/name) with a public key pair. It is unclear how the DSSA delegation architecture resolves such an issue. In contrast to DSSA, GSI explicitly acknowledges the problem of identifying a dynamically generated process. It identifies the process with Distinguished Name (DN) dynamically generated from the human-readable DN of the delegator extended with the hash of the dynamically generated public key for the session. However, within a computer system, the identifier of a process, that is its Process IDentifier (PID) is managed locally. Therefore, it is still ultimately down to the intermediate principals to maintain a persistent association between the PID and the DN of the process. Again, we still have to assume the trust of the intermediate principals. Therefore, we reach the conclusion that it is unclear how the freshly generated key pair helps to reduce the level of trust required on the intermediate principals.

Thus, from our point of view, the advantages of using fresh public key pairs in the DSSA's and GSI's delegation architectures are not clear. Instead of using the fresh key pairs, we propose to exploit the permanent key pair associated with each intermediate principal to support the delegation. The major advantages of our approach is two fold: a) we elminate DSSA's and GSI's need for key pair generation to avoid the major performance bottleneck suffered by these systems; and b) the persistent association between a dynamically generated process and the key pair is no longer needed. In our solution, because there is no freshly generated key pairs, managing the association between a delegation token and dynamically generated processes remains the internal problem of the services in a delegation chain. In addition to the performance gain, the elimination of key pair generation from the DToken delegation process also makes our approach less vulnerable to Denial-of-Service (DoS) attacks, one of the major security concerns of proxy certificate.

3.4 A Single-level Delegation

A single-level delegation is a special case of a chained delegation where there is one intermediate principal between the originating and final principals. It is also a fundamental representative case which is used here to explain some of the important features of the DToken architecture. The delegation example used in

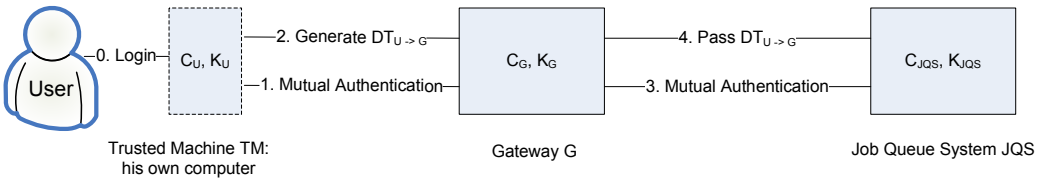


Figure 1: Single-level Delegation in a Gateway-style Distributed System

this section involves three machines: a user machine TM, a client machine C, and a Job Queue System (JQS). TM is a trusted machine (e.g. the user's smart card or an online credential repository) of the user and it is where the user's long-term public key identity certificate C_U and private key K_U reside.

We describe the process of single-level delegation between U and C through a joint creation of a DToken and how the DToken is used between C and JQS. In the model, we use a single abstraction C to represent the intermediate principal between TM and JQS. In practice, the nature of C has an impact on how the system should be designed. Therefore, our presentation will be split into two parts to deal with the following cases: 1) C is a trusted gateway G into the system where G holds its long-term certificate C_G and private key K_G; and 2) C is a public/shared (i.e. untrusted) public computer PC, where PC has C_{PC} and private key K_{PC}.

3.4.1 The Gateway Case

As illustrated in Fig. 1, in this scenario the gateway G has a *persistent* process running as a daemon of the distributed system, typically to support batch job processing. This is a typically architecture style for implementating Grid systems. Accessing to system services (e.g. JQS) needs to go through this gateway.

Creating a Process on the Trusted Machine U authenticates to TM following the local procedures. Once logged in, TM starts a *local* process Pr_{TM_u} running on behalf of U allowing it to use the user's long-term credentials (i.e. C_U and K_U) to interact with G. It is worth noting that on some systems, apart from being protected by local file system protection mechanisms (e.g. the file permission bits), K_U may also be protected by a user specified password. In order to use K_U to respond to the key challenge posed by G, U needs to demonstrate its knowledge about that password. As these are the details of the authentication between U and TM, not directly relevant to the delegation process, we have omitted them from the Figure.

First-level Mutual Authentication The gateway typically deals with concurrent requests from different users simultaneously. Before the delegation takes place, U and G need to mutually authenticate with each other to ensure that U's identity is certified by a *trusted CA* of G and vice versa. Similarly, U could initiate multiple concurrent connections with G, for example, to access different types of services (e.g. file accessing, job processing) provided by the system.

Although there are well-established protocols in practice, such as SSL or TLS, to support mutual authentication under PKI, we need the concept of delegation session to differentiate the concurrent delegations, with different delegated rights and constraints, that may occur between U and G. It is also worth noting that although SSL/TLS has a concept of session, it is not suitable for our purpose because it is designed for reusing SSL parameters (e.g. encryption and digital signature algorithm parameters) so that the need for renegotiating SSL parameters can be reduced.

As part of the mutual authentication process, G needs to send a key challenge to U to ensure that U is indeed the holder of K_U and vice versa. *It is also worth noting that the SSL/TLS protocol not only exchange the end entity certificate (EEC) of communicating parties, i.e. C_U and C_G, but also all the subordinate CA certificates (but not the root CA certificate) of the EECs*. The root CA certificates should be installed through trusted offline means by the administrator of TM and G prior to the communication beginning.

We denote all the subordinate CA certificates of C_U as $C_{U_{sCAs}}$ and those of C_G as $C_{G_{sCAs}}$. Therefore, at the end of the mutual authentication process, G has C_U and $C_{U_{sCAs}}$ whilst U has C_G and $C_{G_{sCAs}}$.

Creating a Process in the Gateway Once U and G are mutually authenticated, G starts a *local* process Pr_{G_u} running on behalf of U. As this is a local process to G, it has access to G's long-term credentials (i.e. C_G and K_G). K_G will be used in the next step to sign the DToken. Although this process is created as a form of delegation from U to G and is associated with (the identity of) U, it doesn't have any associated delegated rights since U has not yet been involved in the delegation.

Generating a DToken between the User and the Gateway The creation of a DToken representing the delegation from U and G is a *joint* process between Pr_{TM_u} and Pr_{G_u} over the mutually authenticated channel created in the previous step. U needs to authorize the delegation by signing the delegation information $I_{U \to G}$ with his private key K_U and G needs to accept the delegation by signing $I_{U \to G}$ with its private key K_G. $I_{U \to G}$ consists of two parts: the information provided by U and that assigned by G. The former includes C_U, C_G, V_{fr}, V_{to}, TS,

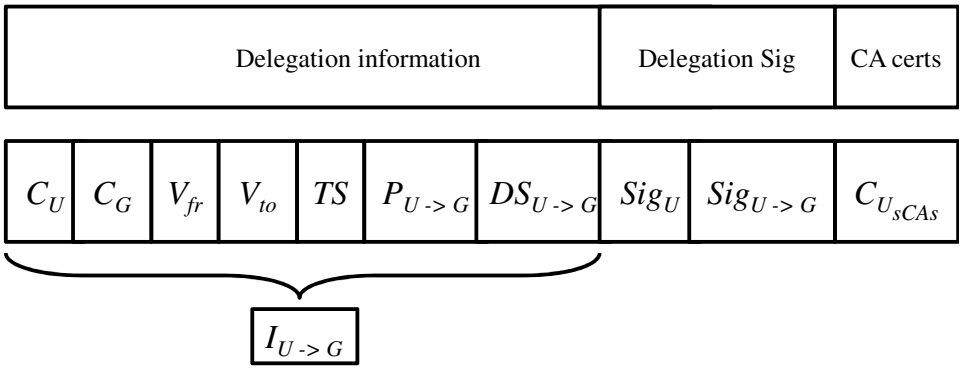


Figure 2: The format of a DToken $DT_{U \to G}$ from U to G in the Single-level Delegation. What is showing here is the format but not the actual content of the token. The content of the $DS_{U \to G}$ is initially empty and will be filled by G in Setp 2 of the protocol of generating a DToken.

$P_{U \to G}$ and the latter the delegation session identifier, denoted as $DS_{U \to G}$.

The signature from G is important for providing the accountability (non-repudiation) of the DToken because U can have all the information, except $DS_{U \to G}$, in $I_{U \to G}$ *without* interacting with G because it is all supposed to be "public". $< I_{U \to G} >_{K_U}$ is the digital signature of the information provided by U signed by K_U. However, when G generates $Sig_{U \to G}$ by signing $< I_{U \to G} >_{K_U}$ with its private key K_G, it cannot deny its acceptance of the delegation, providing accountability for the delegation. As illustrated in Fig. 2, a DToken $DT_{U \to G}$ consists of three pieces of information: delegation information $I_{U \to G}$, delegation signatures Sig_U and $Sig_{U \to G}$, and U's intermediate CA certificates $C_{U_{sCAs}}$, where $Sig_{U \to G}$ is the joint signatures of $I_{U \to G}$ signed by K_U and K_G sequentially. It should be noted that when $I_{U \to G}$ is sent from U, $DS_{U \to G}$ is empty. It is filled by G before $I_{U \to G}$ is countersigned by G. The protocol for generating a DToken is as follows.

1. $U \to G$: send a message containing two pieces of information: a tuple $I_{U \to G}$ and Sig_U, where $I_{U \to G}$ consists of: (C_U, C_G, V_{fr}, V_{to}, TS, $P_{U \to G}$, $DS_{U \to G}$) and Sig_U is the digital signature of $I_{U \to G}$ signed by K_U, that is $< I_{U \to G} >_{K_U}$. Note that at this stage, the $DS_{U \to G}$ field is empty awaiting to be assigned by G.
2. G: verify the validity of the message and check with its local access control policies to ensure that U is authorized to access the resources (e.g. to perform the operations specified in $P_{U \to G}$). When the checks are successful, G fills $DS_{U \to G}$ and signs Sig_U, plus $DS_{U \to G}$, with K_G. This signature is denoted as $Sig_{U \to G}$. Together with C_{U_sCAs}, which is received from the first-level mutual authentication step, G can now generate a DToken $DT_{U \to G}$.

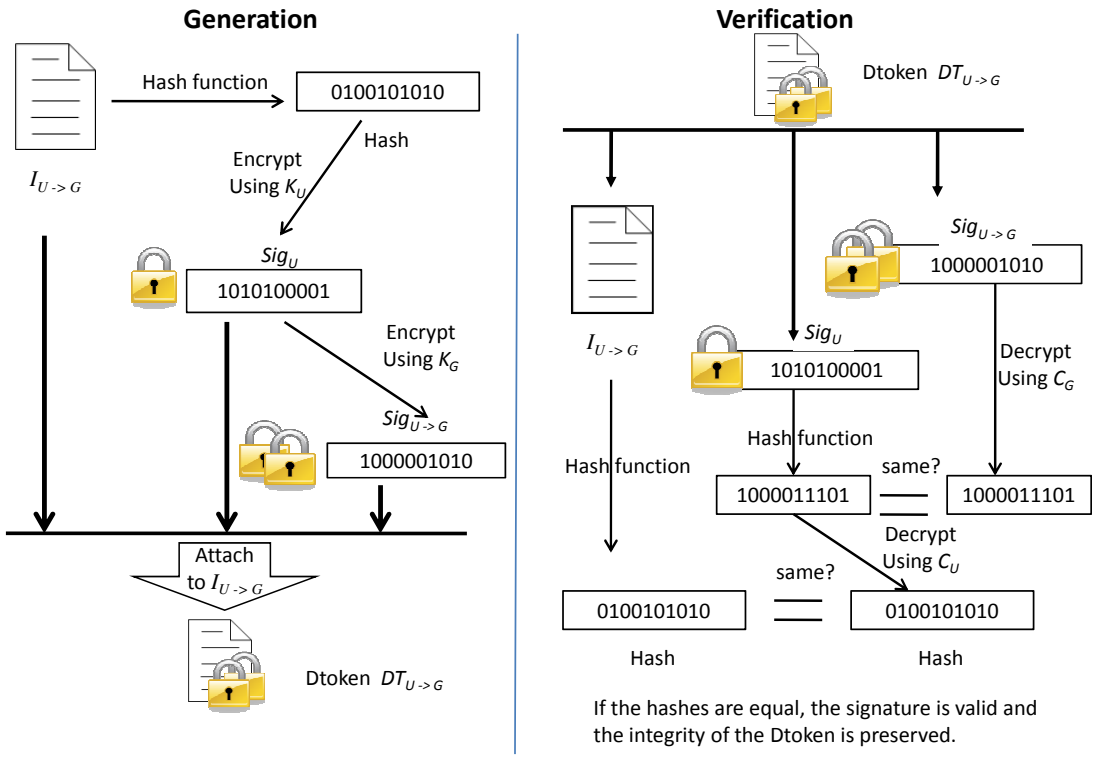


Figure 3: The Generation and Verification of a DToken.

3. $G \rightarrow U$: send $DT_{U \rightarrow G}$.

There are a few points worth noting about the above protocol. In step 1), by signing $I_{U \rightarrow G}$ with K_U, that is, producing Sig_U, U authorizes G to act on its behalf. In step 2), by signing Sig_U with K_G, that is, producing $Sig_{U \rightarrow G}$, G acknowledges that it has performed the validity check of $I_{U \rightarrow G}$ and U is authorized to access the resources governed by G. For the convenience of referencing, Fig. 3 illustrates the major steps involved in generating and verifying (the integrity of) a DToken. For clarity, the principals and their interactions are omitted from the Figure. Also, in step 2), there is no need to check the authenticity of U because that has been confirmed in the first-level mutual authentication. Also, there is no need to sign $C_{U_s CAs}$ as part of the digital signature production process because they are already crytographically signed and thus protected. The reason to include C_U and C_G in $I_{U \rightarrow G}$ is to include the identity of U and G in the delegation token so that they are cryptographically verifiable by anybody receiving the token.

In addition, we only require the underlying communication channel to be mutually authenticated. There is no need to ensure the confidentiality of the information being exchanged over the channel as all the information in the DToken is

"public" and its integrity is ensured by digital signature. To use $DT_{U \to G}$ for exercising delegated rights, a principal needs to prove that it is the legitimate delegatee of the DToken by demonstrating access to the corresponding private key (in this case, it is the private key K_G). This leads us to the next step.

Second-level Mutual Authentication The second-level mutual authentication takes place between G and JQS, the final principal in the delegation chain. The purpose of this step is to ensure that G submits the job request on behalf of U to a G trusted job service - JQS. Similarly, JQS needs to ensure that the request is from a trusted gateway - G. Here, trust means that when a principal submits a certificate, the receiving principal needs to ensure that the certificate is issued by a trusted CA of its own choice.

Similarly to the first-level mutual authentication, we can use SSL/TLS to support this step. If everything goes well, at the end of this step, the following facts about G and JQS are established: G is the authentic holder of K_G and JQS is that of K_{JQS}. G has C_{JQS} and $C_{JQS_{sCAs}}$; whilst JQS has C_G and $C_{G_{sCAs}}$.

Passing and Verifying a DToken Over the mutually authenticated channel between G and JQS, G submits a job request along $DT_{U \to G}$. It is worth noting that the root CA certificate of U is not sent to JQS. It is down to JQS to install it through trusted offline means to ensure its authenticity and integrity.

The subordinate CA certificates allow JQS to perform two checks: a) whether he trusts the CAs (all the subordinate CAs) of U. If not, he will refuse to execute the operations specified in the DToken. b) whether the C_U embedded in the DToken is authentic without relying on (or trusting) the intermediate principal G. This is an important feature that differentiates our delegation scheme from any of the existing public key cryptography based delegation schemes, such as proxy certificate and the DSSA delegation scheme. This feature reduces the risk of the entire system being compromised even if G is compromised as JQS does not need to trust (or rely on) G to perform the above checks.

Recall that after the mutual authentication between G and JQS, JQS can be sure that he has a genuine copy C_G. JQS can ensure the authenticity of C_U by using $C_{U_{sCAs}}$ (extracted from $DT_{U \to G}$) without directly interacting with U. Demanding a direct interaction between the final principal (i.e. JQS in this case) and the originating principal (i.e. U) hinders the scalability of the system because U can become the bottleneck of the system when delegation is a frequent operation in the system. Therefore, this feature enhances the scalability of the system. In addition, a direct interaction between JQS and U implies that U needs to be online to response to the request from JQS (or any intermediate and final principals), and could lead to poor system usability.

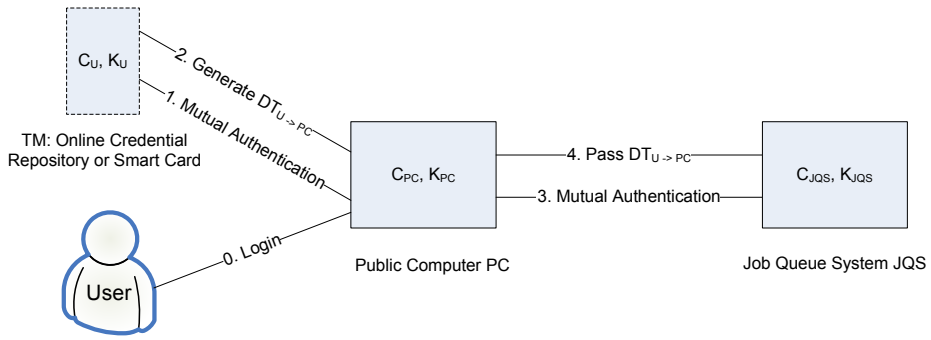


Figure 4: Single-level Delegation through a Public Computer

As illustrated in the right of the Fig. 3, JQS verifies the integrity of the DToken as follows:

1. extract $I_{U\to G}$ Sig_U, and $Sig_{U\to G}$ from the DToken
2. decrypt $Sig_{U\to G}$ using C_G
3. compare the output of step 2) with the hash of Sig_U
4. decrypt Sig_U using C_U
5. compare the output of step 4) with the hash of $I_{U\to G}$

It is worth pointing out that in the last step, $I_{U\to G}$ is the initial content that U has sent to G where the field $DS_{U\to G}$ is empty. If the comparison results of step 3) and 5) are equal, it is not only that the signatures are valid, but also implies the followings:

- Verification 1: the integrity of the DToken is verified.
- Verification 2: G is the authorized to use the DToken. In the second-level mutual authentication step, a two-way key challenge is performed to ensure G holds K_G and JQS has K_{JQS}. Therefore, when the integrity of the DToken is verified, it automatically implies that G is authorized to use it. If not, G would have failed the key challenge in that step.
- Verification 3: the delegation chain ($U \to G \to JQS$) is valid. Verification 1 implies U authorizes G to act on his behalf and G validates that U is authorized to use the resource governed by G.

3.4.2 The Public Computer Case

Fig. 4 illustrates the scenario that U accesses JQS through a public computer PC. Here, the settings are largely similar to those in the gateway case where PC

and JQS have their long-term credentials installed locally. This is represented as the pairs (C_{PC}, K_{PC}) and (C_{JQS}, K_{JQS}) respectively. The method for accessing U's long-term credentials are different from that in the gateway case. Instead of putting his long-term credentials (i.e. C_U and K_U) at PC (which is too risky), U stores them at his trusted machine TM, such as an online credential repository (e.g. MyProxy) or his smart card. Because his long-term credentials are not present at PC, U cannot directly access the services on JQS. He therefore needs to delegate his rights to PC so that it can access the remote services provided by JQS on his behalf. Because of the risky nature of PC, such a delegation has to be conducted on a session (short-term) basis. It would be ideal that once U logs out from PC, the delegation relationship between U and PC is automatically deactivated. This means that PC cannot abuse the delegation after U's session completes.

Creating a Process on the Public Computer U needs to first login to PC following the local authentication procedure supported by PC. Once U successfully logs in to PC, a *local* process Pr_{PC_U} is created on PC for U. Because it is local, it has access to PC's long-term credentials C_{PC} and K_{PC}. It is worth noting that the authentication between U and PC takes place locally without involving any remote service. Then, he needs to authenticate to TM to access his long-term credentials by responding to the challenge presented by TM, for example, using his username and password shared with TM. We believe this is a reasonable assumption in practice. In fact, myProxy is implemented in this way to enhance the usability of online credential repositories by eliminating the need for users to manage their own long-term credentials.

First-level Mutual Authentication Here, the first-level mutual authentication means the authentication between the user U, who has access to C_{PC} and K_{PC}, knows about his username and password on TM, but doesn't have his own long-term credentials. This is different from the gateway case, where the authentication is between TM, who has U long-term credentials, and the gateway machine G. The mutually authenticated channel created in this step is based on the shared password between U and TM. A mutually authenticated channel protects the user from an impersonating server and vice versa. There are well established challenge-response techniques to build a mutually authenticated channel using a shared secret between two communicating parties, see for example, the MS-CHAP v2 protocol [8]. Such a protocol ensures the authenticity of both communicating parties without requiring cleartext passwords to be transmitted over an insecure network.

Creating a Process on the Trusted Machine Once the first-level authentication is successful, TM starts a local process Pr_{TM_U} running on behalf of U. This process has access to U's long-term credentials C_U and K_U.

Generating a DToken between the User and the Public Computer Similar to the gateway case, the creation of a DToken representing the delegation from U to PC is a joint process between Pr_{TM_U} and Pr_{PC_U} over the mutually authenticated channel created in the previous step. However, unlike the standard SSL channel used in the gateway case, this channel is not based on PKI, but based on the shared password between U and TM. Here is the protocol for generating a DToken in the public computer case.

1. $PC \rightarrow TM$: send C_{PC}
2. $TM \rightarrow PC$: send $I_{U \rightarrow PC}$, Sig_U, and $C_{U_{sCAs}}$, where $I_{U \rightarrow PC}$ consists of: (C_U, C_{PC}, V_fr, V_to, TS, $P_{U \rightarrow PC}$, $DS_{U \rightarrow PC}$), where Sig_U is $< I_{U \rightarrow PC} > K_U$, and $C_{U_{sCAs}}$ is the all the subordinate CA certificates of C_U.
3. PC verify the validity of the message. If it is valid, it fills $DS_{U \rightarrow PC}$ and signs Sig_U with K_{PC}. Together with $C_{U_{sCAs}}$, PC can now generate a DToken $DT_{U \rightarrow PC}$.
4. $PC \rightarrow TM$: send $DT_{U \rightarrow PC}$.

The rest of the protocols, namely Second-level Mutual Authentication, Passing and Verifying a DToken, and Reusing a DToken, are the same as that in the gateway case.

3.5 Chained Delegation

For both single-level delegation cases described in the previous section, the delegation can be further extended to a chained delegation, where the delegation depth is greater than one. Fig. 5 illustrates a chained delegation $U \rightarrow G \rightarrow JQS$ through a gateway computer G, where the delegation depth is two. This case extends the single-level delegation scenario presented in Fig. 1 to a remote file system FS, which becomes the final principal of the chain. Instead of only executing a job at JQS, in this scenario a running job at JQS needs to access U's files stored at FS. In addition to the steps presented in Fig. 1, three more steps, illustrated by steps 5, 6, and 7 in Fig. 5, are needed to extend the delegation. These steps are the interactions between computers G, JQS, and FS.

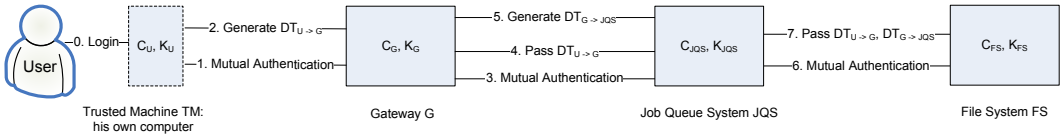


Figure 5: Chained Delegation through a Gatway Computer in a Distributed System

3.5.1 Generating a DToken between the Gateway and the Job Queue System

As illustrated in Fig. 5, before JQS accesses FS for U's files, it needs to obtain a DToken from G so that it is authorized, by G - the delegatee of U, to perform file operations on behalf of U. This extends the original delegation $U \to G$ to $U \to G \to JQS$.

It is worth noting a few settings as a result of steps 3 and 4 in Fig. 5 before moving on to describe steps 5 to 7. As a result of step 3, G has C_{JQS} and all the intermediate CA certificates of C_{JQS}; whilst JQS has C_G and all the intermediate CA certificates of C_G. As the end of step 4, G and JQS both have $DT_{U \to G}$. Here is the protocol describing the interactions of step 5 between G and JQS:

1. $JQS \to G$: request to extend the delegation chain $U \to G$ to $U \to G \to JQS$.
2. G: send $I_{G \to JQS}$, Sig_G, and $C_{G_{sCAs}}$, where $Sig_G = < I_{G \to JQS} >_{K_G}$.
3. $JQS \to G$: JQS verifies the validity of the message. If it is valid, it inserts $DS_{G \to JQS}$ and signs Sig_G with K_{JQS}. Together with $C_{G_{sCAs}}$, JQS can now generate the DToken $DT_{G \to JQS}$.
4. $JQS \to G$: send $DT_{G \to JQS}$.

3.5.2 Passing and Verifying the DTokens

In step 7, two DTokens $DT_{U \to G}$, $DT_{G \to JQS}$ as the proof of the delegation chain $U \to G \to JQS$ are passed along with a file accessing request from JQS to FS. It is also worth noting that the root CA certificate of U, G, and JQS is not passed to FS as part of the DTokens. These root CA certificates are installed by FS itself through trusted means to ensure that it can verify the integrity of C_U, C_G, C_{JQS}, and also the delegation chain without depending on the trustworthiness of G and

JQS. This is an important property differentiating our approach from any of the existing delegation architecture.

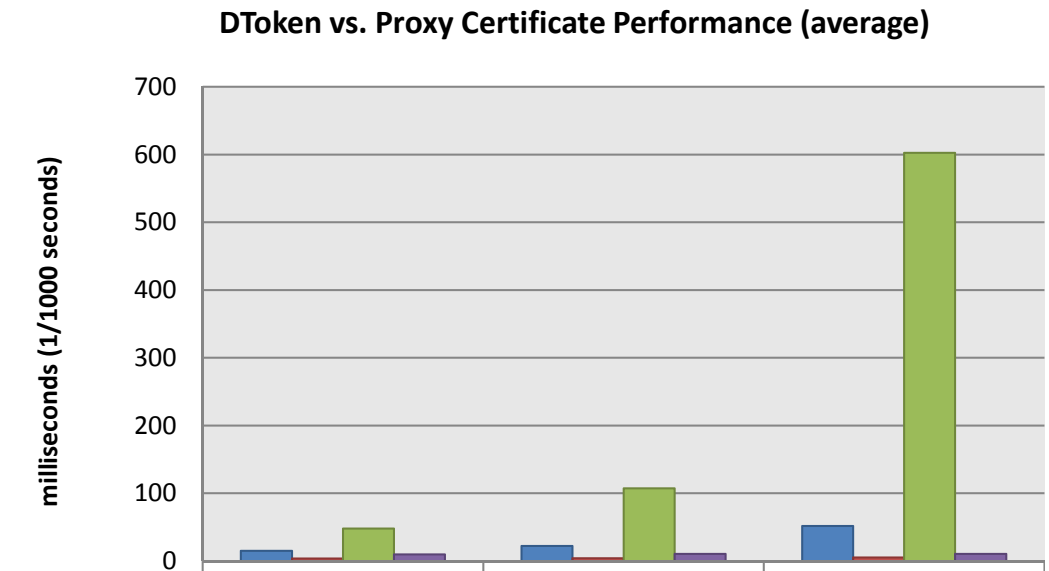
3.6 Evaluation

We have produced a prototype implementation of the DToken solution using the OpenSSL toolkit. The main goal of this prototyping exercise is to investigate the performance implications of creating and verifying DTokens, which are the major operations involved in the architecture. The evaluation focuses on a single-level delegation, in which the size of delegation tokens is trivial (less than 3kb). Therefore, in this case, the communication costs involved in the delegation is clearly not significant. The preliminary evaluation results are encouraging. With typical certificate key sizes, we demonstrate that DToken consistently outperforms the popular de-facto Grid delegation solution - proxy certificate.

Experimental Environment All the experiments are conducted on a Ubuntu OS (8.10) running on top of a virtual box (2.1.4). The host machine is a DELL latitude E6400 laptop with Intel Core 2 Duo Processor (2.8GHz) and 4G memory. The prototype is written entirely in C, is compiled using GCC (4.3.2) and depends solely on the OpenSSL toolkit (0.9.8). The GSI proxy certificate performance data was collected using the command `grid-proxy-init` of the latest globus toolkit (4.2.1). The source program of this command was slightly modified to include performance measuring procedures. All the performance data was collected by repeatedly executing the programs for one hundred times and the average costs of the one-hundred runs are presented here.

Results Fig. 6 illustrates the major performance costs, namely creation and verification of delegation tokens, involved in a single-level delegation scenario. The overall cost of creating a DToken, the major cost of the DToken delegation architecture, is roughly 1/3, 1/5, and 1/10 of that of creating a proxy certificate when the certificate key size is 512, 1024, and 2048 bits, respectively. These results demonstrate that our proposal provides significant performance gain over the proxy certificate approach. On the one hand, because there is no more fresh key pair generation in the DToken architecture, this outcome is well expected. On the other hand, it also shows that our proposal could be less vulnerable to DoS attacks in practice because the cost of creating a delegation token in our architecture is only a fraction of that of generating a proxy certificate.

The results also show that the verification cost involved in both solutions are significantly less than that of the creation cost. For example, when the key size is 2048 bits, the verification cost of a token in DToken is approximately 10% of its



	512 bits	1024 bits	2048 bits
DtokenCreate	15	22	52
DTokenVerify	4	4	5
GSICreate	48	107	602
GSIVerify	10	11	10

Figure 6: Overall Performance Cost Comparsion between DToken and GSI (Proxy Certificate).

creation cost. With the same key size, the verification cost of a proxy certificate in GSI is less than 2% of its creation cost. This suggests that the creation cost of a delegation token in *both solutions* is the major cost. From a performance point of view, the longer the key the less we should be concerned with the cost of verifying a delegation token. Therefore, in the future studies, we could focus on the creation cost of a delegation token, rather than the verification cost.

In addition, the figure indicates that the creation cost of a DToken of key size 2048 bits is comparable to that of a proxy certificate of key size 512 bits (a typical setting in the current Grid deployment environment). From a performance perspective, it suggests that it is much more affordable to employ highly secure cryptography keys (e.g. of key size 2048 bits) in the DToken architecture than in GSI. From a security point of view, the vulnerability introduced by employing the DToken architecture (i.e. all delegation chains share the long-term credentials of a principal) can be lessened by the use of highly secure keys.

3.7 Discussion

DToken is a new type of delegation technology. It is new because it represents a new paradigm in distributed delegation. This section look closely at the differences between DToken and the conventional approaches.

3.7.1 Differences in definition

Our definition of delegation is different from the conventional one adopted in the distributed systems community. In all the existing public key cryptography based delegation work (e.g. [4, 5, 1]), it is the *delegatee who initialises a delegation request*. This is done by the delegatee who generates a delegation token (and a private delegation key) and sends the token to be signed by the delegator. If the delegator authorises the delegation, a signed token is sent back to the delegatee. By using the (signed) token and the key, the delegatee is able to *prove* to others that the delegation is authorised by the delegator. Because the request is initialised from the delegatee, implicitly, the delegatee accepts the the relationship. In our work, the delegation relationship is started by the delegator who approaches the delegatee with a partially generated delegation token addressing to the delegatee. Deciding whether the delegatee should be delegated, what to delegate (e.g. rights and obligations) to the delegatee, how the delegation (e.g. length of delegation, constrains) should be, is the responsibility of the delegator. This is a mimic to the real world delegation scenario where a person at a high point of authority approaches another at a lower point of authority expressing an intention of delegation. If the delegatee accepts the delegation, a complete delegation token is sent back to the delegator.

Our definition of delegation is also more restricted than that of conventional delegation in that both parties need to explicitly agree upon the relationship. Otherwise, neither party will be able to use the delegation token in anyway. For example, the delegator cannot prove that the delegatee has accepted the delegation. Also, nor can the delegatee proven that the delegatee has delegated to it. Once the protocols are complete, others can verify two things with the token: the delegator has authorised the delegation and the delegatee has accepted it. However, in the existing public key cryptography based delegation work, the implicit acceptance from the delegatee is lost in the protocol in the sense that no information about the delegatee can be derived from the delegation token, such as proxy certificate[2] .

[2]It is possible to embed the information of the delegatee in the policy field of a proxy certificate. However, according to the specification of proxy certificate, the policies are added by the delegator, not the delegatee. Therefore, the delegatee can repudiate whether he has initiated the request.

3.7.2 Differences in the role of delegatee

Let us examine what makes DToken a new delegation architecture. In a DToken Grid world, an delegator submits a job to the delegatee with a Dtoken, which can be used by the delegatee to request resources without the intervention from the delegatee. Hence, there is no need for the delegatee to perform a callback to the delegator to request a delegation token. However, this is a must in the conventional approaches, such as proxy certificate. Architecturally, in a proxy certificate Grid world, this means that a *trusted third party*, typically played by a MyProxy server [10], is needed to deal with the callback. This is not a requirement for DToken.

However, it is worth noting that MyProxy also facilitates another role: to support renewal of delegation token. This is important feature for supporting long running jobs. At this point, we haven't decided how the renewal protocols of DToken should work. This remains as part of the future investigation.

3.8 Conclusions and Future Work

This chapter presents the DToken solution, a novel delegation architecture, to address the major well-known limitations of existing delegation solutions. Although some of them are being widely used in practice, the limitations remain open in the literature until now. We have presented the detailed protocols of the DToken architecture and described the preliminary performance evaluation of a prototype implementation. We show that it outperforms significantly than the existing production quality Grid delegation solution - proxy certificate in GSI.

The work presented in this chapter have been implemented within the project. DTokens will be used to support delegation within XtreemOS with the advantages described in this chapter. These advantages will allow XtreemOS to support more efficient and scalable distributed delegation.

Limitations The chapter has focussed on presenting DToken protocols in single level delegation, where the delegation depth is one, and chained delegation where the delegation depth is two. When the delegation chain extends further (delegation depth > 2), additional mechanism is required to ensure the uniqueness of the delegation path from the originating principal and the final principal. In other words, in the current version of DToken, it is possible to construct multiple delegation paths when delegation depth > 2. This will be investigated in the future extension of this work.

The prospect of the technology DToken is a set of protocols describing the delegation relationship between two or more parties. Contrast to existing delegation protocols, it offers an additional property regarding the delegation chain:

non-repudiated traceability. This is an important property for *open distributed systems*, such as Grids, where the system components are composed by geographically dispersed organisations under diverse administrative domains. It can be used as an replacement of proxy certificates. It is worth noting that a presumption of the environment where proxy certificate is used is PKI and we also have this requirement. In theory, it can be used to enhance trustworthiness of the Single Sign On (SSO) technology of XtreemOS [11] which is being developed in parallel in our work package.

4 Interoperability across XtreemOS and gLite platforms

Grid computing deals with interoperability by using standard general-purpose protocols and interfaces. This chapter describes the experience of interoperability between XtreemOS and gLite, a middleware that has been developed in the EGEE project and it is widely used by the European scientific community.

From the point of view of security, we have analysed interoperability along three axes: certificate management, security policies and virtual organisation management. To illustrate interoperability in practice, we have developed a virtual marketplace of computational resources, using resources from XtreemOS and gLite Grids.

The structure of the chapter is the following. Section 4.1 describes the interoperability challenges in certificate management. Then, Section 4.2 analyses interoperability for security policies. Section 4.3 discusses interoperability in VO management. Finally, section 4.4 presents the interoperability case study; including also other challenges such as programability and job management.

4.1 Interoperability Challenges in Certificate Management

Security in both XtreemOS and gLite is based on the concept of *public key infrastructure* (PKI). Each entity user in the PKI environment possesses a public and a corresponding private key. Central to PKI is the concept of *certificates*, which are used to validate the user and the public key associates with the user, and *Certification Authority* (CA), who issue these certificates.

In the case of the certificate management, one important interoperability issue is the structure (type) of a certificate. Most Grid systems, including XtreemOS and gLite, have selected the X.509 format as the standard one. A typical X.509 certificate consists of the following information.

- The version of X.509 that has been used.
- The information about the user or the issuing CA.
- The algorithms used to compute the signature of the certificate.
- The subject whose public key is being certified.
- The validity of the certificate which indicates the time for which the certificate is valid.
- The public key information.

- The signature field which is actually a hash of the above information signed by the CA's private key.

- In addition, there are some optional fields and extensions to customise the certificate.

In relation to XtreemOS and gLite, their X.509 certificates differ in the optional fields and extensions. Structure of the gLite certificate is presented in [22]; gLite certificates follow the standard extensions, including 'CA revocation URL' extensions. Structure of the XtreemOS certificate is presented in deliverable D3.5.9. XtreemOS certificates contain non-standard extensions to carry information about a user's VO attributes.

It is worth mentioning that gLite Grids use *Proxy certificates* to interact directly with a remote service (i.e. to achieve single sign-on and delegation). Proxy certificate consists of a new public/private key pair, signed with the user's personal certificate with a *subject name*. XtreemOS uses *X.509 certificate* with its own extensions. *Proxy certificates* are not used in XtreemOS.

4.1.1 Meeting the Certificate Management Challenges

In order to meet the certificate management challenges, we have developed a *credential conversion service* that acts as a gateway converting a gLite certificate into a XtreemOS certificate. In our case, the gateway validates gLite certificates provided by the UK National Grid Service, converting them into valid XtreemOS certificates. Certain user information is copied from the gLite certificate to the new XtreemOS certificate, including information about the primary VO.

4.2 Interoperability Challenges in Security Policies

There are several challenges that may rise with the interoperability of security policies. We summarise these below.

- *Language issues:* Different Grid systems may have different security policy languages used to control their Grid resources. This means that achieving interoperability with those Grids will require a translation of the policy languages. Different languages have different expressivity levels, so this is not a straightforward to tackle challenge. In the current implementation of XtreemOS VO Policy Service (VOPS), we adopt a XACML 2.0 [16]. In the case of gLite, authorisation at resource level can be achieved with XACML using some of the current available XACML Policy Decision Points.

- *Ontology issues:* The change of domains may also imply changes in their administrative structures. For example, the roles adopted in one VO may

be different from those adopted in others. Even if two VOs had similar roles, it could be the case where the same role is mapped to different sets of permissions in the different VOs. Therefore, the challenge here is to map policy ontologies from one Grid system to another.

- *Enforcement issues:* There are several stages at which security policies can be enforced, such as selection-time, access-time, usage-time and release-time. Interoperating with other middleware systems implies that some notion of *negotiation* of enforcement stages must be reached. Policies written for selection-time enforcement may not be sufficiently strict or relaxed to deal with access-time or usage-time control. Therefore, an agreement must be reached as to when a policy will be enforced in the new domain, and if it is not possible to enforce it at its expected stage, whether it can be changed for other policies at other stages.

In our interoperability case study, both XtreemOS and gLite are using the XACML language for security policies. Therefore, there is not interoperability problems at language level. We will be analysing ontology issues in the next subsection. Let us now discuss enforcement issues. Security policies in XtreemOS are classified into four main categories:

- *User Policies* are specified by the VO users in order to determine the suitability of VO resources to run their jobs. The purpose of these policies is to protect users data and jobs.

- *Resource Policies* are security policies set by the local resource administrators to control access and usage of their resources by the VO users.

- *VO Policies* are specified by the VO owners or managers to determine what is acceptable behaviour in the VO. VO policies are different from user or resource policies in that they provide general rules at the level of the whole VO rather than a specific user or resource belonging to a specific administrative domain.

- *Filtering Policies* are created by matching user and VO policies. These policies are then evaluated at selection time to ensure that the resources selected are suitable.

Traditionally, security policies are enforced at access time according to some model of security [12, 13]. More recently, it has been suggested that policies can also be controlled at runtime to enforce a well-behaved usage of resources [14, 15]. This also includes cases where policies are enforced at the end of the resource usage, i.e. at resource release time.

In XtreemOS, policy enforcement is carried out at two stages: the usual access control stage in which local policies control access to resources from non-local VO users, and the selection stage, where filtering policies are enforced by a special security service called VOPS (VO Policy Service).

User, resource and VO policies of XtreemOS are identical in gLite, but g-Lite has no concept of filtering jobs based on security policies. In gLite, VOs have a configurable acceptable use policy that users sign digitally when they request to join the VO.

4.2.1 Meeting the Challenges in Security Policies

The VOPS service in XtreemOS provides a limited notion of interoperability *within* the VO itself, in the sense that it can manage VO policies belonging to different heterogeneous domains. However, a VOPS runs in the scope of a single XtreemOS-based Grid. Therefore, when VOs across different Grids are involved, a single VOPS cannot deal with the heterogeneity of policies across those Grids.

In order to meet the security policy challenges across different Grids, VOPS will need to have an interface which allow other systems to manage their non-XtreemOS-compliant security policies. Similarly, if XtreemOS-compliant policies are to be sent to other systems, these will need to be translated so as to enforce them in the new Grid domains. These functionalities are areas of future research and development for the VOPS service.

4.3 Interoperability Challenges in VO Management

A Grid has introduced the concept of Virtual Organisations (VOs). In a VO, different individuals, enterprises, organisations come together to share resources and services under a set of rules or policies guiding and governing the extent and conditions of sharing. Our discussion of interoperability will cover two main issues: VO membership management and information security in VO management.

In relation to VO membership management, gLite has pioneered with the creation of the Virtual Organisation Membership Service (VOMS). VOMS is an authorisation system developed for the European Data Grid as part of the DataGrid and DataTag projects. In a VOMS system, a user may be a member of as many VOs as possible, and each VO can be a complex structure with groups and subgroups in order to clearly divide its users according to their tasks. A user, both at VO and group level, may be characterised by any number of roles and capabilities. VOMS uses short lived credential, which includes user information, server information, and the validity period. Later versions of VOMS include the use of Security Assertions Markup Language (SAML) tokens.

XtreemOS has developed its own VO membership management service, which is part of the XtreemOS VO Management Subsystem (X-VOMS). It is inspired in VOMS in maintaining the structure of a VO – VO, groups, roles –, but it does not include capabilities. It also include information about local sites participating in the Grid; these sites in XtreemOS act as sub-ordinate certification authorities, authorising the participation of local resources in a VO (the XtreemOS RCA component). More information about X-VOMS is available in deliverables D3.5.11.

To deal with interoperability in VO membership management, we need to associate/map the structure of a VOMS VO with the structure of a X-VOMS VO (group, sub-groups, roles). This can be achieved by using ontologies. We do not get into details on the use of ontologies for that, and refer the reader to S-OGSA [23], a reference architecture for semantic Grids explaining how this kind of mappings can be realised.

In relation to information security of VO management, the Grid security model has been called Grid Security Infrastructure (GSI), which enables secure authentication and communication over an open network. GSI is based on public key encryption, X.509 certificates, and the Secure Sockets Layer (SSL) communication protocol. Both gLite and XtreemOS follow GSI, however there are some differences, as described below.

- *Use of Proxy Certificates*. gLite use proxy certificates for achieving single sign-on and delegation. Proxy credentials are derived from X.509 end entity certificates, and signed by corresponding end entity private key, to provide restricted proxy and delegation. By contrast, as stated in section 3, XtreemOS does not need to use proxy certificates for achieving single sign-on and delegation.

- *Authorisation at Resource Level*. In gLite, the authorisation of a user on a specific Grid resource relies on the Virtual Organisation Membership Service (VOMS) and a combination of the Local Central Authorization Service (LCAS) and the Local Credential Mapping Service (LCMAPS). LCAS is able to make authorisation decision based on request resource, the identity of the requestor, the VOMS credential, and the proxy certificate. Once the access control or local authorisation decision has been made, the Grid credentials are mapped to the local fabric or operating system through LCMAPS. On the other hand, XtreemOS resource-authorisation relies on X-VOMS; but as explained above the enforcement decision can be done a resource-selection time using the VOPS service. XtreemOS is focusing on fine-grained authorisation, and exploit virtualisation technologies (Linux containers) to achieve isolation (as described in deliverable D3.5.9. Once a resource is selected and accepts to run a job, it is created a container to run

the job, where the container is configured with information from the job description and certificate (identity of the requestor, X-VOMS attributes such as VO, group, etc., and usage restrictions such as maximum CPU capacity, storage and networking access).

These differences can be seen as low-level differences at the infrastructure level. For the end user, or application developer, they are transparent.

4.3.1 Meeting the Security Challenges in VO Management

The current implementation of XtreemOS X-VOMS does not interoperate with EGEE VOMS. To achieve such interoperation would require the existence of mechanisms such as SAGA bindings to deal with both components. However, we should mention that X-VOMS includes additional functionalities not present in VOMS, such as the management of resource certification authorities (RCAs) at the different administrative domains.

4.4 An Interoperability Case Study: A Virtual Marketplace of Computational Resources

In order to illustrate the interoperability XtreemOS-gLite, we have developed a system that facilitates the commercialization of Grid resources on-demand through a virtual marketplace of computational resources, where a seller is capable of listing the Grid resources, and buyer can ask/bid dynamically for required computing resources for their applications. Our system assumes that resources are available in a XtreemOS Grid and in a gLite Grid.

The system provides a computational resource auctioning system built upon a dynamic bid matching algorithm tailored specifically for the trading of computing power. It helps both consumers and providers of computational resources to use the resources efficiently so as to maximize the economical benefits and minimize the idle time for them. The system is developed to integrate into a single framework three key features:

- *Interoperability* is achieved by using a standard programmable interface, the Simple API for Grid Application (SAGA), to bridge the gap between existing grid middleware and application level needs. The same system could run on XtreemOS and/or g-Lite Grids, or interoperate on Grids using resources from both platforms.

- *Cost saving for end users* is guaranteed by allocating the applications to the more economical resource(s), following policies defined by the end users.

- *Dynamic scheduling* is achieved through the virtual marketplace, which implements scheduling and trading algorithms that allocates applications following classical performance parameter as well as the cost of resource usage.

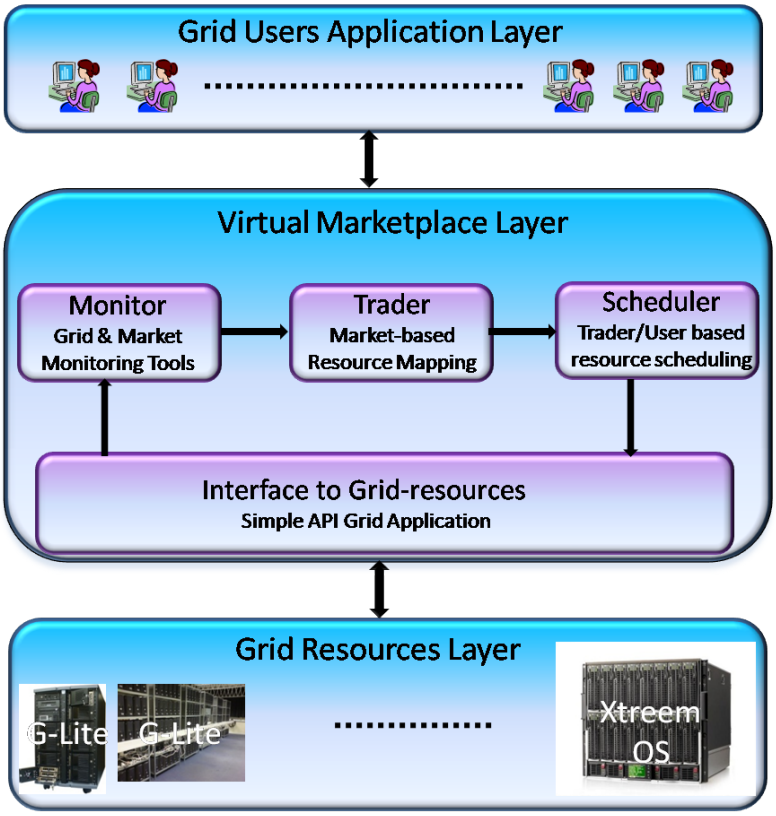


Figure 7: Architecture to provide the interoperability.

4.4.1 Abstract View of the Virtual Marketplace

Figure 7 illustrates a logical layered architecture of the virtual marketplace, representing the entities and their dependency to other entities. The flow of information or control is depicted by arrows. An arrow from an entity X to an entity Y means that X sends information to Y or passes control to Y. Our system offers the mechanisms for deploying and executing the application (e.g. automatic deployment, execution monitoring, and hardware resource discovery) for the business processes to purchase the resources on the Grid. Implementing such a business model requires at least the following basic roles, which belong to three layers: the

Grid users application layer, the Virtual marketplace layer, and the Grid resources layer.

Grid users application layer: This layer allows end-users (scientist, chemist, physician....) to submit applications to the deployed resources. We consider Grid application to be a collection of work items to solve a certain problem or to achieve desired results using the Grid infrastructure. Grid applications can be scientific, mathematical, academic problems or the simulation of business scenarios, like stock market development, that require a large amount of data as well as a high demand for computing resources in order to calculate and handle the large number of variables and their effects.

Virtual marketplace layer: This layer implements monitoring, trading and scheduling services so as to utilize the available Grid resources efficiently and exploit the benefits of the interoperability and scalability of the Grid platform. This layer consists of four main components:

- *Monitor* implements the monitoring/reporting techniques, which monitors resources and reports changes such as dynamic re-allocation of resources, according to changes generated from evolution in the resource market,execution status of submitted applications, etc. Monitoring is achieved by either direct or indirect capture of resource status and pre-defined events. The indirect interface uses logs generated at run-time by the Grid infrastructure. The direct interface is a portal collecting dynamically events generated by monitoring services associated to the Grid infrastructure.

- *Trader* implements the trading algorithms that depends on criteria such as cost, processing power, execution time or resource availability. It is also responsible for sending notifications to users about the status of their request. For example, inform a bidder whether the bid is winning or not.

- *Scheduler* schedule the application on to the selected Grid resource. Scheduling of the end users application is done on to the selected resource by following the analysis provided by the trader.

- *Interface to Grid resources* provides simple access for distributed systems and abstractions for applications and thereby address the fundamental application design objectives of interoperability across different infrastructure. It also supports job submission and data management (efficient data access, data replication, streaming of data, etc.).

Grid resources layer: This layer consist of the resources (server, storage and network) used to execute the end users applications. The submitted application is executed on the selected Grid resources and result is sent back.

4.4.2 Achieving Inteoperability

Our implementation leverages on existing technologies such as trading algorithms [19, 20], grid middlewares/OS [18, 21] and API [3] for Grid applications [17].

The following solutions have been applied in order to deal with the interoperability issues mentioned above.

- In order to tackle with the differences between XtreemOS certificate and gLite certificates, we have used a credential conversion service translating gLite certificates issued by the UK National Grid Service into XtreemOS certificates.
- At the moment, we are not exploiting security policies. However, any required security policy will be expressed using the XACML policy language, such that can be applied in any of the underlying Grid platforms.
- Our system requires that both XtreemOS and gLite Grids to be created separated previous to the creation of the system. In gLite, we are using a VO created by the National Grid Service, selecting from a pool of resources offered by the Grid. In XtreemOS, a VO is created using the VOLife Web front end.

There are other two non-security issues affecting interoperability. One issue is the language employed for describing jobs. gLite uses the Job Description Language (JDL) to describe jobs to be executed in VO resources, while XtreemOS uses the standard Job Submission Description Language (JSDL). Fortunately, a translator [4] JSDL to JDL exists so that job can be submitted to any resource irrespective of its format. Another issue is related to access the Grid in a programmatic way. Fortunately, both gLite and XtreemOS use the Simple API for Grid Applications (SAGA) to access the Grid.

4.4.3 Economically efficient computing

We discuss an abstract business model *economically efficient computing*, where the user owns the software that will be executed on the Grid and has the option to specify the time, cost or hardware resources on which the software should run. Unit of trade is defined as the number of resources required at any time of the interval for the fixed amount of time to execute an application. For example, 4 reasources over 3 hours from 09:00 to 12:00 to execute application1.

Format of the bid/bsk: A bid describes the resources required by the buyer. The resources requirement is specified as: (1) The type of resource (software,

[3] http://forge.gridforum.org/projects/saga-rg/

[4] http://grid.pd.infn.it/omii/jsdl2jdl

processing power, free space, etc.), (2) The number of resources, (3) The start time of the interval for using the resources, (4) The time duration of using the resources, (5) The price expressed in £for use of one resource/min, and (6) The expiration time of the request. If the time limit is reached without the bid being matched, the bid is removed. For example, User A bids for 4 XtreemOS resources to be used for 3 hrs, starting at time 8:00, with bid price £0.5, and time limit 18:00.

An ask describes the resources offered, which are specified as: (1) The type of resources (software, processing power, free space, etc.),(2) The number of resources, (3) The start time and the end time of the interval when the resources are available, (4) The price expressed in £for use of one resource/min, and (5) The expiration time of the offer. If the time limit is reached without the ask being matched, the ask is removed. For example, User B asks for 4 XtreemOS resoruces to be used for 3 hrs, starting at time 18:00, with bid price £0.5, and time limit 21:00.

Trading is performed by means of an auction mechanism. The submitted bids and asks are placed in the bid queue and the ask queue respectively. Each queue is ordered according to the price and time of submission. The bid queue is sorted in decreasing order of price, and the ask queue is sorted in increasing order of price. If two or more orders at the same price appear in a allocated queue, then they are entered by time with older orders placed above the newer orders. An bid/ask remains in the queue until it is allocated, removed due to its expiration time or removed by the submitted user.

The matching algorithm defines how a bid/ask is matched by a set of asks/bids. The matching algorithm initially computes the candidate matches to an ask by means of creating a matrix. Each column of the matrix corresponds to a time slot (i.e. the time interval in which service can be provided). Each row corresponds to a provider that can offer service now, with the cheapest being on the top row. A cell of the matrix is marked if the provider can offer computing resources during this specific time slot. It is the responsibility of the matching module to be invoked periodically, in order to compute matches and remove expired bids and asks from the bid/ask queue. The results of the matching procedure are subsequently passed to the scheduler and the accounting system of the market place.

4.5 Final Remarks

This chapter has studied interoperability issues between XtreemOS and the gLite middleware. The use of standard protocols and interfaces, such as X.509 certificates, XACML policies or APIs such as SAGA, has helped in achieving such interoperability. However, there is additional work required in order to reach a full interoperation between XtreemOS and other Grid middleware.

We have also introduce an example of interoperability through a virtual marketplace of computational resources. Our example answers questions such as "which Grid should be used that will minimize cost along with achieving efficient applications' execution time?", "how end-user can select Grid resources according to pre-defined policies, including cost policies?" and "how to achieve interoperability when using gLite and XtreemOS platforms?". Our trading system provides a portal for end users to avail the computing power of Grid resources, depending on economical and performance parameters.

5 Conclusion and Future Work

In this deliverable, we presented the solutions developed in XtreemOS for achieving single sign-on and traceable delegation. The proposed single sign-on and delegation solutions are pure *operating system* solutions. In classical operating systems, some credentials are bound to users when they log in, UID/GID for instance in Unix systems. All processes running of behalf of the user have these credentials. XtreemOS guarantees the binding of user credentials to all grid requests sent by the user to XtreemOS services.

The deliverable also discusses interoperability across XtreemOS and gLite platforms. The deliverable analysed interoperability in relation to certificate management, use of security policies, and virtual organisation management. To analyse interoperability, the deliverable report the experiences in developing a virtual marketplace of computational resources from XtreemOS and gLite Grids.

Future research directions will be focusing on tackling the identified limitations in the DToken approach. The interoperability case study, the virtual marketplace of resources, is a Grid application, but we are interested in adapting it for the case of having Cloud resources.

References

[1] S. Tuecke and V. Welch and D. Engert and L. Pearlman and M. Thompson. *RFC 3820 - Internet X.509 public key infrastructure (PKI) proxy certificate profile*, June, 2004.

[2] Von Welch and Ian Foster and Carl Kesselman and Olle Mulmo and Laura Pearlman and Jarek Gawor and Sam Meder and Frank Siebenlist. *X.509 proxy certificates for dynamic delegation*, in the 3rd Annual PKI R&D Workshop, 2004.

[3] I. Foster and C. Kesselman and Gene Tsudik and Steven Tuecke. *A security architecture for computation Grids*, in Conference on Computer and Communication Security, vol. 15, no. 3, 2001.

[4] Morrie Gasser and Ellen McDermott. *An architecture for practical delegation in a distributed system* in IEEE Symposium on Research in Security and Privacy, pp. 20-30, May 1990.

[5] B. Clifford Neuman. *Proxy-based authorization and accounting for distributed systems*, in Proc. of the 13th International Conference on Distributed Computing Systems, May 1993.

[6] Philippa J. Broadfoot and Gavin Lowe. *Architectures for secure delegation within Grids*, Technical Report PGR-RR-03-19, Oxford University Computing Laboratory, 2003.

[7] Morrie Gasser and Andy Goldstein and Charlie Kaufman and Butler Lampson. *The digital distributed system security architecture*, in the Proc. of National Computer Security Conference, 1989.

[8] G. Zorn. *RFC 2759 - Microsoft PPP CHAP extensions, version 2*, Jan. 2000.

[9] D. Dolev and A.C. Yao. *On the security of public key protocols*, in Proc. of the IEEE 22nd Annual Symposium on Foundations of Computer Science, pp. 350-357, 1981.

[10] J. Basney, M. Humphrey, and V. Welch. *The MyProxy Online Credential Repository*, in Software: Practice and Experience, Volume 35, Issue 9, July 2005, pages 801-816.

[11] Yvon Jégou. *Credential Management and Security Architecture in XtreemOS: Proposition for Single-Sign-On and Delegation*, XtreemOS Technical Report # 7, April 2009.

[12] Ravi S. Sandhu. *Lattice-based Access Control Models*, in IEEE Computer Journal, volume 26, issue 11, Nov. 1993, pages 9-19.

[13] Ravi S. Sandhu and Edward J. Coyne and Hal L. Feinstein and Charles E. Youman. *Role-based Access Control Models*, in IEEE Computer Journal, volume 29, issue 2, Feb. 1996, pages 38-47.

[14] J. Park and R.S. Sandhu. *The UCON$_{\text{ABC}}$ Usage Control Model*, in ACM Transactions on Information and System Security, volume 7, issue 1, Feb. 2004, pages 128-174.

[15] R. S. Sandhu and J. Park. *Usage Control: A Vision for Next Generation Access Control*, in MMM-ACNS, 2003, pages 17-31.

[16] OASIS. *OASIS eXtensible Access Control Markup Language (XACML) TC*, 2005.

[17] S. Jha and H. Kaiser and A. Merzky and O. Weidner. *Grid Interoperability at the application level using SAGA*, in IEEE International Conference on e-Science and Grid Computing, pages 584-591, 2007.

[18] Christine Morine. *XtreemOS: A Grid Operating System Making your Computer Ready for Participating in Virtual Organizations*, in ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, pages 393-402.

[19] A. Kretsis and P. Kokkinos and E. Varvarigos. *Developing Scheduling Policies in gLite Middleware*, in CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pages 20-27.

[20] Costas Courcoubetis and Manos Dramitinos and Thierry Rayna and Sergios Soursos and George D. Stamoulis *Market Mechanisms for Trading Grid Resources*, in book GECON, year 2008, pages 58–72.

[21] gLite Team *http://glite.web.cern.ch/glite/documentation/default.asp*

[22] D. L. Groep et al. *Grid Certificate Profile*. OGF Document, 2007.

[23] O. Corcho et al. *An Overview of S-OGSA: A Reference Semantic Grid Architecture*, in Web Semantics: Science, Services and Agents on the World Wide Web, Volume 4, Issue 2, Semantic Grid –The Convergence of Technologies, June 2006, Pages 102–115.