Project no. IST-033576

XtreemOS

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

Fourth Specification, Design and Architecture of the Security and VO Management Services
D3.5.13

Due date of deliverable: November 30^{st}, 2009
Actual submission date: December 9^{th}, 2009

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

Type: Deliverable
WP number: WP3.5
Task number: T3.5.2

Responsible institution: STFC
Editor & and editor's address: Benjamin Aziz
Rutherford Appleton Laboratory
Science and Technology Facilities Council
Harwell Science and Innovation Campus
Didcot OX11 0QX
United Kingdom

Version 1.0 / Last edited by Benjamin Aziz / December 4^{th}, 2009

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
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	20/08/09	Benjamin Aziz	STFC	Structure of the deliverable defined
0.1	02/10/09	Ian Johnson	STFC	Expansion of design and other comments
0.2	12/10/09	Benjamin Aziz	STFC	Added the sections on VO management and security capabilities
0.3	19/10/09	Benjamin Aziz	STFC	Added the sections on description of VO management and security services
0.4	23/10/09	Primož Hadalin	XLAB	Added the Monitoring and Auditing Capabilities
0.5	29/10/09	Benjamin Aziz	STFC	Added the Section on Trust Model
0.6	10/11/09	Matej Artač	XLAB	Added the RCA sequence diagrams and class diagrams.
0.7	12/11/09	Aleš Černivec	XLAB	Added the details in the VOPS sections.
0.8	12/11/09	Ian Johnson	STFC	Expanded description of use cases for Grid creation and Grid population
0.9	12/11/09	Benjamin Aziz	STFC	Added the Class Diagrams for XVOMS and CDA
0.9	19/11/09	Ronald Fowler	STFC	Added description of XVOMS API
0.9	19/11/09	Ian Johnson	STFC	Expanded use cases for VO termination and user removal. Added description of CDA API
0.9	19/11/09	Chengchun Shu	ICT	Added description of Web Front-end API
0.9	20/11/09	Benjamin Aziz	STFC	Added the Executive Summary and the Introduction sections.
0.9	20/11/09	Benjamin Aziz	STFC	Added the Conclusion.
0.9	20/11/09	Alvaro Arenas	STFC	Revision before internal review.
1.0	20/11/09	Benjamin Aziz	STFC	Generating 1st version for internal review.
1.0	04/12/09	Benjamin Aziz	STFC	Consolidating improvements after internal review.
1.0	08/12/09	Alvaro Arenas	STFC	Final revision before submission.

Reviewers:
Thilo Kielmann (VUA), Santiago Prieto (TID)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.5.2	Specification, design and architecture of XtreemOS security services	STFC*,INRIA,SAP,ULM,XLAB,ICT
T3.5.3	Security policy management and enforcement	XLAB*,STFC,SAP
T3.5.8	VO Lifecycle Management Systems	STFC*,ICT,XLAB
T3.5.9	Security of Grid Level Services	STFC*,ULM,INRIA,SAP,XLAB
T3.5.12	Monitoring and Auditing Services	XLAB*,STFC,SAP

°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
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
PKI	Public Key Infrastructure
RCA	Resource Certification Authority
VOPS	Virtual Organization Policy Service
VOLife	Virtual Organization Lifecycle service
XtreemFS	XtreemOS File System
XVOMS	XtreemOS Virtual Organization Management Service

List of Figures

Executive Summary

This deliverable aims at providing a complete and clear reference on the security and virtual organisation management services in XtreemOS. These services include the XtreemOS Virtual Organisation Service (XVOMS), the Resource Certification Authority (RCA), the Virtual Organisation Policy Service (VOPS), the Monitoring Service and the Auditing Service.

The focus of the deliverable has been centred on describing these services from three main perspectives; each targeting a different layer of system design and architecture:

- *High-level capabilities*: these are high-level depictions of the possible actions that users of the XtreemOS system can perform using the security and virtual organisation management services. Capabilities simply state what goals can and cannot be achieved by means of the various use cases that the services enable the users to carry out.

- *Services design*: these provides intermediate-level view of the services in a manner capturing the logical components of each service by means of class diagrams. The design also provides sequence diagrams to express interactions among the different classes and their users.

- *Services interfaces*: these provide a low-level view of the services realised through concrete application programming interfaces.

Apart from the above novelty in the presentation of the security and virtual organisation management services, the current deliverable also introduces two main new features compared to the previous specifications:

- Termination capabilities for virtual organisations, which describe how virtual organisations can be terminated and their state be dissolved.

- Monitoring and auditing, which allow for the monitoring of resources for changes in their attributes such that auditing and non-repudiation features can be supproted by XtreemOS applications.

1 Introduction

This deliverable defines the final specification of the security and Virtual Organisation (VO) management services. The deliverable has been written keeping in mind readers who are not necessarily experts in XtreemOS as well as the XtreemOS expert developers and end users. It also aims at giving as precise as possible an idea of what will be available for the audience at the end of the project regarding the security and VO management services and the capabilities enabled by these services. As a result, the deliverable was designed to provide a high-level view of the services in the form of capabilities and use cases, which state which actions can and which cannot be done using the services, an intermediate-level view of the design of the services useful for system engineers and low-level application programming interfaces useful for system developers and programmers.

1.1 A Brief History of the Deliverable

This deliverable builds on the material that was presented in previous versions of the specification of XtreemOS security and VO management services.

- *D3.5.3*: This deliverable introduced the first version of the specification of the security and VO management services in XtreemOS [1]. This specification was based on the requirements arising from the case studies in XtreemOS as well as the constraints of the underlying technology of Linux, and it included the early design of the services as well use cases of how the services can be used to achieve the requirements. The deliverable was released in May 2007.

- *D3.5.4*: This was the second version of the services specification and it was released in December 2007 [2]. The deliverable focused on demonstrating how the security and VO management services could be used to fulfil the security requirements of the application execution management and the XtreemFS file system services in XtreemOS.

- *D3.5.11*: The main focus of this deliverable was on the definition of the XtreemOS trust model, and how the security and VO management services contribute to establishing and managing this trust model. The deliverable was released in January 2009 [3].

In its essence, the current deliverable constitutes a single standalone document, which encompasses the research and development effort presented in all the above previous versions.

1.2 New Features

The current deliverable introduces mainly two new features for the security and VO management services:

- Termination capabilities for VOs: this is a new feature of the services, which deals with the termination of VOs and the dissolution of VO state. Although previous deliverables had already defined interfaces for performing certain termination-related actions, such as the removal of users, there was no overall solution as to how to tackle the issue of VO terminations.

- Monitoring and Auditing: This is a new feature as well, which aims at monitoring resources for changes in their attributes. Such capability can then be used to carry out auditing capabilities for purposes of, for example, billing and non-repudiation.

1.3 Structure of the D3.5.13

The structure of this deliverable is as follows. In Section 2, we describe briefly the trust domains, actors and the logical structure of the various security and VO management services in XtreemOS. In Section 3, we give a full description of the security and VO management capabilities in XtreemOS, which include the different use cases involving the actors and the services. In Section 4, we give a brief description of the XtreemOS trust model. In Section 5, we outline the detailed design of the XtreemOS security and VO management services. In Section A, we provide a detail description of the security and VO management services API. Finally, in Section 6, we conclude the deliverable.

1.4 How the Deliverable Meets WP3.5 Tasks

The deliverable represents the output frontend to research and development effort in several tasks in WP3.5. The following represents the mapping of the tasks to the corresponding sections in the deliverable:

- **T3.5.2: Specification, design and architecture of XtreemOS security services.** This is the task coordinating the architectural work in WP3.5 and guiding the production of this deliverable.

- **T3.5.3: Security policy management and enforcement.** This task contributed to all issues related to policy management reported in this deliverable, in particular Section 4 and Subsections 5.4 and 6.4.

- **T3.5.8: VO Lifecycle Management Systems.** This task contributed to all issues related to VO and credential management reported in this deliverable, in particular Section 4 and Subsections 5.1, 5.2, 5.3, 6.1, 6.2, 6.3 and 6.7.
- **T3.5.9: Security of Grid Level Services.** This tasks aligns the security work in WP3.5 with the work in other work packages in SP3. Main outputs contributed to Section 3.
- **T3.5.12: Monitoring and Auditing Services.** This task contributed to all issues related to the monitoring and auditing of security events reported in this deliverable, in particular Subsections 5.5, 5.6, 6.5 and 6.6.

2 The Structure of the Security and VO Management Services

This section gives an overview of the general structure of the security and Virtual Organisation (VO) management services in XtreemOS, including a description of the individual services and the actors involved in the interactions with these services.

2.1 Trust Domains

The Security and VO Management services in XtreemOS are based on three main trust domains, as shown in Figure 1.

Figure 1: The Security and VO Management Trust Domains.

These domains are described as follows:

- The Resource Site domain: This includes sites that offer resources to the Grid and any VOs formed out of the Grid.
- The User Site domain: This includes sites that provide users of VOs who will submit jobs to the resources included in those VOs.
- The Core Site domain: This represents the core site in which the Security and VO Management (and possibly other XtreemOS) services may be running. From the trust point of view, the Core Site represents the root of trust for both the Resource and User Sites.

2.2 Actors

Having defined the main trust domains in the previous section, we now introduce the main actors of the Security and VO Management services.

- The User: this actor is the user of VOs, who is also registered in the Grid within which the VOs are created.
- The VO Administrator: this actor is a previous User who created a VO and became the owner and administrator of that VO. Therefore, the VO Administrator has full authority on managing the VO.
- Resource Administrator: this is the actor owning the resources offered to VOs. The actor could be either a whole site administrator or the owner of a single machine belonging to his site.
- The Grid Administrator: this actor is responsible for managing the core XtreemOS security and VO management services.

These actors are shown in Figure 2.

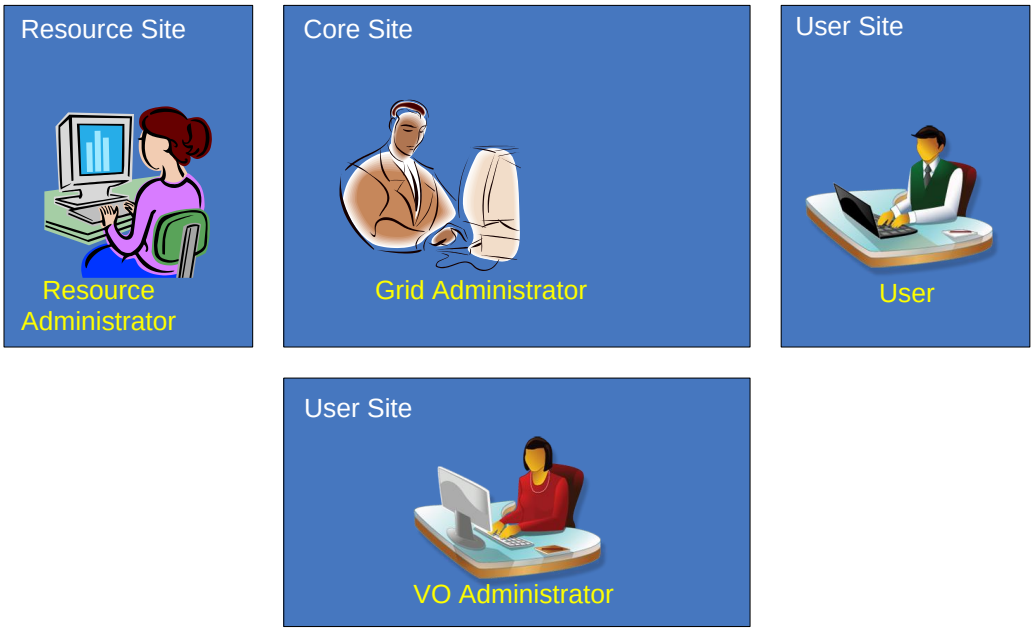


Figure 2: The Security and VO Management Actors.

2.3 The Core Security and VO Management Services

The core XtreemOS security and VO Management services are shown in Figure 3 These services consist of the XtreemOS VO Management Service (XVOMS), the Resource Certification Authority (RCA), the VO Policy Service (VOPS) and the

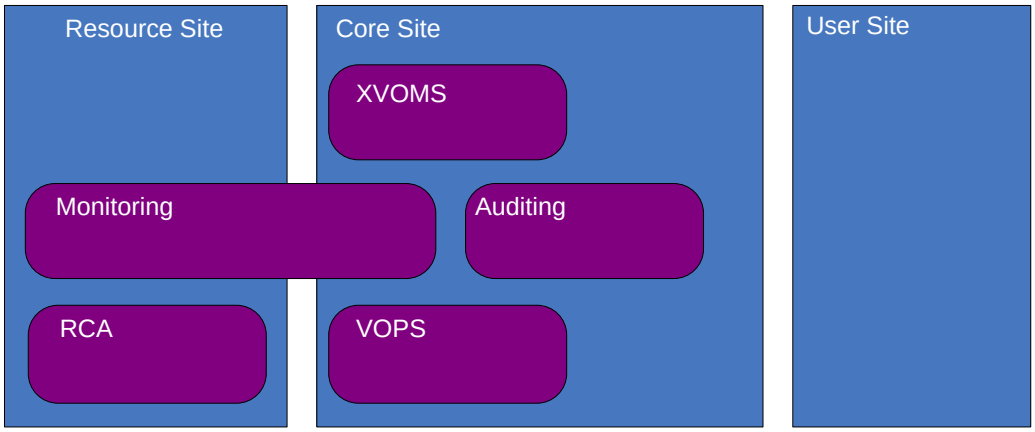


Figure 3: The Security and VO Management Services.

Monitoring and Auditing services. This last service is one of the new features of this specification. In the following sections, we give a brief overview of each of these services.

2.3.1 XVOMS

The XVOMS (XtreemOS Virtual Organisation Service) is a VO and trust management service whose architecture is illustrated in Figure 4. The XVOMS service

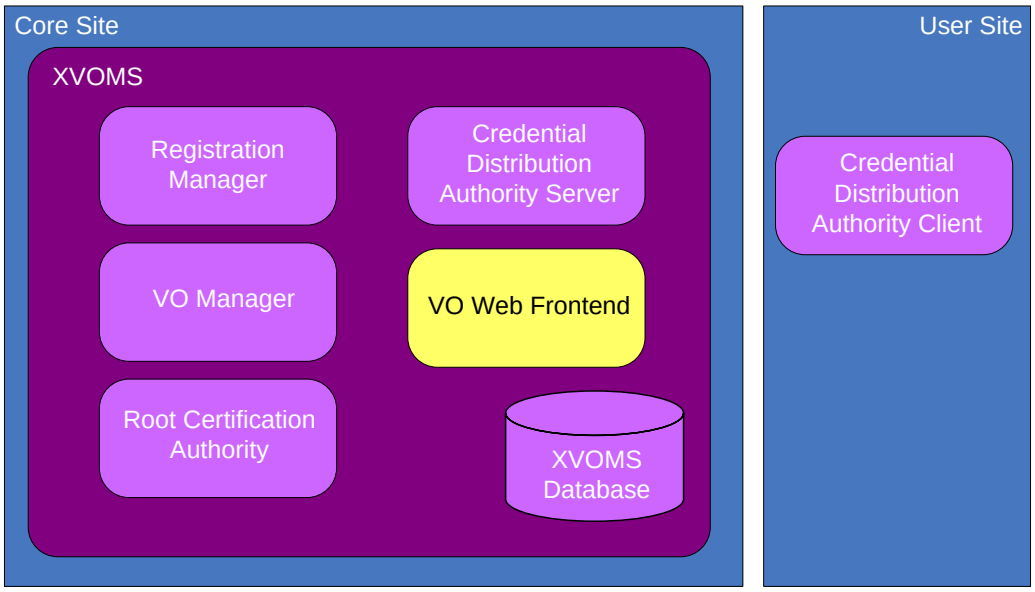


Figure 4: The XtreemOS VO Management Service (XVOMS).

consists of the following components:

The Root Certification Authority. This is a manual service that creates the XtreemOS trust anchor, the root certificate, and uses it to certify the identity of core services within XtreemOS. This service can be performed offline to avoid compromise of the root private key. The certification of core services can optionally be performed by the CDA service, described next.

The Credential Distribution Authority Server. This component, also referred to as the CDA Server, is responsible for distributing XtreemOS identity certificates ("XOS-Certs") to users. The XOS-Cert, an X.509 v3 public key certificate with XtreemOS extensions, is defined in D3.5.5. The CDA may optionally be configured to also provide service certificates, certifying the identity of XtreemOS core services.

The Credential Distribution Authority Client. This is a client-side program that interfaces with the CDA Server, in the case when it is not possible to use the Web-based VO Web Frontend interface.

The Registration Manager. This component is responsible for managing the initial registration of users and RCAs with the XtreemOS system.

The VO Manager. This component controls the lifecycle of the VO.

The XVOMS Database. This is the main database in XVOMS in which all the information regarding the user and RCA registrations, VO membership and lifecycle is stored.

The VO Web Frontend. This is a Web-based interface to the functionality offered by XVOMS.

2.3.2 RCA

The Resource Certification Authority (RCA) is a certification authority at the level of administrative sites offering resources to XtreemOS VOs. The RCA consists of the following components, as shown in Figure 5. The RCA is responsible for bootstrapping trust in the individual resource domains. This trust is used by other XtreemOS components such as the Application Execution Management (AEM) component to be able to submit jobs to resources.

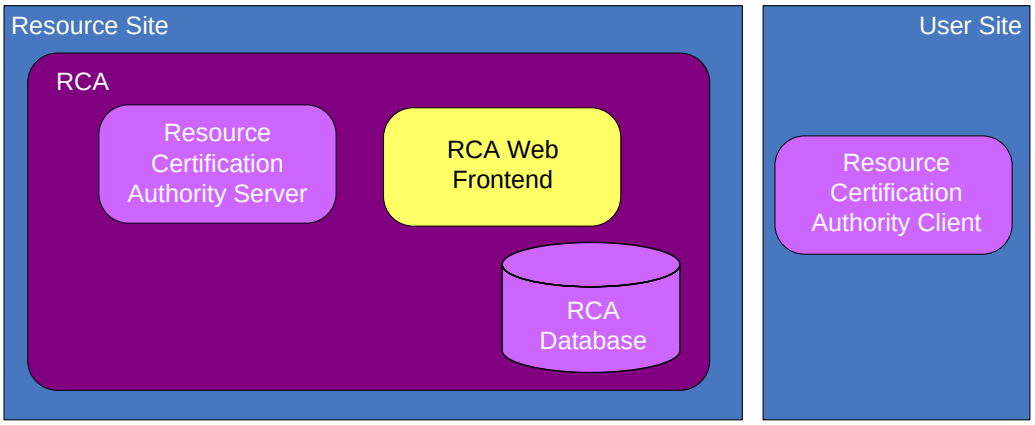


Figure 5: The Resource Certification Authority (RCA).

The following sections give an overview of the main logical components the RCA is composed from.

The RCA Server. This is the main component, which provides the functionality of the RCA. The server is responsible for issuing certificates to resources.

The RCA Client. The RCA Client is a DIXI-based client-side program that can interact with the RCA. It can be installed on individual nodes and be used to ease the interactions with the RCA server.

The RCA Web Frontend. The RCA Web Frontend is an alternative, Web-based interface to the RCA Server instead of the RCA Client.

The RCA Database. The RCA Database stores the state of resources in each administrative domain. This state could indicate that a resource is unregistered with the Grid, registered with the Grid and if so, whether it is currently offered to any VOs in the Grid. The main interface to the RCA Database is through the RCA Server functionality.

2.3.3 VOPS

The VOPS (Virtual Organisation Policy Service) is used to manage and enforce VO policies. The service consists of the following components, as shown in Figure 6.

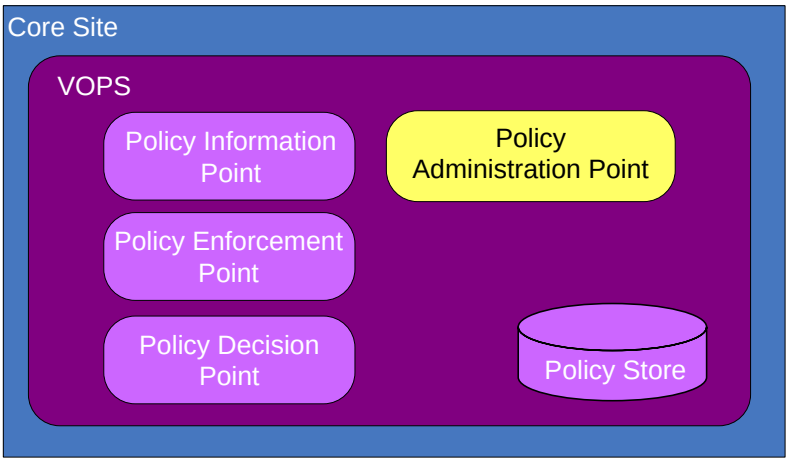


Figure 6: The VO Policy Service (VOPS).

The Policy Enforcement Point. The Policy Enforcement Point (PEP) is where the users' requests are intercepted in order for these requests to be checked and appropriate decisions enforced on the requests. The users' requests may carry user credentials regarding their attributes and the attributes of the context.

The Policy Decision Point. The Policy Decision Point (PDP) is the component which enforces the security policies on user requests. The PDP contains the logic that is computed against the policies and the users requests.

The Policy Information Point. The Policy Information Point (PIP) is a component of VOPS which queries information about the request arriving from a user, additional user credentials and information about the context of the request and the system.

The Policy Administration Point. The Policy Administration Point (PAP) allows the site or resource administrator to add, delete and update policies in the policy store.

The Policy Store. The Policy Store (PS) is a database containing all the policies related to the different resources.

2.3.4 Monitoring and Auditing

Monitoring and auditing are responsible for receiving status, changes and events, providing feedback on user behaviour and resource performance to interested par-

ties, as well as storing metrics and events to historical database. The Monitoring described in this document and used for the security infrastructure purposes is a super-set of the monitoring in the AEM. This means that while it leverages functionality and metrics of the AEM's monitoring, it also includes other services' events and metrics. As a result, the monitored information spans a wider spectrum of information, which can be used for assessing the security status of the distributed system and analysing past behaviour of both the system and the actors. Another important aspect of the security auditing is the ability to record specific actions, non-refutably proving potential accountability for harmful behaviour.

Figure 7 shows the abstract view of the monitoring and auditing system and the domains it spans.

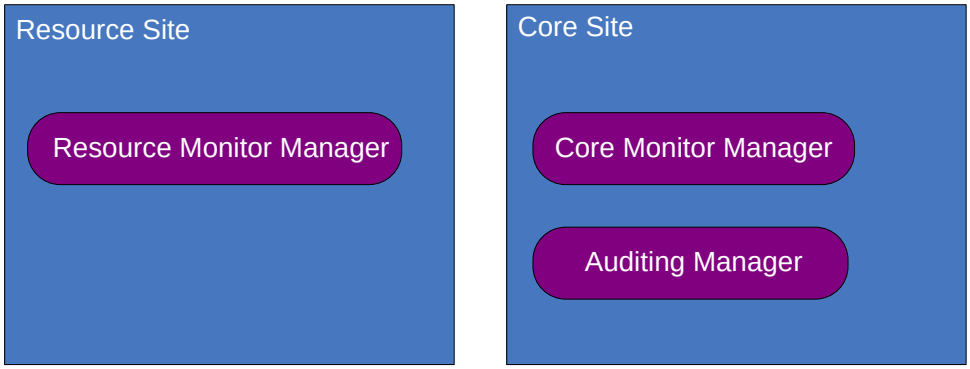


Figure 7: Monitoring and auditing architecture.

Due to the complex nature of the monitoring system and both the domains and the granularity it needs to cover, the monitoring system is made up of a set of components, shown on Figure 8.

Resource Monitor Manager. Resource Monitor Manager is responsible for monitoring resource related metrics and events. Such metrics include CPU utilization, memory usage, jobs status and jobs exit code. These metrics are generally gathered from node-level services, such as AEM, or RSS, as well as from standard kernel or system-level services. Resource Monitor Manager provides means for resource administrators to get feedback on resources behaviour.

Core Monitor Manager. Core Monitor Manager handles monitoring related to grid and VO and provides means for grid administrators to get feedback on core XtreemOS security. Core monitoring data is generally gathered from VO management and other core related services.

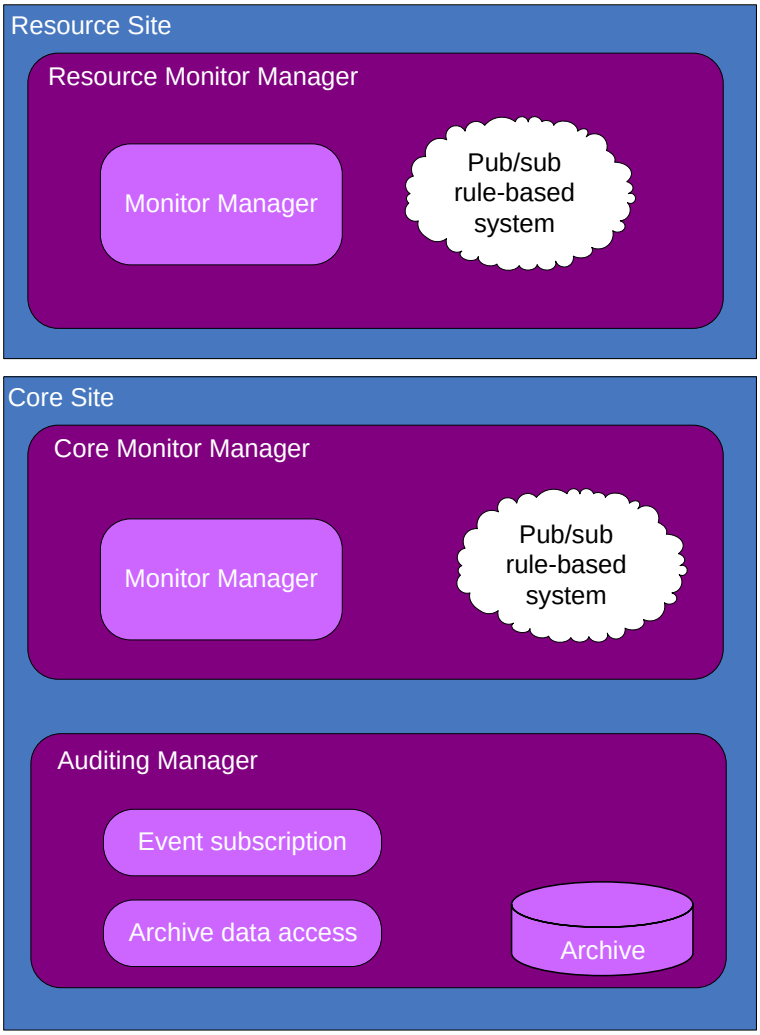


Figure 8: Components of the monitoring and auditing.

Auditing Manager. The purpose of Auditing Monitor Manager is the have a history of important monitored data. Auditing Monitor Manager uses other Monitor Managers as a source and stores collected monitoring data in historical database which can be later queried and analyzed.

3 Security and VO Management Capabilities

This section presents the different capabilities associated with the security and VO management functionality in XtreemOS.

3.1 Describing Use Cases

We describe the use cases using the following terms:

- **Goal** The goal of the use case is a result that brings value to the user.
- **Actors** The entities involved in the use case.
- **Success Scenario** The indication that the use case has been carried out successfully.
- **Pre-conditions** The conditions that must hold before the use case can be executed.
- **Post-conditions** The conditions that must hold after the use case has been executed.
- **Interfaces used** The nature of the interfaces utilised in this use case, either a web front-end or command-line program.
- **Basic Course of Action** The steps that are performed to execute this use case.
- **Optional Course of Action** The steps that may be performed instead of, or in addition to, the Basic Course of Action.
- **Execution Order** (Optional) This indicates a loose ordering between use cases.
- **Sequence Point** (Optional) For those use cases that need to be executed in a strict order, this indicates the relative order of execution with reference to the first use case in the sequence.

3.2 Grid Management Capabilities

This section contains the capabilities for managing the Grid infrastructure underlying VOs. It includes the registering and removal of users and RCAs with the Grid, the registering of local resources with RCAs, the setting-up of the root CA and the running of the various security and VO management services. The Grid

management capabilities are depicted in Figure 9, and below we describe each of the use cases involved.

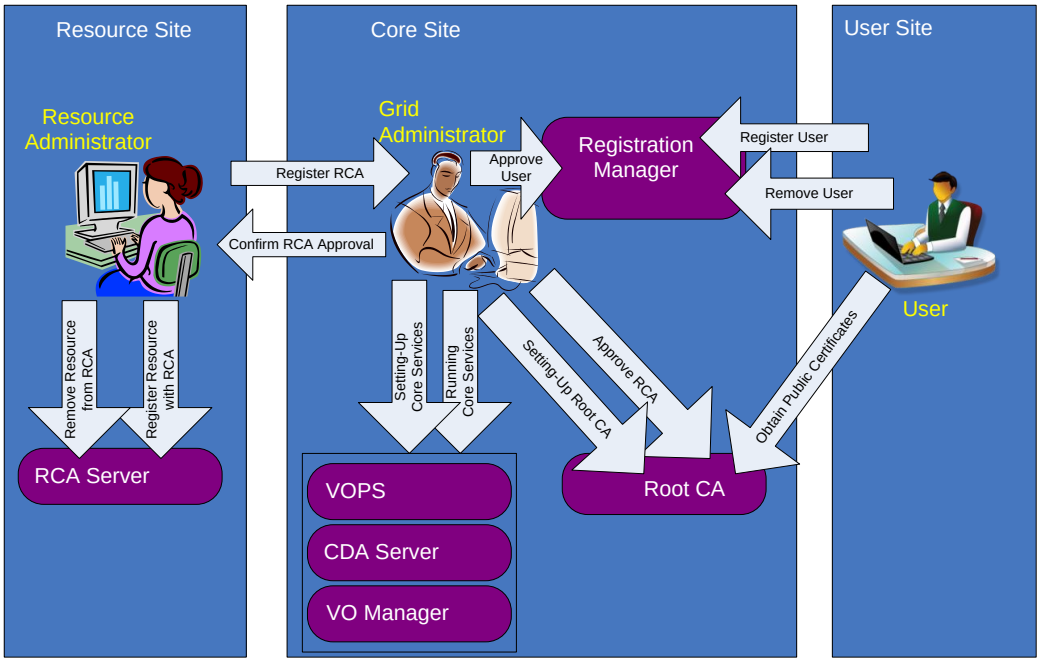


Figure 9: The Grid Management Phase.

Setting up Grid Infrastructure The following steps, 3.2.1 to 3.2.7, are necessary to set up the Grid infrastructure (database, certificates, services).

3.2.1 Configuring and Creating the Root CA

Goal

This use case sets-up a Root CA in the Core Site. This establishes the root of trust for this Grid.

Actors

The Grid Admin.

Success Scenario

The Root CA private key is generated and secured. The Root CA public key certificate is available for processing Certificate Signing Requests for services.

Pre-conditions

The Grid Admin has access to a physically secure machine. Ideally, this is not a networked machine. The Grid Admin has a description of the Grid for which s/he is setting up this Root CA. The Grid Admin has a definition of the Root CA public key certificate lifetime. (This determines when the certificate expires.)

Post-conditions

The Root CA is configured and the Root CA private key is secured. The Root CA public key certificate is available for distribution to other nodes in this Grid.

Interface Used

Command-line programs.

Basic Course of Action

These need to be performed when a Grid is initially set up.

1. The Grid Admin edits the Root CA configuration file to provide a description of this Grid's organisation/project name, and other details such as the lifetime of the Root CA public key certificate.
2. The Grid Admin runs the command "create-rootca" to create the Root CA private key and Root CA public key certificate. The Grid Admin provides a passphrase for the private key and ensures that the key is securely stored.
3. The Grid Admin places the Root CA public key certificate on a networked core node, making it available for public distribution via use case 3.2.7, "Obtaining public certificates"..

Optional course of action - Extending the operation of the Root CA

Before the Root CA certificate expires, a new one should be generated and distributed.

1. The Grid Admin uses the existing Root CA private key to generate a new Root CA public key certificate.
2. The Grid Admin places the Root CA public key certificate on a networked core node.

Execution Order

This is an early step in configuring the Grid and must be executed before any of the Core Services are configured and started running.

3.2.2 Creating the X-VOMS Database

Goal
Create and initialise the X-VOMS database.

Actors
The Grid Admin.

Interface Used
Command-line program.

Success Scenario
The X-VOMS database is initialised and ready for use.

Pre-conditions
The X-VOMS database hasn't already been initialised.

Post-conditions
The X-VOMS database is initialised and ready for use. A password for the database "root" user has been set. A username and password for the Grid Admin have been set to their default values.

Execution Order
This is an early step in configuring the Grid and must be executed before the **CDA** and **VO Web** Core Services are configured and started running.

Basic Course of Action

1. The Grid Admin executes the "xvoms_init.sh" command.
2. The Grid Admin enters a password for the database "root' user and for the "xtreemos-admin" user.

This completes the set up of the X-VOMS database. The X-VOMS database is now available for use by other Core Services.

3.2.3 Setting-up the Core Services

Goal
Configure all the services needed in the core site and start them running.

Actors
The Grid Administrator, and (optionally) a node Administrator.

Success Scenario
The core services are running and ready for users.

Pre-conditions
The Root CA has been configured.

Post-conditions
The core services have security credentials created, consisting of private keys and public key certificates. The operating characteristics of the core services have been defined in configuration files. The core services are running.

Interface Used
Command-line programs.

Basic Course of Action
For each of the core services (CDA, RCA, VOPS, XtreemFS DIR, XtreemFS MRC, XtreemFS OSD), security credentials are created and configuration properties defined by carrying out the following steps:

1. The Node Admin executes "Configuring a single Core Service" use case, 3.2.4.
2. The Grid Admin executes the "Processing Certificate Requests" use case, 3.2.6.

The Root CA certificate and the CDA certificate are considered the "public certificates" and are placed on a networked node for distribution. Additionally, the Root CA certificate should be made available for downloading from the home page of the VO Web front-end.

3.2.4 Configuring a single Core Service

Goal
This use case sets-up a Core Service and starts it running. NB The VO Web front-end is handled separately, in the use case "Configuring the VO Web front-end" below.

Actors
The Node Admin. (This role could be played by the Grid Admin operating on the particular core node for a service.)

Interface Used
Command-line programs.

Success Scenario
Private key for this service installed and secured. Operating characteristics

of the service defined in appropriate properties file. Public key certificate for this service installed.

Pre-conditions

The core service is not already running.

Post-conditions

The core service is configured and running.

Trigger for

This use case triggers 3.2.6 “Processing Certificate Requests”.

Basic Course of Action

1. The Node Admin creates a private key for a core service and a Certificate Signing Request by running the ’create-csr’ command, specifying the type of the service.
2. The Node Admin sends the CSR to the operator of the Root CA. This action triggers use case “Process-CSR”.
3. The Node Admin receives back a public key certificate for this service from the operator of the Root CA, and installs the certificate in the appropriate location.
4. The Node Admin configures the operation of the service by editing its configuration file.
5. The Node Admin starts the service running.

Optional Course of Action - extending the operation of the core service

1. The Node Admin uses the existing private key for a core service to generate another Certificate Signing Request by running the ’create-csr’ command, specifying the type of the service.
2. The Node Admin sends the CSR to the Grid Admin. This triggers the use case ’Process-CSR’.
3. The Node Admin receives back a public key certificate for this service from the operator of the Root CA, and installs the certificate in the appropriate location.
4. The Node Admin re-starts the service.

3.2.5 Configuring the VO Web front-end

Goal
This use case sets-up the VO Web front-end and starts it running.

Actors
The Grid Admin.

Interface Used
Command-line programs.

Success Scenario
The VO Web front-end is available for users.

Pre-conditions
The VO Web front-end is not already running. The X-VOMS database has been initialised. The CDA service has been configured.

Post-conditions
The VO Web front-end is configured and running.

Basic Course of Action

1. The Grid Admin places the Root CA public key certificate in a location where it can be downloaded from the VO Web home page.
2. The Grid Admin configures the VO Web front-end by specifying in its configuration file the location of CDA private key and public key certificate, and the passphrase for the CDA private key.

   Optional step: securing the VO Web front-end using the SSL protocol

   - The Grid Admin creates a private key and CSR for the VO Web front-end. The Grid Admin converts the CSR into a public key certificate, and creates a Java keystore file containing the private key and public key certificate. The Grid Admin configures the Tomcat web server to use the SSL protocol and specifies the location of the keystore file.
3. The Grid Admin starts the Tomcat service to make the VO Web front-end available to users.

3.2.6 Processing Certificate Requests

Goal
Generate a certificate verifying the identity of a Core Service.

Actors

The Grid Admin.

Interface Used

Command-line program to generate certificates. A communications channel (e.g. email) for receiving a Certificate Signing Request and sending the generated certificate back to the sender. If authentication of the sender is required, an out-of-channel method (e.g. telephone) may be used.

Success Scenario

A CSR for a core service is converted into a public key certificate and sent back to the originator of the request.

Pre-conditions

The root CA has been configured and created.

Post-conditions

A public key certificate for a service has been generated from the received CSR.

Triggered by

This use case is triggered by The Node Admin in step 2 of the use case 3.2.4 "Configuring a Core Service". The Grid Admin can choose to collect multiple Certificate Signing Requests (for a set of Core Services) and process them as a batch, rather than processing each CSR as it is received.

Basic Course of Action

1. The Grid Admin receives a CSR for a core service
2. (Optional) The Grid Admin contacts the sender of the CSR to authenticate this request.
3. The Grid Admin runs the "process-csr" command to convert the CSR into a public key certificate.
4. The Grid Admin sends the certificate to the originator of the request.

Error Course of Action

1. If authentication fails in step 2 above, this use case ends. No certificate is returned to the originator of the request.

3.2.7 Obtain Public Certificates

Goal

This use case allows the User to obtain the public key certificates of relevant services from the root CA.

Actors

The User.

Interface Used

Command-line tool to access repository of service certificates. Need not (should not) be on same machine as Root CA.

Success Scenario

The User has the public certificates of the trusted certification authorities installed on the system, providing her a way for checking for trust in the communication.

Pre-conditions

The certificates for the Core Services have been created by the use case 'Setting-up Core Services'.

Post-conditions

The User has the public certificates of all the security and VO Management services.

Basic course of action

1. The User runs a command-line program to download the public certificates from the repository set up in step 3.2.3.

Populating the Grid With the Grid infrastructure in place, we can allow users to register with the Grid, and add resources to the Grid. These use cases precede the creation of VOs.

3.2.8 Sign up to Grid

Goal

Allow the user to sign up a Grid. Provide a means to request a Grid account by providing account details (username/password) and contact details (Name, Organisation, and email address).

Actors

A prospective User.

Interface Used

VO Web Front-End.

Success Scenario

The User receives a message stating whether they have been allowed access to this Grid or not.

Pre-conditions

The X-VOMS database must be set-up and running.

Post-conditions

The User's account and contact details are registered in the XVOMS database. The User and the Registration Manager share a password.

Trigger for

This use case triggers 3.2.9.

Basic course of action

1. The User provides account and contact details via the registration form on this Grid's VO Web page.
2. The User receives an email confirming that they have been approved as a User of this Grid.

Alternative course of action - User is not approved

- The User receives an email stating that they have not been approved as a User of this Grid.

Special course of action for Grid Administrator

The Grid Admin does not need to register to join the Grid. The Grid Admin is allowed to login with a pre-defined username and password.

3.2.9 Approve User

Goal

This use case allows the Grid Admin at the Core Site to approve a registration request by a prospective User.

Actors

Grid Admin.

Interface Used

VO Web Front-End.

Success Scenario

The Grid Admin either approves a prospective User application or rejects it.

Pre-conditions

The prospective User has submitted a requested to register in the Grid.

Post-conditions

The User's request is approved, and their status is set to "Approved". Alternatively, the User's request is rejected,

Triggered by

This use case is triggered by 3.2.8.

Basic course of action

The Grid admin selects an applicant from the list of pending user registrations. The Grid Admin uses the contact details for an applicant to get in touch with them (e.g. by telephone or email) to confirm that they should be allowed to join the Grid. If the Grid Admin satisfies themself that the applicant should join the Grid, their status is set to "Approved".

Optional course of action

If the Grid Admin recognizes a prospective User's contact details etc, their application can be approved immediately without any further communication. Alternatively, the Grid Admin may require a more elaborate proof of identity, such as checking the applicant's credentials in person, and/or having them vouched for by someone trusted by the Grid Admin.

3.2.10 Sign in to VOWeb front-end

Goal
Allow a registered User to login to this Grid.

Actors
Any user (User, Grid Admin or Site/Resource Admin).

Interface Used
VO Web Front-End.

Success Scenario
The actor is logged-into this Grid.

Pre-conditions
The actor has been approved to use this Grid.

Post-conditions
The actor has access to the VO Web functionality which is described in the following use cases.

Basic course of action
The actor enters their username and password. If these details match in the X-VOMS database, and the actor's has been approved, they are presented with access to the VO Web functionality.

Execution Order
This use case needs to be executed successfully before an actor can execute any of the following use cases. This use case has its inverse in §3.2.12.

3.2.11 Remove User from Grid

Goal
Allow a Grid Admin to remove a registered user from this Grid and stop all of their running jobs.

Actors
Grid Admin

Interface Used
VO Web Front-End.

Success Scenario
The User can no longer login to the Grid. The User's login is disabled. User contact details are kept in the X-VOMS database for a time. The User

is removed from membership of any VOs they have joined. The User is kept as the owner of any VOs they have created; this merely serves as a placeholder.

Pre-conditions
The VO Web front-end must be running. The User must be registered and approved.

Post-conditions
The User is no longer registered in this Grid. The X-VOMS database contains no references to the User's account or contact details.

Basic course of action
The Grid Admin selects a user from the list of approved users in the Grid. There is an option to remove the user from the Grid. When the Grid Admin performs this operation, the system performs the actions described in §3.2.13

3.2.12 Leave Grid

Goal
This use case allows a registered User to remove themself from the X-VOMS database.

Actors
User

Interface Used
VO Web Front-End.

Success Scenario
The User can no longer login to the Grid. The User's account and contact details have been removed from the X-VOMS database.

Pre-conditions
The X-VOMS database must be running within the domain and the User must have already registered.

Post-conditions
The User is no longer registered with the X-VOMS database. The X-VOMS database contains no references to the User's account or contact details.

Basic course of action
The User goes to their Account Maintenance page on the VO Web interface

and selects the option 'Leave Grid'. They are prompted for their password; if they supply the correct password, their account is disabled. An e-mail is sent to the User confirming that their account has been removed. The system then performs the actions described in §3.2.13.

3.2.13 System removes user

Goal

This use case provides the common steps needed for removing a user from the Grid.

Actor

The X-VOMS system.

Basic course of action

The system performs the following steps:

1. The user's status is changed to 'deleted' in the X-VOMS database. The user can no longer sign-in to the VOWeb front-end.
2. The system revokes all XOS-Certificates issued to the user. The user can no longer assert their identity in the Grid, so they can no longer submit jobs, login to nodes with the "xos-ssh" command, or start an interactive job.
3. The AEM system stops all of the user's running jobs, and cancels any job reservations they have. The user can no longer use any CPU resources in the Grid.
4. The system informs the XtreemFS system to delete all XtreemFS volumes owned by the user. The user can longer use the XtreemFS filesystem.
5. The user is removed from any VOs that they are a member of. See use case 3.4.4.

3.2.14 Change Password

Goal

Allows a registered User to change their password.

Actors

Any actor.

Interface Used

VO Web Front-End.

Success Scenario
The Actor has changed their password.

Pre-conditions
The X-VOMS database must be running within the domain and the User must have already registered.

Post-conditions
The User's password for accessing the VO web front end is changed.

Basic course of action
The Actor goes to their Account Maintenance page on the VO Web interface and selects the option 'Change Password'. They are prompted for their existing password. If they supply this correctly, they are prompted to provide a new password, which is then requested a second time for verification. If these two passwords match, the Actor's password is changed to the new one.

3.2.15 Register RCA

Goal
This use case registers RCAs with the underlying Grid infrastructure. It precedes any VOs established later.

Actors
The Resource Admin (Site Admin).

Interface Used
Registration Manager API.

Success Scenario
The RCA is registered in the XVOMS database. The Resource Admin and the X-VOMS database share a password for managing the RCA.

Pre-conditions
The Registration Manager must be set up and running already.

Post-conditions
A new entry on the registered RCA in the pending requests list.

Basic course of action
The Resource Admin uses the VO Web page to enter the details (description, access point address) on the RCA to be registered.

3.2.16 Approve RCA

Goal
This use case allows the Grid Admin to approve a request by an RCA Admin to join the Grid.

Actors
The Grid Admin.

Interface Used
Root CA API.

Success Scenario
The RCA Admin's request is approved/rejected and the root CA is informed.

Pre-conditions
A request by the RCA Admin is pending.

Post-conditions
The RCA appears in the list of the available RCAs, and can be used by the VO and Resouce Admins.

Basic course of action
The Grid Admin uses the VO Web page to see the list of currently pending RCAs to be approved. The Grid Admin then selects the items on the list and approves or rejects the RCA registration requests.

Alternative course of action
The Grid Admin processes the certificate request for the RCA as per 3.2.6 "Processing Certificate Requests".

3.2.17 Confirm RCA Approval

Goal
In this use case, the Grid Admin informs the RCA Admin of the result of the request to register the RCA in the Grid.

Actors
The Grid Admin and the Resource Admin (Site Admin).

Interface Used
The Web or command-line interface.

Success Scenario
The Site Admin at the RCA is informed of the result.

Pre-conditions

The request by the Resource Admin was approved/disapproved.

Post-conditions

The RCA Admin is aware that the RCA is usable by the VO and Resource Admins.

Basic course of action

By the Grid Admin's approval of the RCA in the VO Web page, the system automatically sends an e-mail to the Resource Admin, notifying of the registration approval decision.

Alternative course of action

The Grid Admin communicates the signed RCA certificate for the Resource Admin.

3.2.18 Register Resource with RCA

Goal

This use case registers machines (resource nodes) with a RCA running within the administrative domain of the machines. It precedes any VOs established later.

Actors

The Resource Admin.

Interface Used

Web interface or RCA command-line interface.

Success Scenario

The machine is registered with the RCA including its details, such as the services running on it and its computational characteristics (CPU, memory, storage capacity etc.).

Pre-conditions

The RCA must be set up and running within the domain.

Post-conditions

The resource's details have to appear on the list of the resources pending for registration.

Basic course of action

The Resource Admin runs the "`rca_apply`" command from the command line of the resource to be registered. The command collects the information on the resource and sends them to the RCA.

3.2.19 Approve the Resource Registration

Goal

This use case allows the machine to obtain the machine certificates and thus to be used within the grid. It precedes any VOs established later.

Actors

The Resource Admin (Site Admin).

Interface Used

Web interface and RCA command-line interface.

Success Scenario

The machine becomes available to be used as a resource in any VOs created in the later phases.

Pre-conditions

A request for the resource must be pending at the RCA.

Post-conditions

The Resource Admin can obtain machine identity and machine attribute certificates for the resource.

Basic course of action

The Site Admin uses the "`rca_list_pending`" command to obtain the list of the resources currently pending for registration approval. For the resources to be approved, the Site Admin then runs the "`rca_confirm`", providing the ID of the resource to be approved as a parameter.

3.2.20 Obtain the resource identity certificate

Goal

This use case provides with the machine identity certificate that the services can use as trust credentials.

Actors

The Resource Admin.

Interface Used

RCA API.

Success Scenario

The machine obtains the machine identity certificate and the machine attribute certificate to identify and describe the resource.

Pre-conditions
The resource must be registered and approved at the RCA.

Post-conditions
The resource must be able to identify itself towards other resources and services.

Basic course of action
The Resource Admin issues the "`rca_request`" command, which creates the machine's private key, has the machine identity certificate signed by the RCA, and installs it.

Error Course of Action If the resource is still pending registration or has been rejected in step 3.2.19, this use-case ends. The resource cannot obtain the machine certificate.

3.2.21 Update Registered Resource information in RCA

Goal
This use case updates the information on the node in the RCA database.

Actors
The Resource Admin.

Interface Used
Web interface or RCA command-line interface.

Success Scenario
The RCA contains the up-to-date information on the services running on the node and its computational characteristics (CPU, memory, storage capacity etc.).

Pre-conditions
The resource is registered with the RCA, and caracteritistics of the resource have changed.

Post-conditions
The resource will be able to obtain the resource attribute certificate with up-to-date information.

Basic course of action
The Resource Admin runs the "`rca_update`" command from the command line of the resource to be registered. The command collects the information on the resource and sends them to the RCA.

3.2.22 Remove Resource from RCA

Goal

This use case removes a machine (node) already registered with its domain RCA from the RCA database.

Actors

The Resource Admin (Site Admin).

Interface Used

RCA API.

Success Scenario

The machine is no longer registered with the RCA and its details are removed from the RCA.

Pre-conditions

The RCA must be set-up and running within the domain. The machine must be registered with the RCA.

Post-conditions

Any request for obtaining machine identity or machine attribute certificates must fail.

Basic course of action

The Resource Admin can use the "`rca_list_registered`" to obtain the list and information on the currently registered resources. The Resource Admin then uses the "`rca_remove`" command, providing the ID of the resource to be removed as a parameter.

3.3 VO Creation

The VO creation capability facilitates the setting-up of VOs, their attributes and their policies. It is not concerned with populating VOs with users and resources, which is considered to be part of the VO evolution capabilities discussed later. Figure 10 represents the VO creation capability, which consists of a single use case described below.

3.3.1 Create VO

Goal

This use case creates a VO. It marks the starting point of the VO lifecycle.

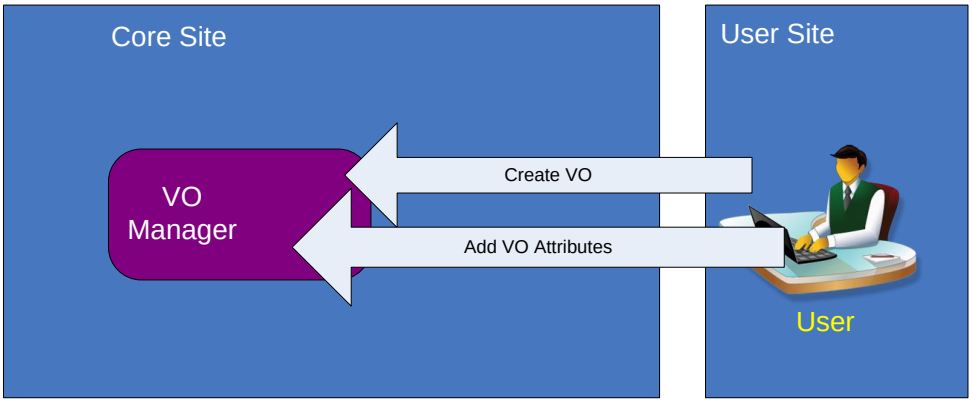


Figure 10: The VO Creation Phase.

Actors
A User.

Interface Used
VO Web Front-End or VO Manager native API.

Success Scenario
A new VO is created for the user.

Pre-conditions
The VO has not been created.

Post-conditions
The VO is created with a unique identifier (GVID). The user is recorded as Owner and Admin of the VO.

Basic course of action
The User selects the 'Create VO' option and is prompted to supply a name for the VO and a description. Submitting these details will create the VO and store its details in the X-VOMS database.

3.3.2 Add VO Attributes

Goal
This use case defines the various VO attributes, such as groups and roles within those groups.

Actors
The VO owner.

Interface Used
VO Web Front-End.

Success Scenario
User has added some extra attributes to the VO.

Pre-conditions
The VO has been created.

Post-conditions
The VO has extra attributes defined, from the set comprising groups and roles. (NB Attributes can be added/changed at any later point in the VO lifecycle.)

Basic course of action
The VO owner adds groups and roles to a VO.

3.4 VO Evolution

The VO evolution capabilities facilitate the management of users, resources and policies within VOs. The following sections describe the various capabilities grouped by users, resources and policy management.

3.4.1 User Management

This capability is related to the management of users as shown in Figure 11. User management includes the following use cases.

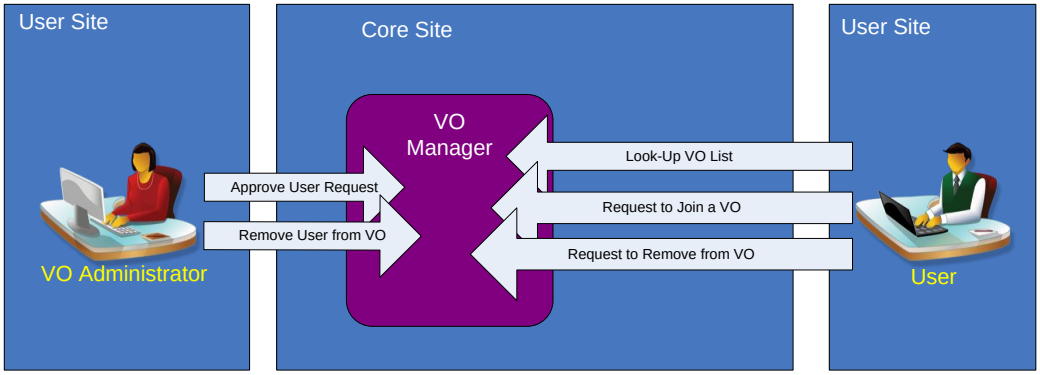


Figure 11: VO Evolution - User Management.

3.4.2 Request to Join VO

Goal
Allow a User to request joining a VO.

Actors
The User.

Interface Used
VO Web Front-End.

Success Scenario
The user has submitted a request to join a VO.

Pre-conditions
The VO has been created, it has as a minimum a GVID.

Post-conditions
The User's request is queued for approval by the VO Administrator.

Basic course of action
The User looks at the list VOs in the Grid. The User selects a VO and submits a request to join that VO.

Triggers
This action triggers use case §3.4.3, "Approve User Request to Join VO".

3.4.3 Approve User Request to Join VO

Goal
Allow a VO Administrator to approve/reject of a User's request to join a VO.

Actors
The VO Admin.

Interface Used
VO Web Front-End.

Success Scenario
The VO Admin has approved or rejected a User's application to join one of the VOs owned by the VO Admin.

Pre-conditions
The User's request has been submitted and is part of the queue of requests to join a VO waiting for approval.

Post-conditions
The User's request is approved or denied. If approved, the User is added as a member of the VO.

Triggered-by
This use case is triggered by §3.4.2.

Basic course of action
The VO Admin looks at the list of pending requests to join VOs that they own. The VO selects a request and can either approve or reject it.

3.4.4 Remove User from VO

Goal
Allow the VO Admin to remove a User from a VO owned by the VO Admin.

Actors
The VO Admin.

Interface Used
VO Web Front-End.

Success Scenario
The User has been removed from the VO

Pre-conditions
The VO contains one or more users, in addition to the VO Admin.

Post-conditions
The User is no longer a member of the VO. The User's association with the VO has been removed from the X-VOMS database.

Basic course of action
The VO Admin selects a VO that they own from a list. This shows the VO membership, i.e. the Users who are members of that VO. The VO Admin selects one of the Users and selects an option to remove them from the VO. The system then follows the steps in use case §3.4.5

3.4.5 Common steps for removing a user from a VO

Actors
The X-VOMS system

Basic course of action

The system performs the following actions:

1. The user is removed from the relevant VO structure in the X-VOMS database. The user can no longer create an XOS-Certificate for this VO.
2. The system revokes the user's XOS-Certificate(s) which reference this particular VO. The user can no longer assert their membership of this VO.
3. The System informs the AEM to stop any jobs which the user has submitted to this VO. The user stops using resources allocated to this VO.
4. The system informs the AEM to cancel job reservations the user has for this VO. The user can no longer reserve any resources allocated to this VO.

3.4.6 User leaves VO

Goal

Allow a User to leave a VO they have joined.

Actors

The User.

Interface Used

VO Web Front-End.

Success Scenario

The User has left the VO and is no longer a member of it.

Pre-conditions

The User is a member of the VO.

Post-conditions

The User is no longer a member of the VO. The User's association with the VO has been removed from the X-VOMS database.

Basic course of action

The User selects a VO from the list of VOs that they have joined. The user selects an option to remove themself from this VO.

The system then performs the actions in use case §3.4.5

3.4.7 Resource Management

This capability is related to the management of resources as shown in Figure 12. Resource management includes the following use cases.

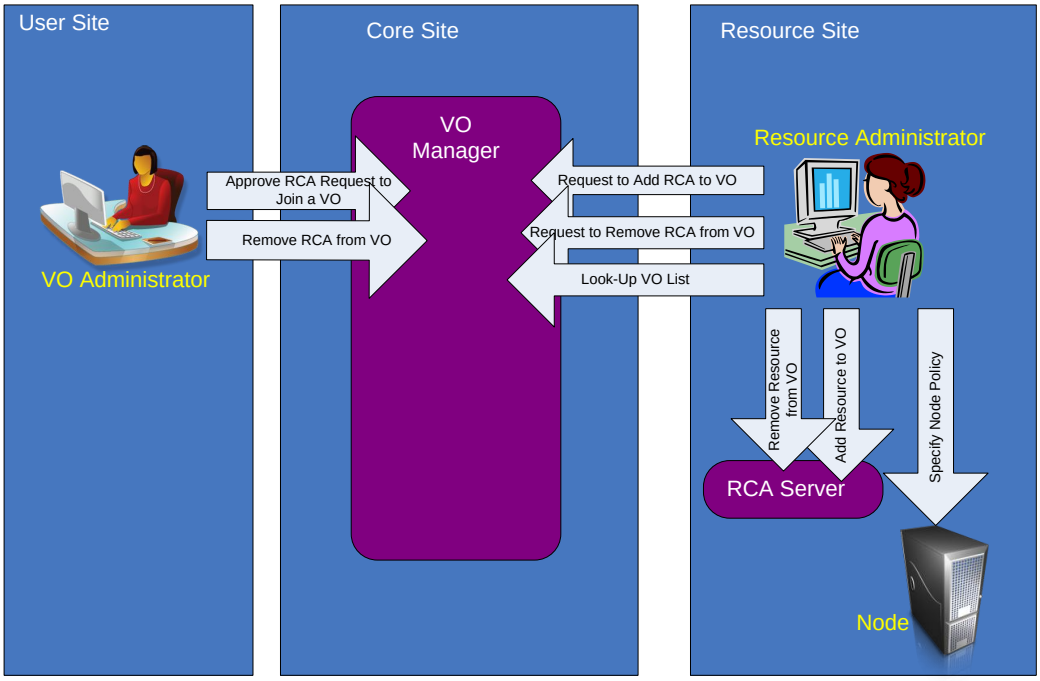


Figure 12: VO Evolution - Resource Management.

Request to Add RCA to VO

Goal

Allow a Site Administrator to join the RCA they control to an existing VO.

Actors

The Resource Admin (Site Admin).

Interface Used

VO Web Front-End or VO Manager native API.

Success Scenario

The Site Administrator's request is queued for approval by the VO Administrator.

Pre-conditions

The VO exists, and the RCA has been registered and approved.

Post-conditions
The request must appear on the list of pending requests.

Basic course of action
The Site Administrator uses the Web interface to list known and active VOs. Selecting certain VOs, the Site Administrator can choose to post a request for the RCA joining a VO.

Approve RCA Request to Join VO

Goal
The Site Administrator's request to join a RCA to a VO is approved or rejected.

Actors
The VO Admin.

Interface Used
VO Web Front-End or VO Manager native API.

Success Scenario
The request is approved or denied. If the request is approved, RCA is notified of the change using RCA's native API.

Pre-conditions
The RCA is registered with this Grid. A request to join a RCA to a VO is queued waiting for approval.

Post-conditions
If approved, the RCA must allow the registered resources to join the VO.

Basic course of action
The VO Admin uses the Web interface to check for pending requests. Finding the request for an RCA to join the VO, the VO Admin can approve the request or reject it.

Remove RCA from VO

Goal
This use case represents the removal of a RCA from a VO.

Actors
The VO Admin.

Interface Used
VO Web Front-End or VO Manager native API.

Success Scenario
The RCA is removed from the VO, and RCA is notified of the change using RCA's native API. Any resources registered in the RCA that are members of the VO will also be removed from the VO.

Pre-conditions
The RCA is a member of the VO.

Post-conditions
The resources must not be able to join the VO or renew their VO machine attribute certificates.

Basic course of action
The VO Admin uses the Web interface to list the RCAs contributing resources to the VO, and selects the RCA removal.

Request to Remove RCA from VO

Goal
In this use case, the Site Admin of the RCA request to remove their RCA from a VO.

Actors
The Resource Admin (Site Admin).

Interface Used
VO Web Front-End or VO Manager native API.

Success Scenario
The RCA is removed from the VO.

Pre-conditions
The RCA is a member of the VO.

Post-conditions
The resources handled by the RCA must not be able to join the VO.

Basic course of action
The Resource Admin uses the Web interface to request the RCA's removal from the VO.

Approve RCA Request to Remove VO

Goal
The Site Administrator's request to remove a RCA from a VO is approved or rejected.

Actors
The VO Admin.

Interface Used
VO Web Front-End or VO Manager native API.

Success Scenario
The request is approved or denied. If the request is approved, RCA is notified of the change using RCA's native API.

Pre-conditions
The RCA is registered with this Grid and is a member of the VO. A request to remove a RCA from a VO is queued waiting for approval.

Post-conditions
If approved, the resources handled by the RCA must not be able to join the VO or renew their VO machine attribute certificates.

Basic course of action
The VO Admin uses the Web interface to check for pending requests. Finding the request for an RCA to remove the VO, the VO Admin can approve the request or reject it.

Add Resource to VO

Goal
This use case allows a site administrator to offer a resource for use in a VO.

Actors
The Resource Admin (Site Admin).

Interface Used
RCA Web Front-End or RCA's command-line front-end.

Success Scenario
The resource is registered with the VO and can obtain the machine attribute certificate.

Pre-conditions
The RCA is a member of the VO and the resource is currently not registered with the VO.

Post-conditions
The resource must be capable of obtaining the VO machine attribute certificate.

Basic Course of Actions
The Resource Admin who owns or maintains the node issues the `rca_resource_voa` console command, which uses the RCA Client API to obtain the machine attribute certificate.

Obtain the resource's VO credential

Goal
This use case allows a resource administrator to have the resource use in a VO.

Actors
The Resource Admin.

Interface Used
RCA's command-line front-end.

Success Scenario
The Resource Admin obtains the machine attribute certificate on the relevant resource that can be installed and used as the credential for the resource in the VO.

Pre-conditions
The RCA is a member of the VO and the resource is registered with the VO.

Post-conditions
The resource must be capable of providing the VO credential when needed to contribute to the VO. *Basic Course of Actions*
The Resource Admin who owns or maintains the node issues the `rca_resource_voc` console command, which uses the RCA Client API to obtain the machine attribute certificate.

Remove Resource from VO

Goal
This use case allows a site administrator to remove a resource from a VO.

Actors
The Resource Admin (Site Admin).

Interface Used
RCA Web Front-End or RCA's command-line front-end.

Success Scenario
The resource is no longer registered with the VO.

Pre-conditions
The RCA is a member of the VO and the resource is currently registered with the VO.

Post-conditions
Any request for obtaining the VO machine attribute certificate must fail.

Basic Course of Actions
The Resource Admin who owns or maintains the node issues the `rca_resource_vo r` console command, which uses the RCA Client API to obtain the machine attribute certificate.

Specify Node Policy

Goal
This use case allows a site administrator to specify the security policies related to the VO at each of their resources.

Actors
The Resource Admin (Site Admin).

Interface Used
Resource Policy Management Tool.

Success Scenario
The resource policy is updated with the VO policy.

Pre-conditions
The resource is in the domain of the Site Admin.

Post-conditions
Any policy decisions must take into account the policy.

3.4.8 VO Policy Management

This capability is related to the management of VO Policies as shown in Figure 13. VO policy management includes the following use cases.

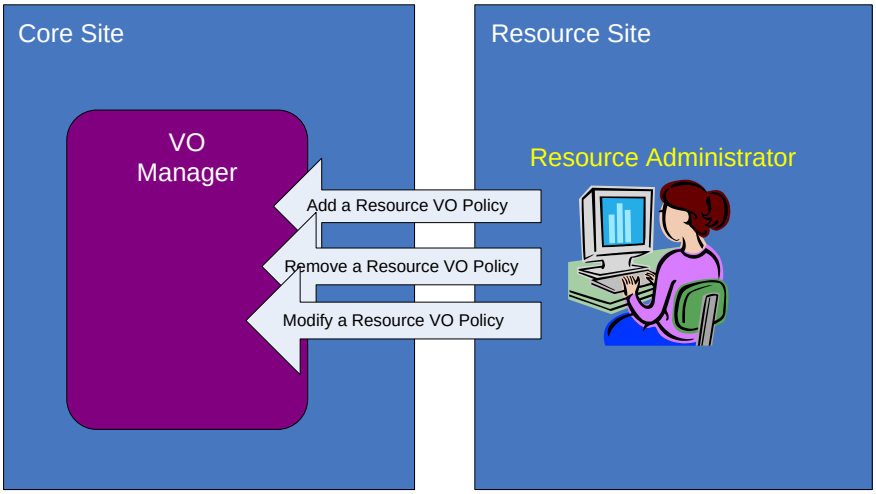


Figure 13: VO Evolution - VO Policy Management.

Modify VO Policy

Goal
: This use case concerns the modification of VO policies already present in the VOPS.

Actors
: The Resource Admin (Site Admin).

Interface Used
: VOPS PAP.

Success Scenario
: Access to particular resource is changed (eihter approved by possibly relaxed rule) or denied (by adding new constraints) for some users.

Pre-conditions
: The resource is in the domain of the Site Admin and a VO policy exists for that resource.

Post-conditions
: The VO policy is updated for the particular resource.

Add VO Policy

Goal
This use case concerns the addition of new of VO policies.

Actors
The Resource Admin (Site Admin).

Interface Used
VOPS PAP.

Success Scenario
Access to the resource is constrained for particular users.

Pre-conditions
The resource is in the domain of the Site Admin.

Post-conditions
The VO policy is created for the new resource.

Delete VO Policy

Goal
This use case concerns the deletion of a resource's VO policy in the VOPS.

Actors
The Resource Admin (Site Admin).

Interface Used
VOPS PAP.

Success Scenario
Constraints on VO level are reduced enabling more users to contact particular resource.

Pre-conditions
The resource already has a VO policy in the VOPS.

Post-conditions
The resource VO policy is deleted from the VOPS.

3.5 VO Operation

Here, we discuss the capabilities preceding full VO operation, which we call VO Operation capabilities. These are divided into user and resource capabilities.

3.5.1 Users

The VO Operation for users capabilities are shown in Figure 14. They consist of the following cases.

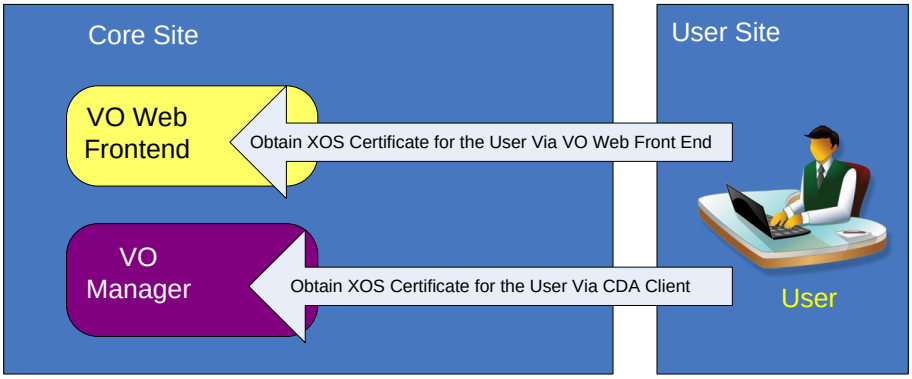


Figure 14: The VO Operation Phase (Users).

3.5.2 Obtain XOS Certificate for the User via VO Web Front-end

Goal

This use case allows a User to obtain an XOS certificate from the VO Web Front-End of the CDA Server.

Actors

The User.

Interface Used

VO Web Front-End.

Success Scenario

Pre-conditions

The User is already registered in the Grid.

Post-conditions

The User has an XOS certificate.

3.5.3 Obtain User XOS-Certificate from the CDA server

Goal

This use case allows a User to obtain an XOS certificate from the native API of the CDA Server and using a CDA Client.

Actors
The User.

Interface Used
CDA client command-line program. CDA server uses CDA Native API.

Success Scenario
The user receives an XOS Certificate containing their public key, Grid identity, and VO attributes.

Pre-conditions
The User is already registered in the Grid.

Post-conditions
The User has an XOS certificate. The CDA server has a record of the certificate's serial number, expiry data, and the global identifier which requested it.

Basic course of action
The user invokes the "get-xos-cert" command.

1. The user invokes the "get-xos-cert" command, specifying the VO and primary VO Group.
2. The user inputs their Grid login details (username and password from §3.2.8) to the CDA client, which authenticates with the CDA server.
3. The CDA client creates a new private key, unless the user has specified the use of an existing private key. The CDA client sends a Certificate Signing Request to the CDA server, containing their public key.
4. The CDA server returns to the user an XOS-Certificate containing the user's identity, their public key, and their VO attributes, all signed by the CDA's private key. The CDA server creates a record of this certificate, containing its serial number, expiry date, and user global identifier (GUID).
5. The CDA client verifies the XOS-Certificate; if successful, it outputs the XOS-Certificate.

3.5.4 Resources

The VO Operation for resources capabilities are shown in Figure 15. They consist of the following cases.

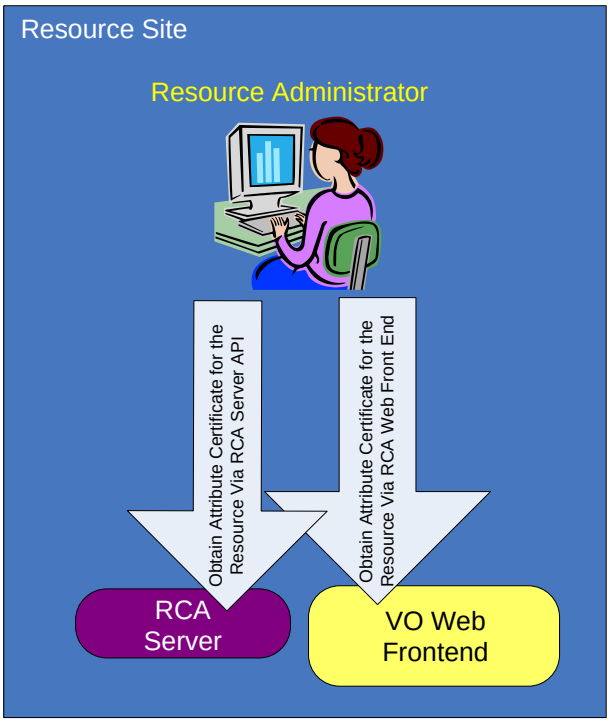


Figure 15: The VO Operation Phase (RCAs).

Obtain Attribute Certificate for the Resource via RCA Web Front-end

Goal

This use case allows a Resource Admin to obtain an attribute certificate from the RCA Web Front-End of the RCA Server.

Actors

The Resource Admin.

Interface Used

RCA Web Front-End.

Success Scenario

The resource has an attribute certificate.

Pre-conditions

The resource is already registered in the Grid.

Post-conditions

The Resource Admin must be capable of installing the up-to-date certificates.

Obtain Attribute Certificate for the Resource via console commands

Goal
This use case allows a Resource Admin to obtain an attribute certificate from the RCA Server's API.

Actors
The Resource Admin.

Interface Used
RCA command-line utilities

Success Scenario
The resource has an attribute certificate.

Pre-conditions
The resource is already registered in the Grid.

Post-conditions
The Resource Admin must be capable of installing the up-to-date certificates.

Basic Course of Actions
The Resource Admin who owns or maintains the node issues the `rca_request` console command, which uses the RCA Client API to obtain the machine attribute certificate.

3.6 VO Termination

The VO termination capability is concerned with the termination of VOs. This capability is shown in Figure 16. The capability consists of the following use cases.

Find and Delete VO Policies

Goal
In this use case, the VO Admin finds and deletes the VO policies for the VO being terminated.

Actors
The VO Admin.

Interface Used
VOPS API.

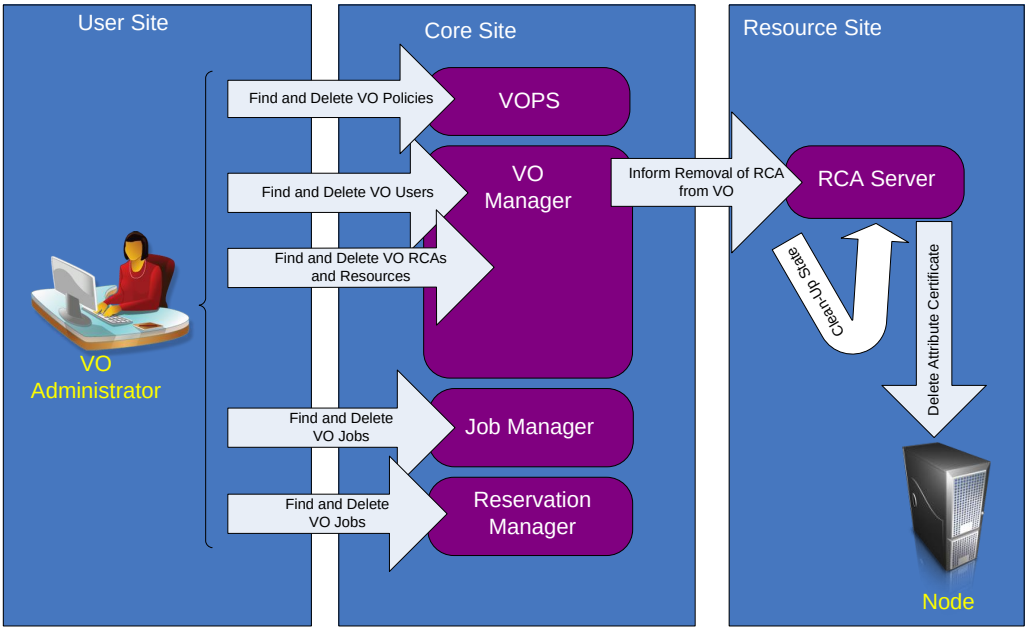


Figure 16: The VO Termination Phase.

Success Scenario

There are no policies for the VO. Jobs can not run in the deleted VO since it does not exist any more.

Pre-conditions

The VO has policies associated with it in the VOPS.

Post-conditions

The relevant VO policies are deleted from the VOPS.

Find and Delete VO Users

Goal

In this use case, the VO Admin finds and deletes the VO users for the VO being terminated.

Actors

The VO Admin.

Interface Used

VO Manager API or VO Web Front-End.

Success Scenario

Pre-conditions
The VO has some users.

Post-conditions
The relevant VO users are deleted from the VO Manager.

Find and Delete VO RCAs

Goal
In this use case, the VO Admin finds and deletes the VO RCAs for the VO being terminated.

Actors
The VO Admin.

Interface Used
VO Manager API or VO Web Front-End.

Success Scenario

Pre-conditions
The VO has some RCAs registered.

Post-conditions
The relevant VO RCAs are deleted from the VO Manager.

Delete Attribute Certificate

Goal
In this use case, RCA deletes the attribute certificate of any resources belonging to the VO being terminated.

Actors
RCA

Interface Used

Success Scenario

Pre-conditions
The resource is in the VO and has an attribute certificate.

Post-conditions
The resource's attribute certificate is deleted.

Inform Removal of RCA from VO

Goal
In this use case, the VO Manager informs the RCA that the RCA has been removed from the VO being terminated.

Actors
VO Manager

Interface Used
RCA API

Success Scenario
The RCA is informed of its removal from the terminated VO.

Pre-conditions

The RCA is active in the VO. *Post-conditions*
The RCA should no longer be able to provide any activities in the VO.

Find and Delete VO Jobs

Goal
In this use case, the VO Admin finds and deletes all the active jobs in the VO being terminated.

Actors
VO Admin

Interface Used
Job Manager API

Success Scenario
All jobs are terminated in the relevant VO.

Pre-conditions

Post-conditions
No job must exist in the VO.

Find and Delete VO Job Reservations

Goal
In this use case, the VO Admin finds and deletes all the active job reservations in the VO being terminated.

Actors
VO Admin

Interface Used
Job Reservation API

Success Scenario
All job reservations are deleted for the relevant VO.

Pre-conditions

Post-conditions
No reservation for the job must exist in the VO.

Clean-up State

Goal
In this use case, the RCA cleans-up the internal state relevant to the terminated VO.

Actors
RCA

Interface Used
RCA API

Success Scenario

Pre-conditions

Post-conditions
The internal state related to the terminated is cleaned.

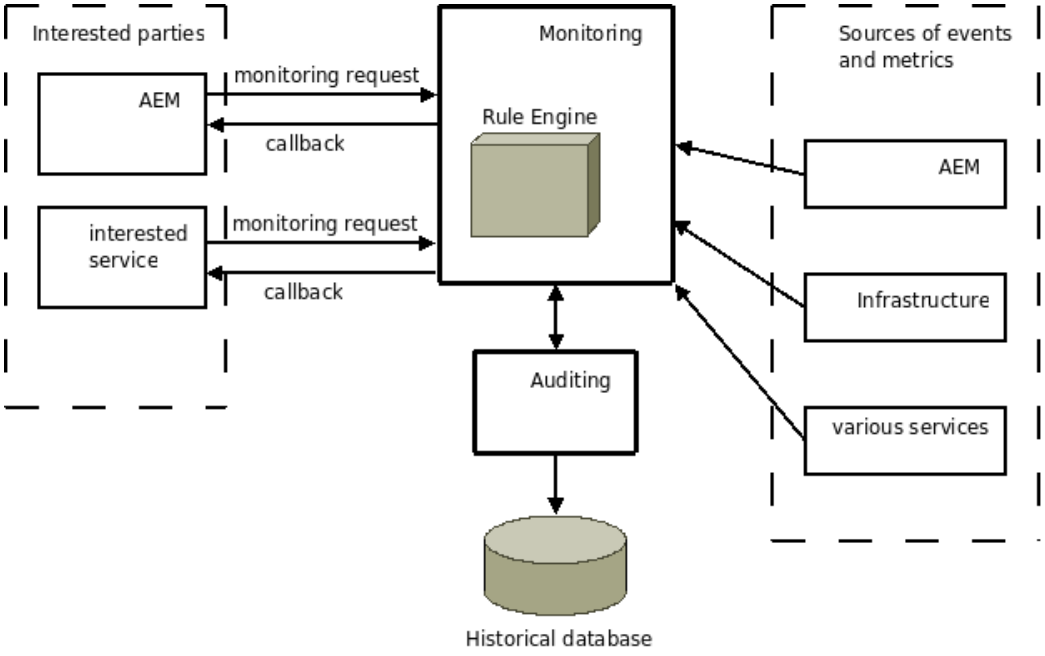


Figure 17: Monitoring overview.

3.7 Monitoring and Auditing Capabilities

Figure 17 illustrates the interactions between services, actors and the monitoring system.

Monitoring provides a means for interested parties to get notified and act upon via callback to different changes that can happen in e.g. infrastructure, services, jobs, resources. In order for interested party to exploit Monitoring it must make a monitoring request which describes what to monitor and when to trigger a callback. The description are basically monitoring rules which are interpreted by rules engine Monitoring uses.

3.7.1 Monitoring capabilities

Monitoring resource metrics.

Goal. This use case allows administrator getting notified when particular resource metric value changes or reaches certain state. Examples of resource metrics are CPU utilization, memory usage, network traffic.

Actors. Any type of administrator.

Interface Used. Monitoring API or command-line tools.

Success scenario. Administrator gets notified on metrics value changes.

Pre-conditions. The User is a member of the VO.

Post-conditions. Callback gets triggered on specified resource metric change.

Basic Course of Action.

- The administrator constructs monitoring rule which describes metrics to monitor and uses monitoring API or `admin_monitoring_rules` command-line tool.
- Services interested in consuming resource metrics use the Monitoring API, passing a rule that describes the metrics to monitor.

Monitoring events.

Goal. This use case allows administrator to get notified when particular event occurs. The user can determine how the occurrence of the event is detected which includes parsing of log files and debug information or by notification issued by other services.

Actors. Any type of administrator.

Interface Used. Monitoring API or command-line tools.

Success scenario. Administrator gets notified on events.

Pre-conditions. The User is a member of the VO.

Post-conditions. Callback gets triggered on specified event occurrence.

Basic Course of Action.

- The administrator constructs monitoring rule which describes events to monitor, and uses monitoring API or `admin_monitoring_rules` command-line tool.
- The administrator uses monitoring API in a custom client program to subscribe to the events.

Monitoring jobs.

Goal. This use case allows administrator to monitor job related information. Different job metrics can be monitored e.g. status, submission time, exit status, but also broader job information can be captured which includes number of jobs currently running on a node or over several nodes.

Actors. Any type of administrator.

Interface Used. Monitoring API or command-line tools.

Success scenario. Administrator gets notified on job metrics changes.

Pre-conditions. The User is a member of the VO.

Post-conditions. Callback gets triggered on specified job metrics change.

Basic Course of Action.

- The administrator constructs monitoring rule which describes jobs and job metrics to monitor, and uses monitoring API or `admin_monitoring_rules` command-line tool.
- The administrator uses monitoring API to add monitoring rule.

Monitoring nodes.

Goal. This use case allows administrator to do monitoring of the nodes. Example of what can be monitored on the node is state of the node and containers running on the node.

Actors. Any type of administrator.

Interface Used. Monitoring API or command-line tools.

Success scenario. Administrator gets notified on node metrics changes.

Pre-conditions. The User is a member of the VO.

Post-conditions. Callback gets triggered on specified node metrics change.

Basic Course of Action.

- The administrator constructs monitoring rule which describes node and node metrics to monitor and uses monitoring API or `admin_monitoring_rules` command-line tool.
- Services interested in consuming resource metrics use the Monitoring API, passing a rule that describes the metrics to monitor.

VO policy violation monitoring.

Goal. This use case enables the notification of policy violations. This is a pre-requirement for enforcing certain policy types in the XtreemOS system services.

Actors. Any type of administrator.

Success scenario. Administrator gets notified on VO policy violation.

Interface Used. Monitoring API or command-line tools.

Basic Course of Action.

- Monitoring reads VO policies from VOPS and construct monitoring rules based on those policies.
- Interested components e.g. AEM or administrators subscribes to the notification of VOPS policies violations using monitoring API or `subscribe` command line tool.

Pre-conditions. The User is a member of the VO.

Post-conditions. Callback gets triggered when VO policy is violated.

3.7.2 Auditing capabilities

Archiving monitored data.

Goal. This use case allows that collected monitoring data is archived in historical database.

Actors. Any type of administrator.

Interface Used. Auditing API or command line tools.

Success scenario. Monitoring data is archived in historical database.

Pre-conditions. Monitoring data is collected.

Post-conditions. Collected monitoring data is stored in historical database.

Basic Course of Action.

- The administrator constructs auditing rule which describes what monitoring data to store, and uses auditing API or `admin_auditing` command-line tool.
- The administrator uses auditing API to add auditing rule.

Securing monitored data.

Goal. This use case allows that collected monitoring data is secured using encryption or is access protected.

Actors. Any type of administrator.

Interface Used. Auditing API.

Success scenario. Monitoring data is protected form unauthorized access.

Pre-conditions. The Auditing service and its archive are set up.

Post-conditions. Any events collected from the monitor is archived in a secure database.

Basic Course of Action. The administrator edits the configuration files of the Auditing Service to use the machine's private key or the service's private key.

Identifying failed user logins.

Goal. This is a specific use case which allows that failed user logins such as user attempting to submit a job with an invalid user certificate to be identified and stored in security log.

Actors. Any type of administrator.

Interface Used. Any interface for user interaction.

Success scenario. Failed user login is identified and archived in historical database.

Pre-conditions. User attempted to login with invalid user certificate.

Post-conditions. Failed login event details are archived in historical database.

Querying historical database.

Goal. This use case allows querying historical database to retrieve information that happened in the past.

Actors. Any type of administrator.

Interface Used. Auditing API or command-line tools.

Success scenario. Administrator gets queried records from historical database.

Pre-conditions. The administrator is a member of the VO.

Post-conditions. Historical data is retrieved.

Basic Course of Action. The administrator uses Auditing API or `auditing_query` command-line tool to obtain the historical data.

VO state report generation.

Goal. This use case allows generation of detailed report about VO state in a period of time.

Actors. Any type of administrator.

Interface Used. Auditing API or command-line tools.

Success scenario. Administrator gets VO state report.

Pre-conditions. The user is a member of the VO.

Post-conditions. VO state report is generated.

Basic Course of Action. The administrator issues `generate_auditing_report` command-line tool to obtain the historical data output.

Node state report generation.

Goal. This use case allows generation of detailed report about state of a particular node in a period of time.

Actors. Any type of administrator.

Interface Used. Auditing API or command-line tools.

Success scenario. Administrator gets node state report.

Pre-conditions. The user is a member of the VO.

Post-conditions. Node state report is generated.

Basic Course of Action. The administrator uses auditing API or `generate_auditing_repor` command-line tool to generate node state report.

User behaviour report generation.

Goal. This use case allows generation of detailed report about user behaviour a period of time.

Actors. Any type of administrator.

Interface Used. Auditing API or command-line tools.

Success scenario. Administrator gets user behaviour report.

Pre-conditions. The user is a member of the VO.

Post-conditions. User behaviour report is generated.

Basic Course of Action. The administrator uses auditing API or `generate_auditing_report` command-line tool to generate user behaviour report.

4 The XtreemOS Trust Model

This section gives a general and brief overview of the XtreemOS trust model, first discussed in [3], and used to integrate the different security and VO management services by setting-up trust between them. It also concerns the setting-up of trust between these services and users and resource providers. The main elements of this model are shown in Figure 18.

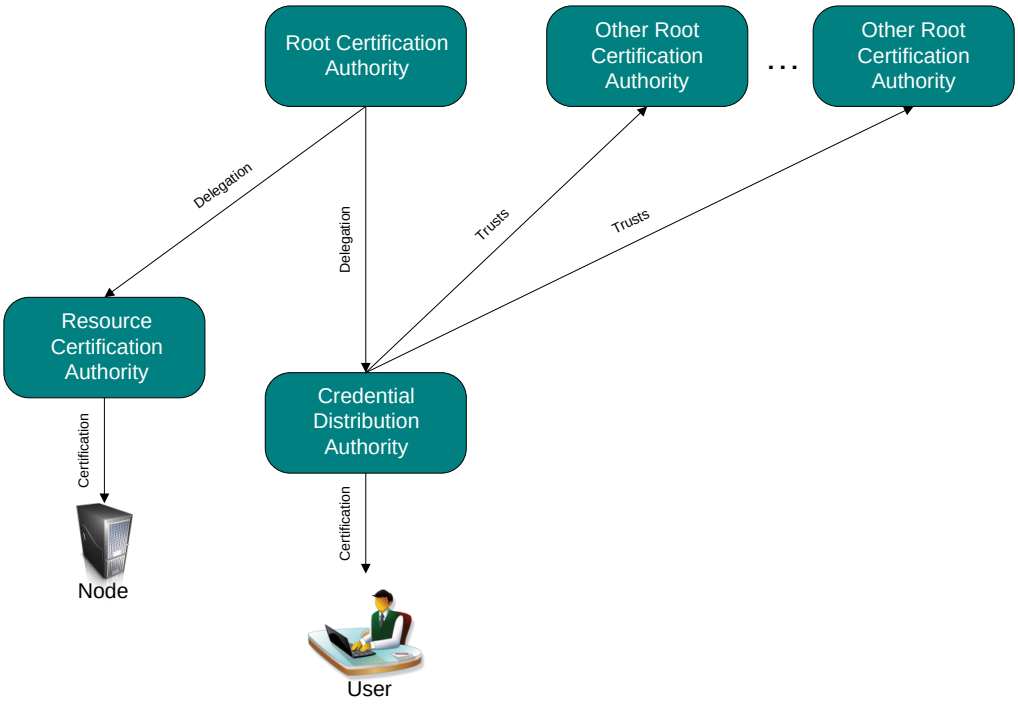


Figure 18: The XtreemOS Trust Model.

In the following sections, we describe briefly these elements. More detail of the trust model and its machinery are presented in a dedicated Deliverable D3.5.9 [4].

4.1 Elements of the Trust Model

The XtreemOS trust model revolves essentially around these main concepts: *credentials*, *certification authorities*, *users*, *resources* and *protocols*.

4.1.1 Credentials

Credentials are pieces of data held by the different actors that provide some form of information required by the capability that the actor requires to perform. Credentials may or may not have been created using cryptographic means. However, most of the credentials used represent some form or another of digital certificates based on the X-509 standard [5]. More detail of the description of these credentials will be presented in the deliverable on trust model in XtreemOS [4].

Passwords. Passwords are the simplest form of credentials created by users and RCA administrators during the initial request for registration phase (part of the Grid Management Capabilities) in a Grid from the Administrator. These passwords will be used in a password-based authentication protocol, such as the Secure Remote Password (SRP) [6], to establish secure (i.e. secretised and authenticated) communication channels.

Root CA Certificates. The Root CA issues identity certificates to the different XtreemOS security and VO management services. This includes issuing certificates to subordinate authorities such as CDAs and RCAs. The certificates will delegate trust from the Root CA to the CDAs and the RCAs. Note that a CDA may choose to trust other Root CAs belonging to other Grids.

User Certificates. User certificates are issued by CDAs and they bind the identity of a user to his public key within a specific Grid scope. Later on, when users become members of VOs, they are issued with attribute certificates to certify that they are members of those VOs.

Resource Certificates. Resource certificates are identity certificates binding the identity of a resource to its public key.

4.1.2 Certification Authorities

Certification authorities represents trust roots and subordinate trust points in the model. Their main task is to issue digital certificates to users and resources.

The Root Certification Authority. The Root Certification Authority (root CA) is the trust anchor for any XtreemOS-based Grid. The root CA issues identity certificates to the core services, such as XVOMS, RCA and VOPS services. Ideally, a root CA should be running on the most secured machine in the Grid, which may even be an offline machine not connected to any network. This is because

compromising the root CA will result in compromising the whole tree of trust in an XtreemOS Grid.

The Certificate Distribution Authority. The Certificate Distribution Authority (CDA) is a subordinate of the root CA to which the root of trust is delegated. The CDA is responsible for certifying user identities and attributes.

The Resource Certification Authority. The Resource Certification Authority (RCA) is a subordinate of the root CA to which the root of trust is delegated. The RCA is responsible for certifying resource identities.

4.1.3 Users

These are the users of the XtreemOS system as discussed in 2.2.

4.1.4 Resources

These are the individual machines that offer resources to the XtreemOS Grid.

4.1.5 Protocols

The final concept in the trust model is the protocols that are used to carry certificates and other data messages among the certification authorities, users and resources in the model. We discuss briefly these protocols next.

The Online Registration Protocol. In many online registration systems, users access a web form to enter their account details (username/password) and contact details. To guarantee the confidentiality of their account details, the web form will use HTTPS. This entails the applicant having to trust and accept the SSL certificate presented by their web browser upon initiating the connection, unless the server certificate has been signed by one of the root certificates installed in their browser. This is an unlikely scenario in using Grids, as the widely-recognised root certificate authorities are commercial operations and getting a SSL certificate signed by them is expensive. The options the user is faced with are:

- They trust without question the SSL server certificate. This is not an uncommon solution, but it doesn't give much confidence in the registration process;
- They can import the SSL server certificate (or the certificate used to sign the SSL server certificate) into their browser. This raises the issue of how the user can obtain the certificates in the first place.

We can achieve the aim of reducing the certificate set up overhead needed with conventional PKI-based approaches by giving the user a *certificate-less* method of securely registering their details.

Here, the idea is that the user/RCA administrator (the 'requester') can access the registration manager over a confidential channel by using SRP [6], a secure password-based protocol. At this stage, the authenticity of the entities involved does not need to be guaranteed, as there is a conventional approval step following registration whereby the identity of the requester can be established. The registration client and manager can share the username/password for a pre-defined registration account to establish an SRP channel. The registration client can hide the *channel* username/password from the requester by automatically establishing an SRP connection when it is invoked, rather than the user needing to provide the username and password for the channel account. Once the SRP channel has been established, the requester can provide their account details (username and password), and their contact details. The registration manager stores this information in the registration database for approval by conventional (manual) means, which we do not describe here.

The Secure Communications Protocol At the end of the first use of the SRP protocol above, the requester has registered a shared username and password with the registration manager. In the next step, the requester will use SRP again to establish a secret session key with the CDA. This time, SRP will use the shared username and password to establish the authenticity of both entities involved in the communication. From now on, we write $[A \longrightarrow B]$ to denote a message sent from A to B over an SRP-secured channel. This implies that A knows it is talking to B and vice versa and they both share a secret session key. In the following sections, we demonstrate the use of such a secure channel to build up certificate distribution protocols among the users/RCAs and the CDAs.

The User Certificate Distribution Protocol In this first certificate distribution protocol, a user U aims at obtaining a root certificate and an identity certificate from the CDA service, C:

1. $[U \longrightarrow C]$: CSR_U
2. $[C \longrightarrow U]$: $(< \mathit{cert}_U >_{SK_C}, < \mathit{cert}_C >_{SK_C})$

where CSR_U is a request from the user U for the certificate signing by CDA, $< \mathit{cert}_U >_{SK_C}$ is the user's identity certificate signed by the private key of C, and $< \mathit{cert}_C >_{SK_C}$ is a self-signed root certificate issued and signed by C.

The Resource Certificate Distribution Protocol In the second protocol, the RCA, R, aims at obtaining a root certificate and identity certificate from CDA, C:

1. $[R \longrightarrow C]$: CSR_R
2. $[C \longrightarrow R]$: $(< cert_R >_{SK_C}, < cert_C >_{SK_C})$

where CSR_R is a request from the RCA for the certificate signing by the CDA, $< cert_R >_{SK_C}$ is the RCA's identity certificate signed by the private key of C, and $< cert_C >_{SK_C}$ is a self-signed root certificate issued and signed by C.

The Protocol between Machines and RCAs In general, machines need to register with at least one local RCA securely. Because machines are operated within the same administrative (trust) domain as their RCA, the problem of establishing a secure channel between a machine and its RCA is resolved locally within the domain. We do not describe here how a secure channel is obtained between a machine and its RCA.

4.2 Setting-Up Trust

An important advantage of the XtreemOS model as outlined thus far is that it cleanly separates user credential management from resource credential management through the use of the CDA and RCA services. Because of this separation of concerns, the addition or removal of users will not impose significant performance and configuration impact on resource management in VOs, and vice versa, with the addition and removal of resources. It also implies that the CDAs and RCAs can easily be maintained (e.g. upgraded or exchanged) in an independent manner as long as the format of their certificates remains compatible with each other.

The key advantage however of the XtreemOS trust model is that it facilitates the bootstrapping of trust through using a set of offline and online processes as described in the following sections. These processes will be described in more detail in Deliverable D3.5.9.

4.2.1 The Registration Process

This process represents in some sense a pre-authentication process. The user or the RCA administrator register their details over a secret (but not authenticated) channel with the online registration manager (see the Grid Management Capabilities in Section 3.2). Their details are recorded in a special database waiting for approval by the Grid administrator. In fact, this vetting step is the only step in the model which requires the administrator's intervention. We consider this to be reasonable because of the trust bootstrapping problem. As part of the user's

or RCA administrator's details, they will be allowed to choose a username and a password shared with the registration manager. As part of this phase, individual resources (e.g. machines) in each administrative (trust) domain can be registered by the RCA/resource administrators with their local RCA.

4.2.2 The Secure Communications Process

The user, who already has registered in the Grid through the process of the previous section and shares with the latter a username and a secret password, proceeds to establish a secure (i.e. secretised and authenticated) channel with the CDA. The RCA, on the other hand, can also establish a secure channel with the root CA. These channels will ensure that communicating parties have been mutually authenticated and that they share a secret session key, which can be used to encrypt all messages exchanged over the channel.

4.2.3 Certificate Distribution Process

The secret session key established at the end of the previous phase is used by the users to obtain from the CDA identity and attribute certificates, which will allow the users to enter the operational mode of the VO. Similarly, the RCAs will obtain identity certificates from the root CA. Once the RCA is certified, it can further issue identity certificates to all the resources it manages in its administrative domain.

5 Detailed Design of the Security and VO Management Services

5.1 XVOMS Design

5.1.1 The XVOMS Classes

In this section, we describe the different classes constituting the design of the XVOMS component. These are divided into the XVOMS Database classes and the XVOMS Utility classes. In both cases, the classes implement a general interface called the *UtilInterface* class, which allows the management of an XVOMS session context.

The XVOMS Database Classes The XVOMS Database structure is expressed by the entity diagram of Figure 19.

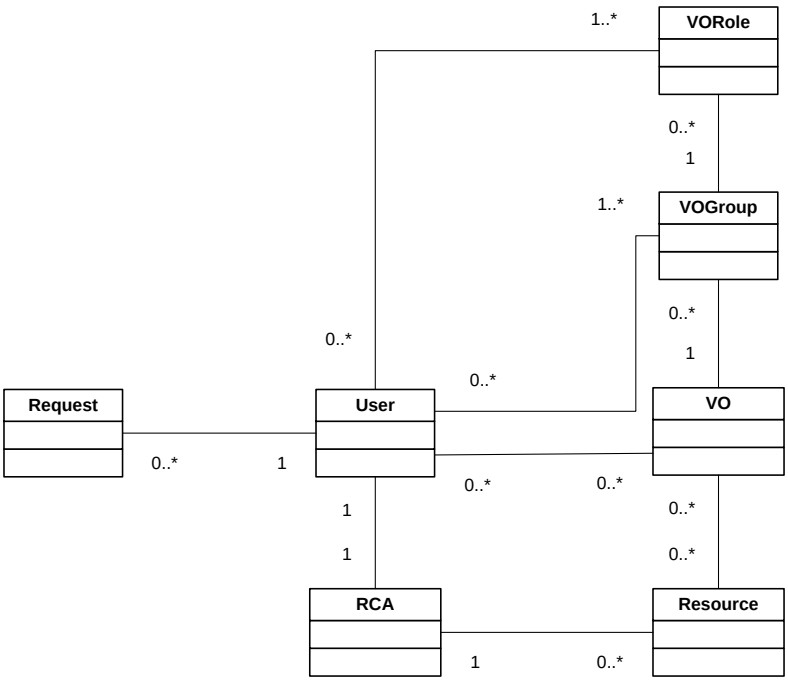


Figure 19: The XVOMS Database Class Diagram.

The structure consists of the following classes:

- *User*: This class represents the XtreemOS actors, which are the shown in Figure 2. It is worth mentioning that the user here, depending on their role,

could become Grid, Resource or VO Administrators. For example, when the user is managing the RCA, it is then considered to be the Resource Administrator in Figure 2.

- *Request*: This class represents requests that the users (actors) submit to the XVOMS system. For example, this could represent the request from a User to join a VO as in Figure 11.

- *RCA*: This class represents the datatype of the RCA entity in Figure 5.

- *Resource*: This class represents the datatype of the Resource entities.

- *VO*: This class represents the datatype of the VO entities.

- *VOGroup*: VOs may have groups, in which case this class represents the datatype of such groups.

- *VORole*: a VO may have roles, in which case this class represents the datatype of such roles.

The above classes are related via a number of relationships, as follows:

- *User and VO*: This is a many-to-many association with no minimum or maximum constraints in either direction.

- *User and VOGroup*: This is a many-to-many association, where a user to at least one VOGroup within a VO, and a VOGroup may have any number of Users.

- *User and VORole*: This is a many-to-many association, where a user belongs to at least one VORole within a VOGroup, and a VORole can have any number of Users.

- *VO and VOGroup*: This is a one-to-many association, where a VO can have multiple VOGroups, but a VOgroup can only belong to one VO.

- *VOGroup and VORole*: This is a one-to-many association, where a VOGroup can have multiple VORoles, but a VOrole can only belong to one VOGroup. There is no relationship between VOs and VORole, however, a VO may have one general group in which all VORoles are defined.

- *VO and Resource*: This is a many-to-many association, where a VO can have multiple Resources and a Resource can belong to multiple VOs.

- *Resource and RCA*: This is a many-to-one association, where a Resource can belong to one and only one RCA. However, a RCA can provide multiple Resources.
- *User and Request*: This is a one-to-many association, where a User can have several requests but a request always belongs to one user.
- *User and RCA*: This is a one-to-one association, where a RCA User (i.e. the Resource Administrator) can manage one RCA and a RCA is managed by one User.

The XVOMS Utility Classes Figure 20 represents the class diagram of the XVOMS component, which consists of two main classes, *UserUtil* and *VOUtil*.

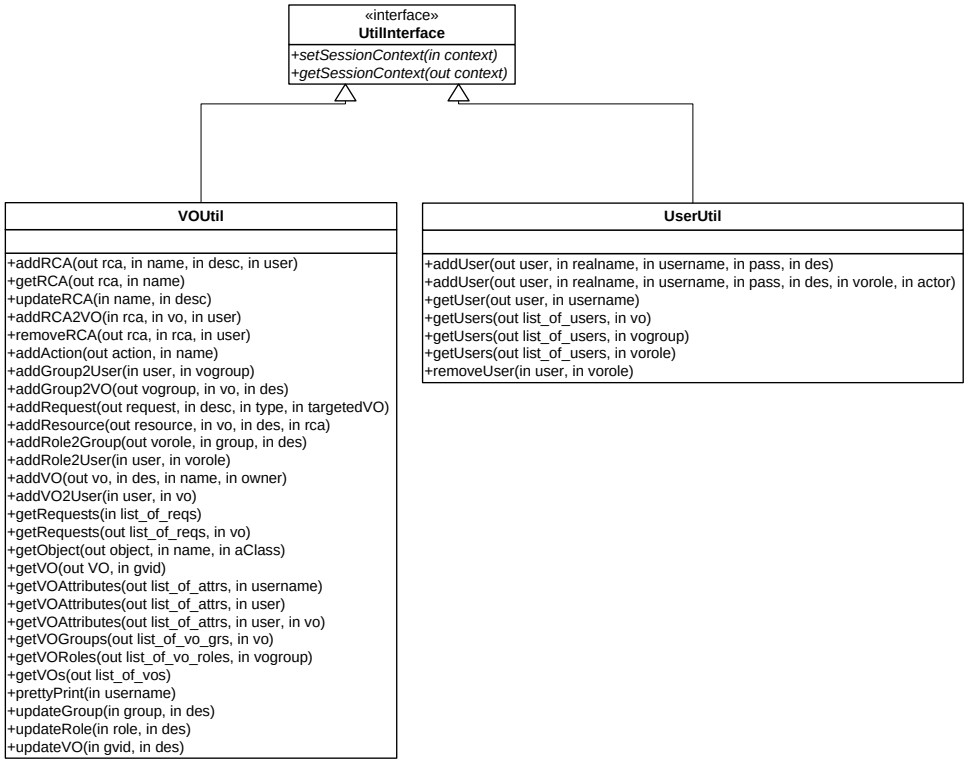


Figure 20: The XVOMS Utility Class Diagram.

Class *UserUtil* provides methods for the management of users in XtreemOS whereas class *VOUtil* provides methods for the management of VOs. More description of these methods will be provided in Section A.1 under the XVOMS API.

5.1.2 The XVOMS Interactions

This section illustrates the processes involved in use of XVOMS via the VO web front end. In Figure 21 the registration process for a new user is shown (use case 3.2.8). The user registers his/her details through the VO web interface. The Grid Administrator then inspects the request and carries out the necessary checks of identity and authorisation, as defined for the current grid (use case 3.2.9). If these conditions are met the Grid Administrator approves the request, otherwise it is denied. If approved the user can then login to the VO web front end as a grid user with the username and password previously defined (use case 3.2.10).

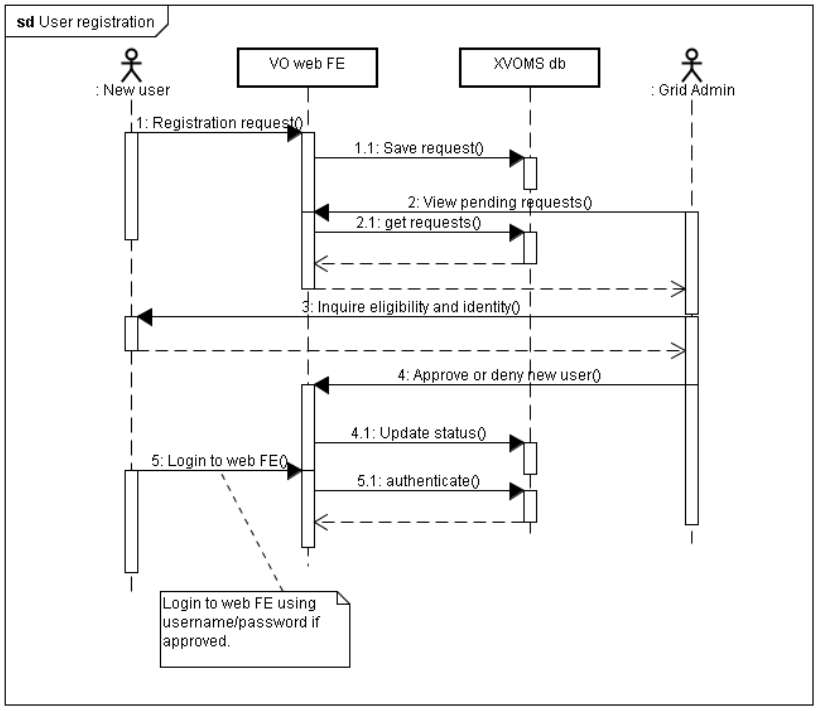


Figure 21: XVOMS registration.

In Figure 22 the steps for a user to join an existing VO are shown (use case 3.4.2). In this case the request is made to the VO Admin via the VO web front end. If the VO owner approves the request the user becomes a member of the VO.

The procedure for a user to leave a VO they had previously joined is shown in Figure 23 (use case 3.4.6). This assumes that the user is not the VO Admin.

The deletion of an existing VO is shown in Figure 24. The deletion of a VO requires that all the users, jobs and reservations of the VO have already been deleted.

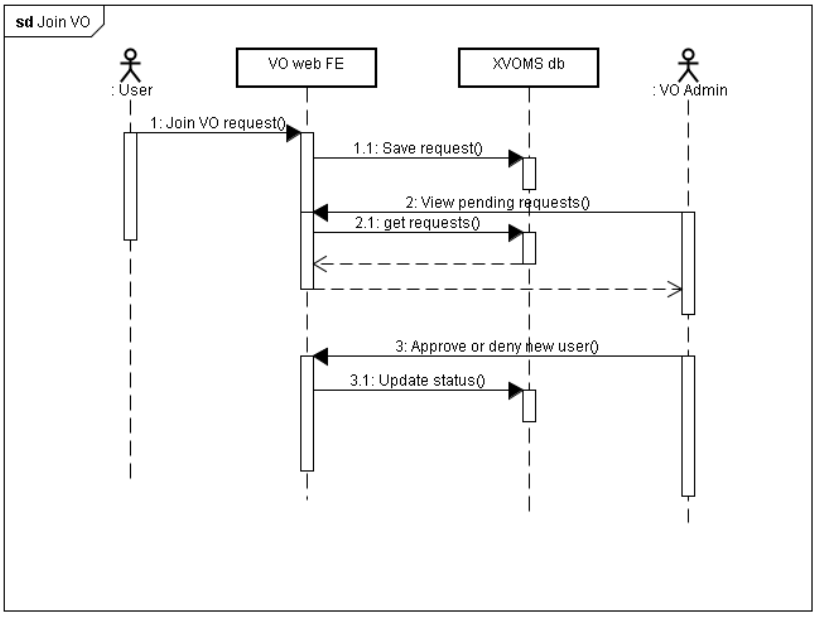


Figure 22: XVOMS join VO.

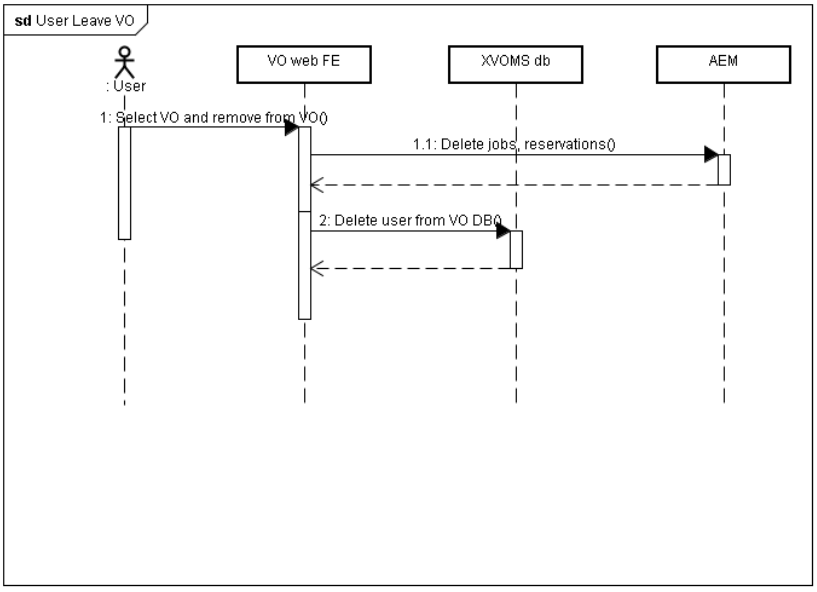


Figure 23: XVOMS leave VO.

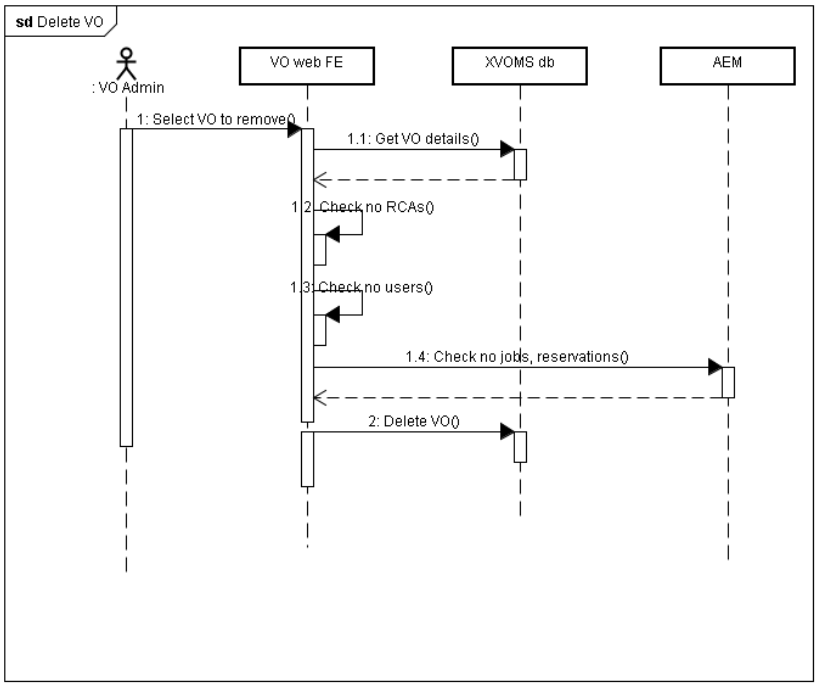


Figure 24: XVOMS delete VO.

5.2 CDA Design

5.2.1 The CDA Classes

The **cdaclient** package comprises two main classes, the *CDAClient* class and the *PeerChecker* class, as shown in Figure 25.

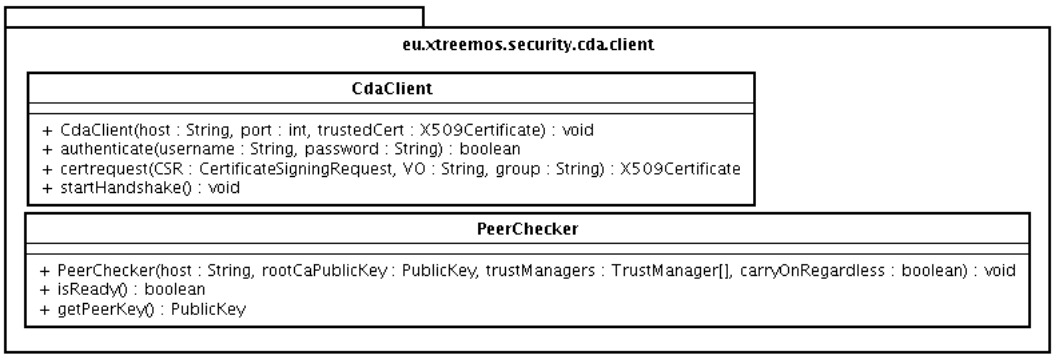


Figure 25: The cdaclient package.

The CDAClient class is used by clients (users) to interact with the CDA server. It has two main purposes; to authenticate a user and to allow that user then to send certificate requests to a CDA server. The *PeerChecker* calss then is used to allow the client to authenticate the CDA server it is connecting to. A more detail description of the methods of these two classes is provided in the section on the CDA Client API, A.2.2.

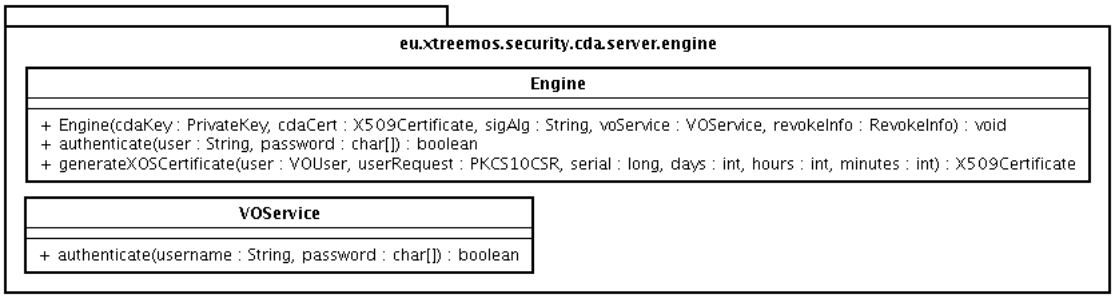


Figure 26: The cdaserver package.

The CDA Server Classes The CDA Server consists of two classes: the main CDA engine class called *Engine*, and a helper class called *VOService*, as shown

in Figure 26.

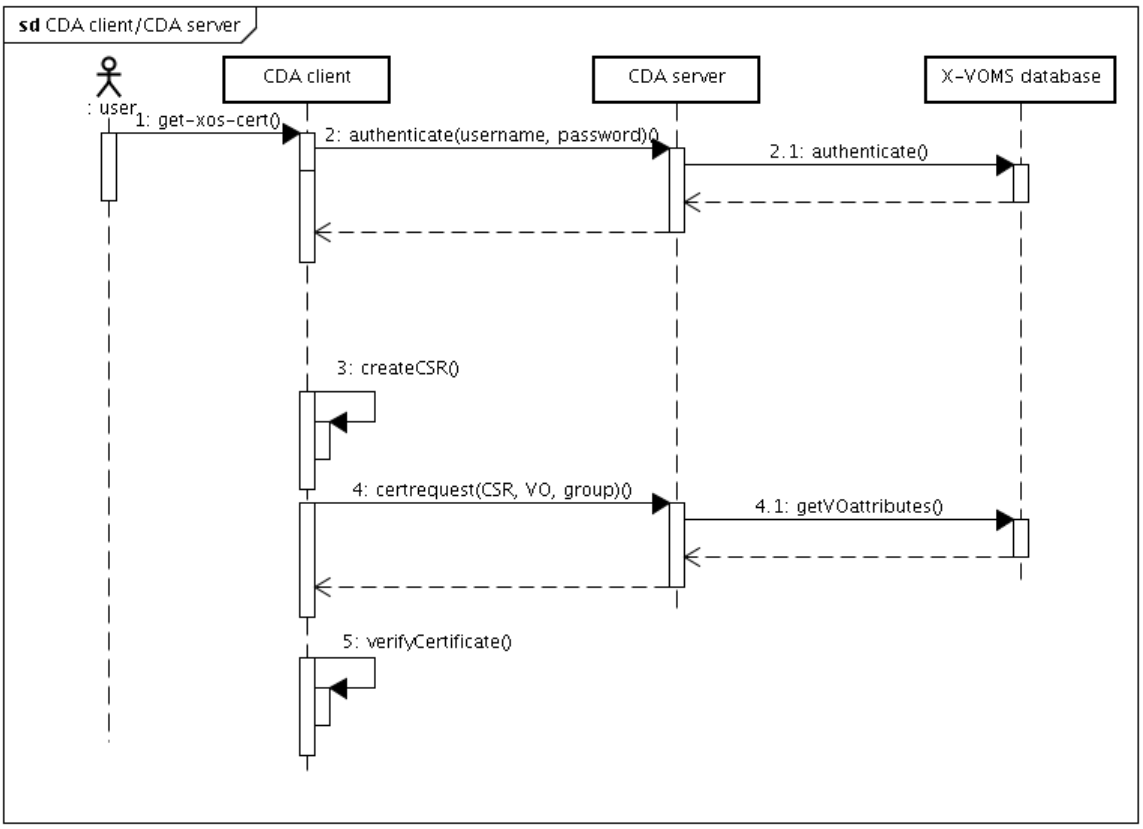
The *Engine* class is the main CDA engine, which is responsible for issuing XOS certificates, whereas the *VOService* class is used to authenticate users before they can use the engine. More detail of both classes is provided in the section on the CDA Server API, A.2.3.

5.2.2 The CDA Interactions

The diagram in Figure 27 shows the interactions between the CDA client and the CDA server. It also shows the interactions between CDA server and the X-VOMS database.

The user invokes the "get-xos-cert" command to start the CDA client. This prompts the user for their username and password, which are sent to the CDA server in the **authenticate** request. If the user is authenticated OK, the CDA client program creates a Certificate Signing Request with the **createCSR** method. The CDA client then sends the CSR, along with the chosen primary VO and primary VO group, to the CDA server, and receives an XOS-Cert in return. The CDA client program then verifies this certificate.

Figure 27: CDA client program, “get-xos-cert”, obtains an XOS-Cert from CDA server.



5.3 RCA Design

5.3.1 The RCA Classes

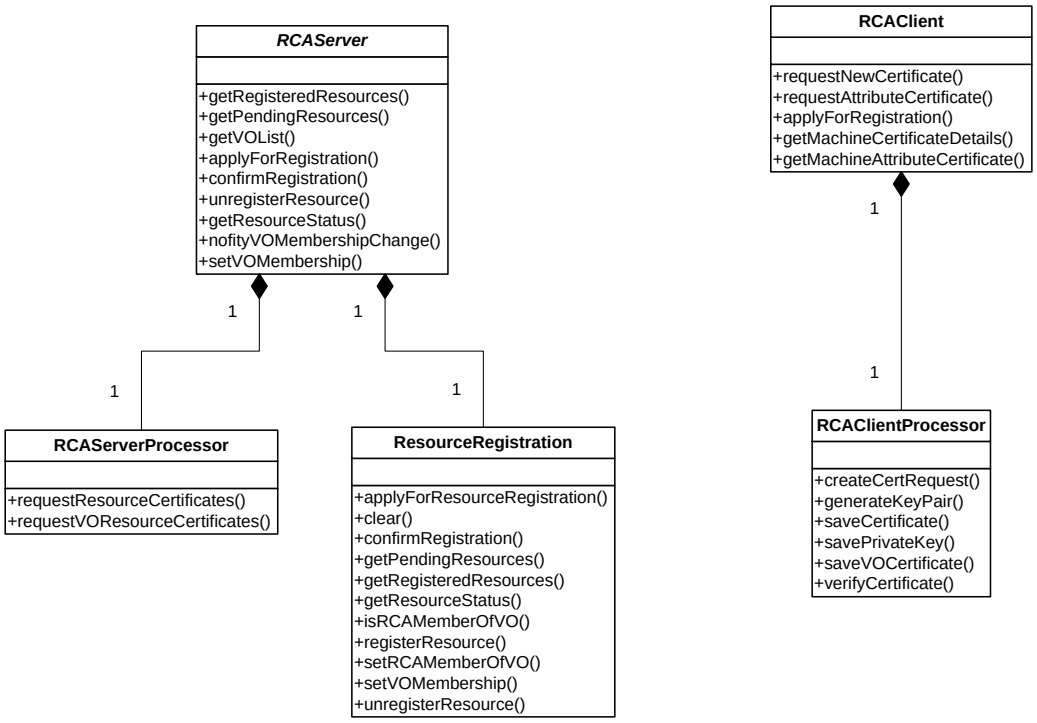


Figure 28: RCA server and RCA client class diagrams.

The design of the RCA components, as depicted on the Figure 28, followed the philosophy of separating the service's core functionality from the front-end that uses a specific messaging bus. The reference front-end developed for XtreemOS uses the DIXI framework and messaging bus.

RCAServer. This is the front-end of the RCA server. It encapsulates the methods for the RCA's functionality, implemented by the depending classes **RCAServerProcessor** and **ResourceRegistration**. Please refer to the API section (A.3.1) for the detailed explanation of the exposed calls.

RCAServerProcessor. The class represents the functionality of storing and manipulating the resource registration entries and query result retrieval. It also maintains the list of VOs that the RCA is contributing to. This is reflected in the methods of the class.

RCAServerProcessor. This class implements the RCA's core functionality, i.e., the capability to compose certificates for the resources, and to sign them using the RCA's service private key. It provides a method for each type of the certificates that might be requested for signing by the resource (machine identity, machine and VO attribute certificates).

RCAClient. The front-end for the service that resides on each node and provides an interface with the RCA server. The API section (A.3.2) explains the methods exposed by this class.

RCAClientProcessor. This class implements the functionality of the RCA client. This includes storing, installing and retrieving the appropriate machine's certificates, generating the machine's key pair, and saving the result obtained through the front-end from the RCA.

5.3.2 The RCA Interactions

The diagrams in this section present the typical processes related to the usage of the RCA. The first one, on Figure 29, shows how the Resource Administrator in charge of the site uses the web interface to send the registration request for the RCA (use-case 3.2.15). The Grid Admin then learns from the web interface about the pending registration requests, and approves the one from the resource admin (use-case 3.2.16), and the Resource Admin gets notified of the approval (use-case 3.2.17).

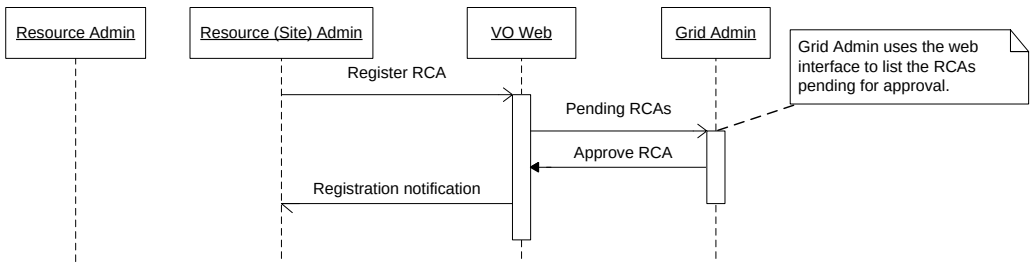


Figure 29: RCA registration.

The RCA can take part in any number of VOs, effectively enabling the nodes registered with the RCA to offer their resources to these VOs. The process of registering a RCA in a VO is shown on Figure 30. In this sequence, the Resource Admin in charge of the site uses the web interface to obtain a list of the available VOs and decides on a VO to offer the RCA to, then posts a request (use-case 3.4.7). The

VO Admin then sees this request as pending, and decides to approve it, making the web application to notify the RCA of the change (use-case 3.4.7).

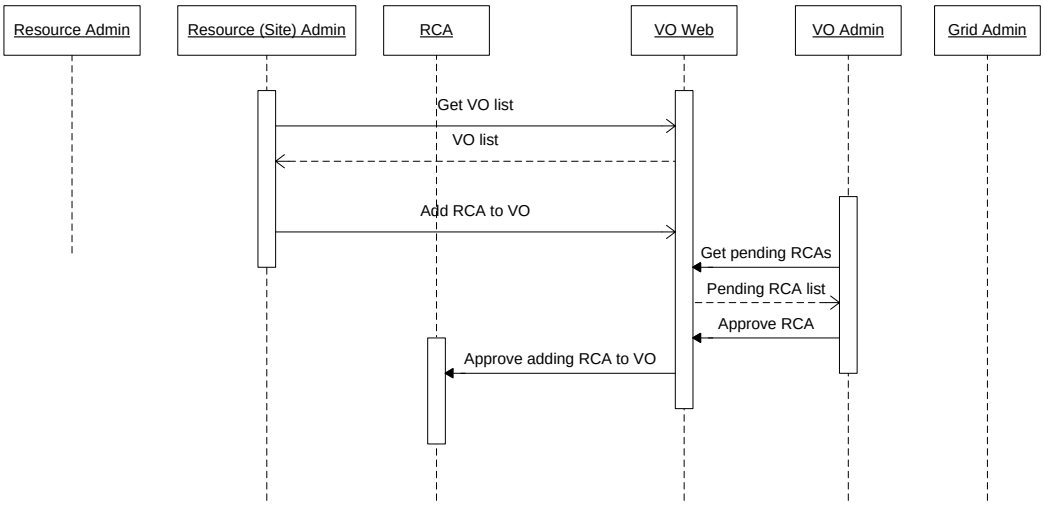


Figure 30: Registering the RCA to a VO.

It is possible to obtain a list of VOs that an RCA is contributing to by querying for the list either from the web interface or through command line directly at the RCA's API. Both cases are shown on Figure 31.

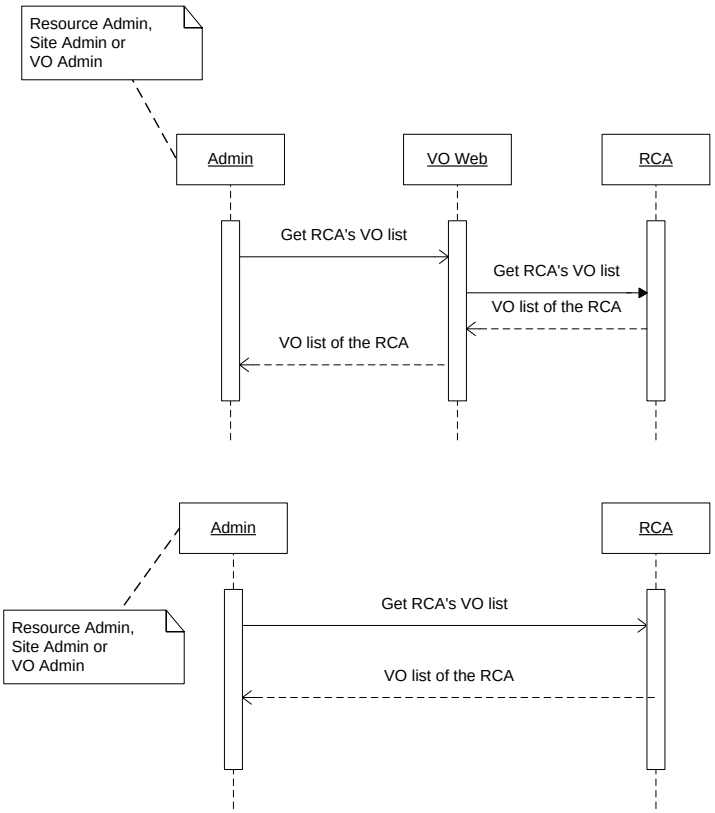


Figure 31: Querying for VOs the RCA is contributing to.

To remove an RCA from a VO, the VO Admin can either use the web interface, which then notifies RCA of the change, or use a command line interface. As the RCA is notified of the change, it also sends the message to the RCA Clients of all the resources registered within the RCA to remove the VO machine attribute certificate. This process is shown on Figure 32.

During the RCA's lifetime, new resources then get registered with the RCA. This process is shown on the Figure 33. The Resource Admin sets up a resource, and sends a registration to the RCA (use-case 3.2.18), effectively creating a request that is pending. The Resource Admin in charge of the site later uses RCA to check the list of requests currently pending, and confirms the resource (use-case 3.2.19). At this point, the Resource Admin owning the resource can request the machine certificates to be signed by the RCA (use-case 3.2.20).

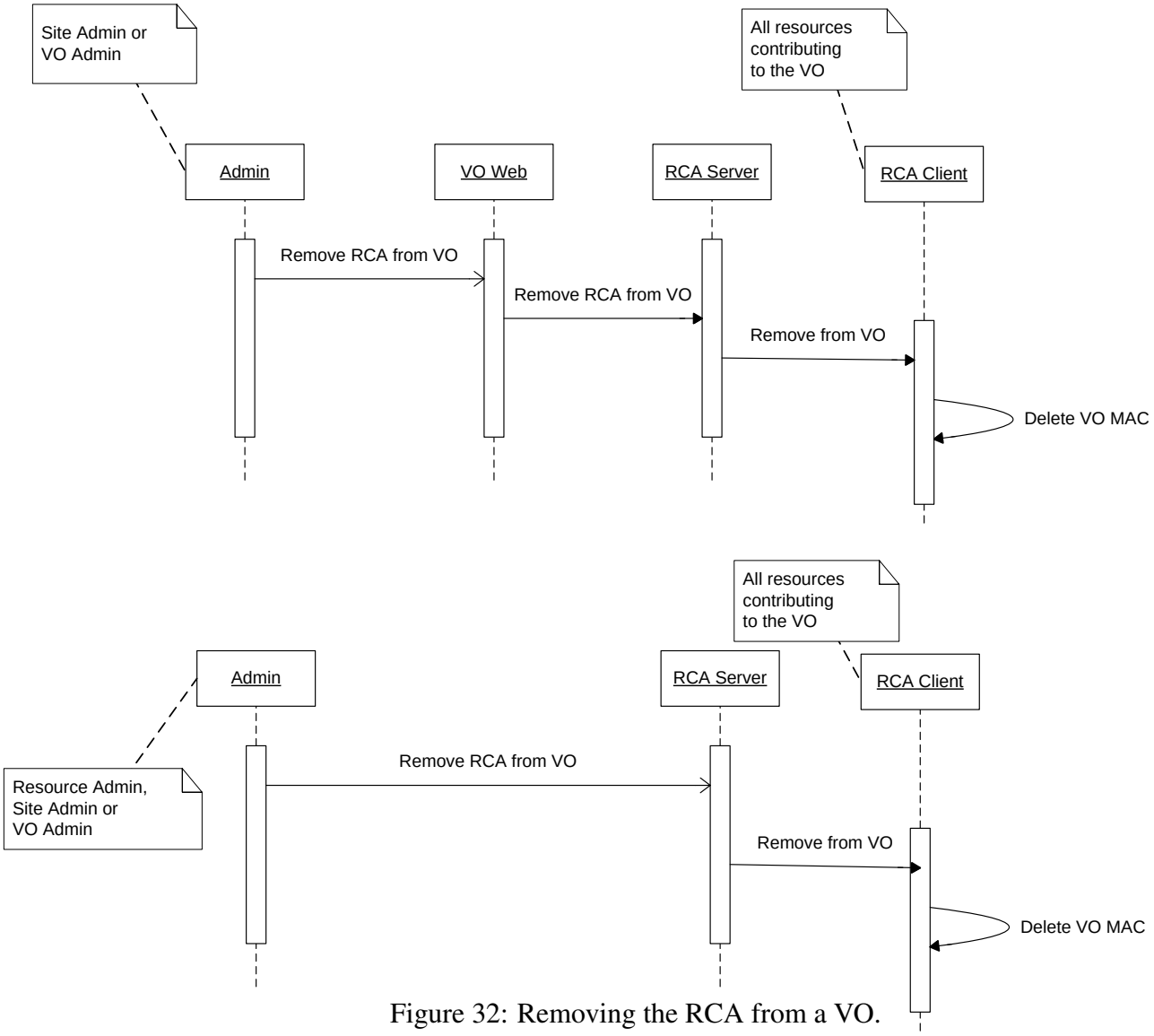


Figure 32: Removing the RCA from a VO.

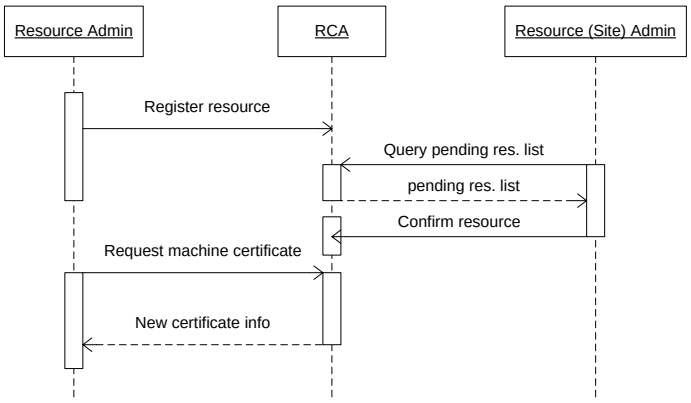


Figure 33: Registering a resource.

In the actual implementation, certain steps from the sequence on Figure 33 are more complex, and further details are shown on Figure 34. The process assumes that the Resource Admin uses the RCA Client to act as a local proxy service to the RCA Server in the process of applying for registration. The line marked as “Resource confirmed” shows the point after which the Resource Admin can successfully request a machine certification. The certification consists of locally creating the public and private key pair, of which the public key is sent to the RCA server in the form of a certification request. The RCA server then uses its private key to sign the certificate.

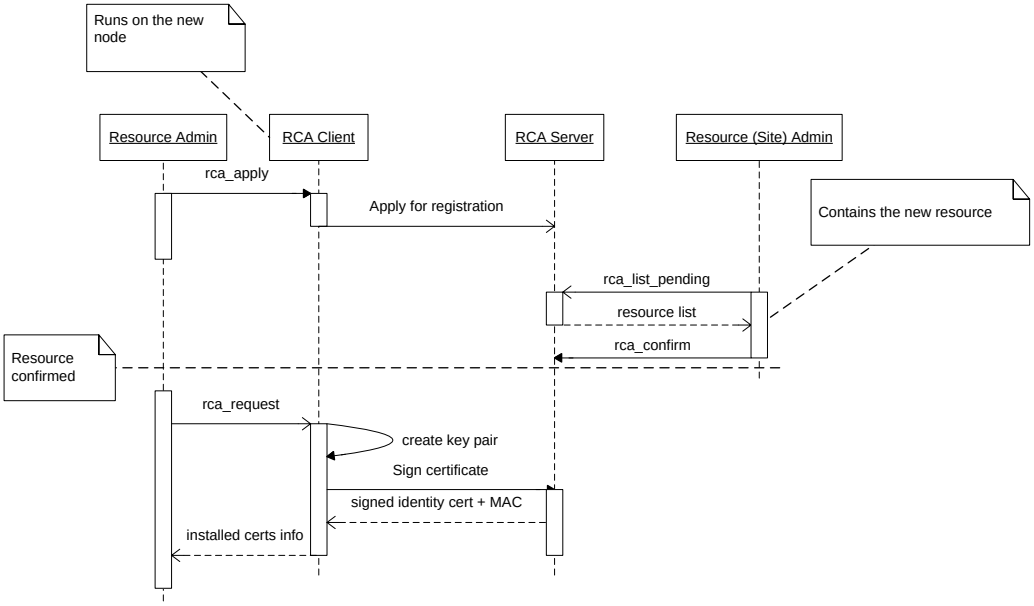


Figure 34: A detailed sequence of registering a resource.

Once the RCA is registered to contribute to a VO, it is possible to have the underlying resources contribute to the VO. The process to achieve that is shown on Figure 35. The VO Admin can use the web interface to ask for a resource to be added to the VO. Similarly, the Resource Admin can use the command-line interface to directly contact RCA (use-case 3.4.7). RCA Server then provides to to the RCA Client the signed machine attribute certificate with the VO identified in the extensions (use-case 3.4.7). The Resource Admin then needs to install the VO MAC for the resource to be actively available to the VO.

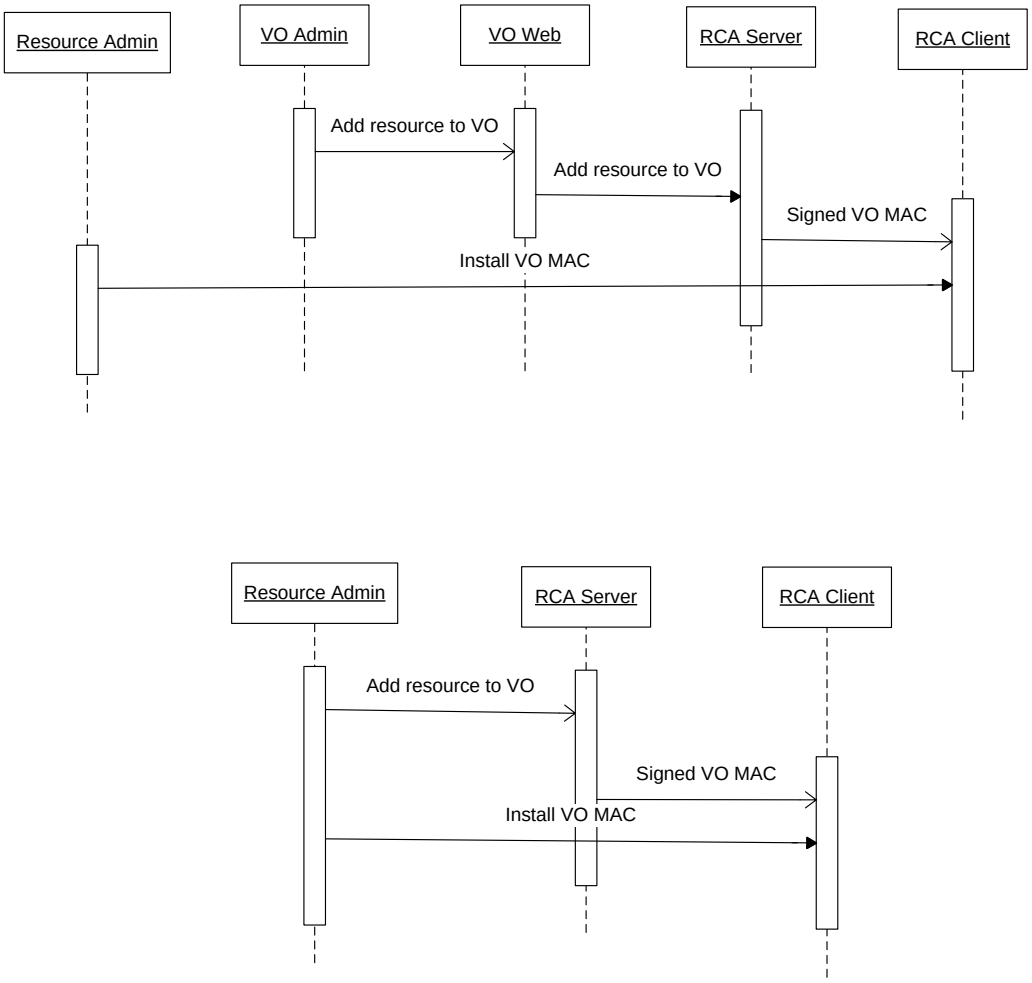


Figure 35: Adding a resource to the VO.

A similar process is involved when the node needs to be removed from the VO, and is shown on Figure 36. Using either the web interface or the command line, the admin can ask the RCA to remove a resource from a VO. This in effect causes the RCA Client to be notified about the removal, and deletes the machine attribute certificate for the VO, if installed (use-case 3.4.7).

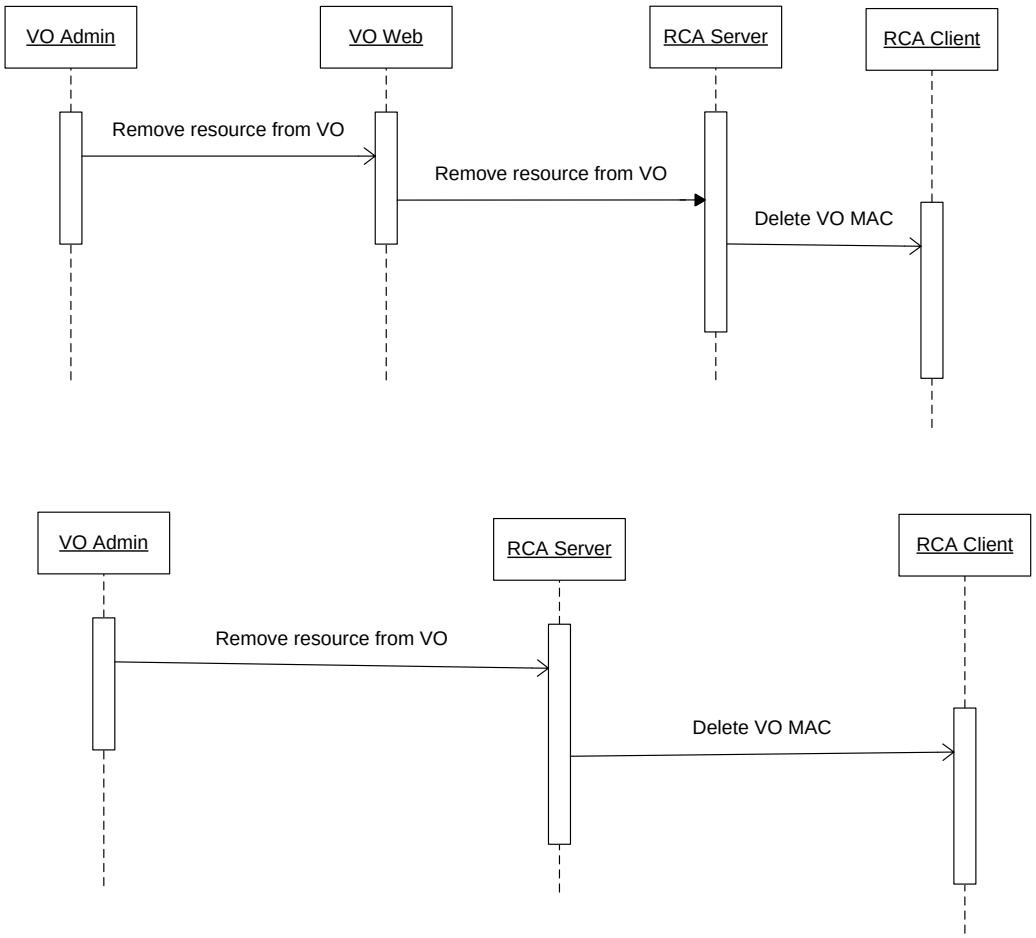


Figure 36: Removing a resource from the VO.

When the resource needs to be removed as per use-case 3.2.22, the Resource Admin or the VO Admin can follow the process shown on Figure 37. Using the web interface, the admin selects a registered resource and requests its removal from the list of the registered resources list. In the case of the command-line interface, the admin can query a list of currently registered resources, and provides the ID of the one to be removed as the console command parameter. This removes the resource from the RCA db, and notifies the RCA Client of the node that has been removed, so that it can clear any installed machine certificates.

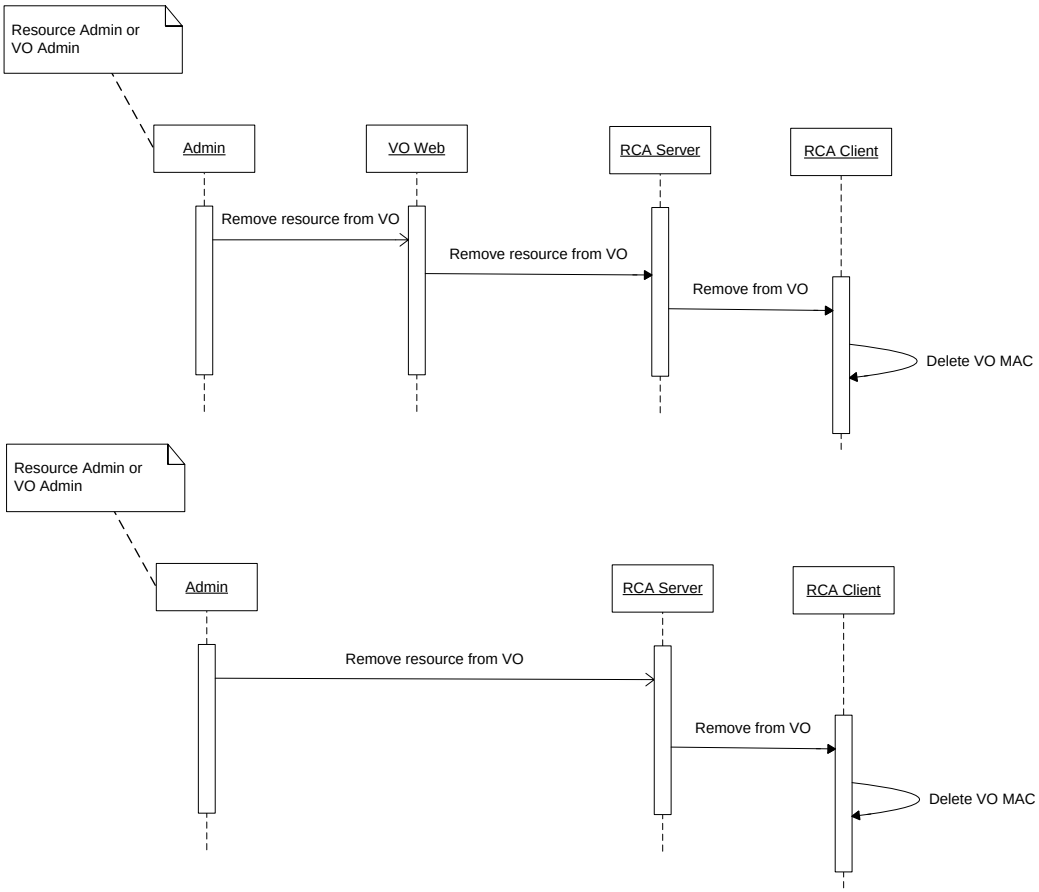


Figure 37: Removing a resource from the RCA.

5.4 VOPS Design

5.4.1 The VOPS Classes

Due to the need of having ability to check policies on different levels (VO level, node level), we provide classes to comply with these requirements.

VOPSServer Main class containing server logic.

EXistsDB Policy storage maintained as an eXist XML database through API provided by the eXist service. Database is organized in tree like structure root (xmldbURI/db)

- system (administrative collections)

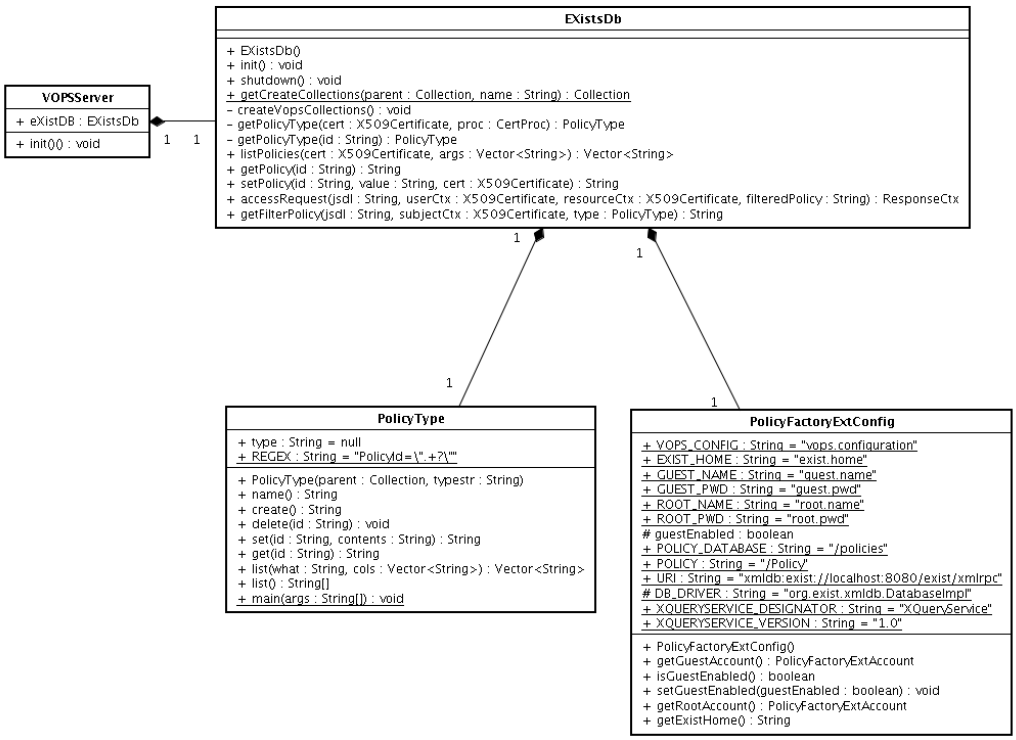


Figure 38: VOPS server class diagram.

- policies

  vo

  user

  resources

On first start database structure is empty, except system is created by database, polices (and sub collections) are created with constructor. Class PolicyType determines type of the policy.

PolicyType Helper for holding collection and query service together.

PolicyFactoryExtConfig. Holds configuration of the PolicyFactory which connects to eXist's XML DB.

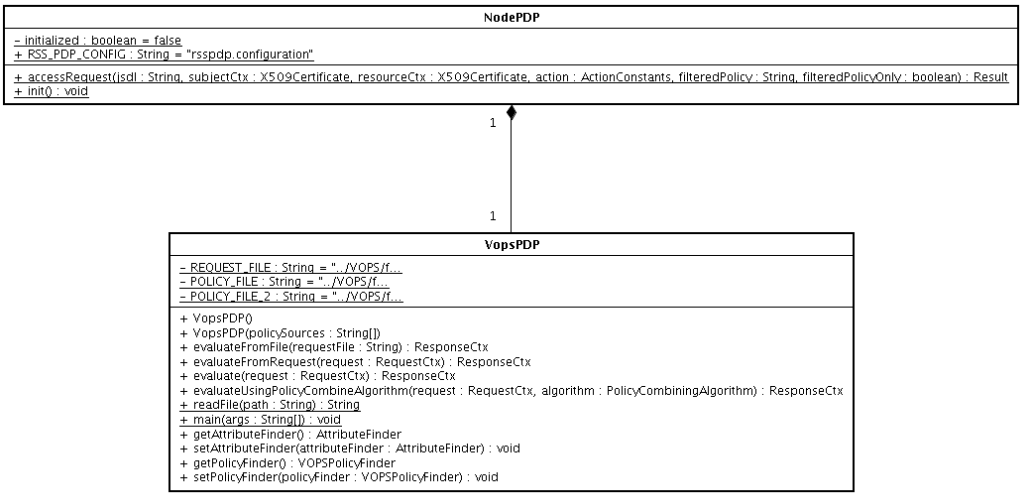


Figure 39: VOPS local decision point class diagram.

NodePDP Class provides method to create a decision based on input parameters, such as job description, user and resource certificate, action and filtered policy. It also provides ability to take into account policies, which are stored locally and besides provided filtered policy help with the decision process.

VopsPDP Provides implementation of the decision engine. It constructs Policy Decision Point with different modules, provided by the Sun's XACML implementation.

5.4.2 The VOPS Interactions

The diagrams in this section present the typical processes related to the usage of the VOPS. In following diagrams only one actor is presented - Resource Administrator, but we can easily replace Resource Administrator with VO Administrator or VO user. Each of these actors can manage policies belonging to a resource, VO or VO user respectively. In figure 40 administrator modifies a resource policy which is first obtained from VOPS server using VO Web frontend. VO Web frontend queries the VOPS server (internally querying Exist XML database) to obtain a list of resource policies belonging to the administrator. Administrator chooses which policy she will modify and afterwards modified (updated) policy is submitted towards VOPS server.

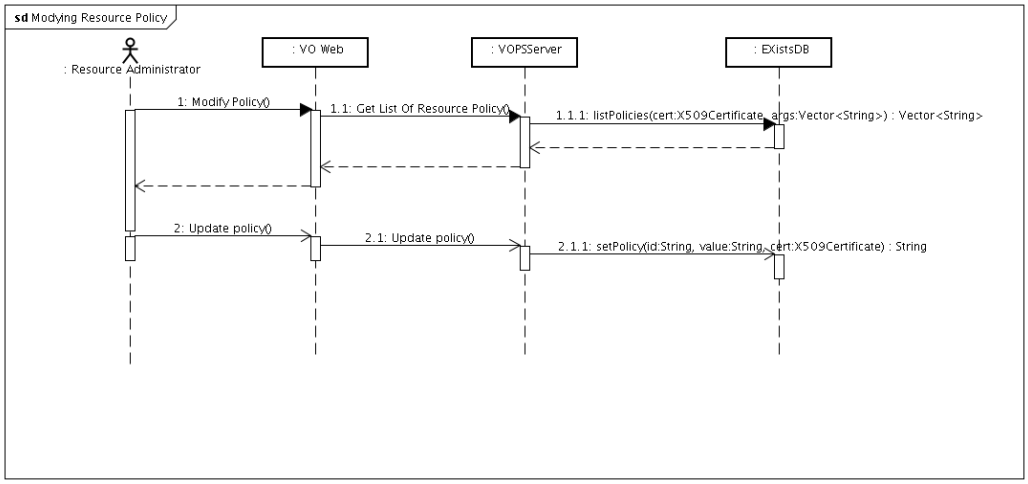


Figure 40: Modifying a resource policy from VOPS database.

In figure 41 sequence of adding a new policy is presented. Administrator chooses Add resource Policy action in the VO Web frontend, where she can specify new policy. After submitting the newly created policy, VO Web sends policy towards VOPS server through Set Policy method where the policy is stored internally in the XML database (through *setPolicy()* method of the *ExistsDB* instance).

Removing a policy (figure 42) is very similar to modifying an existing policy (figure 40). Administrator first chooses Delete Policy method in the VO Web frontend, which gets the list of policies which fall in the domain of the administrator. The chosen policy's id is propagated back to the VOPS server and eventually to ExistsDB which actually deletes the policy from the database.

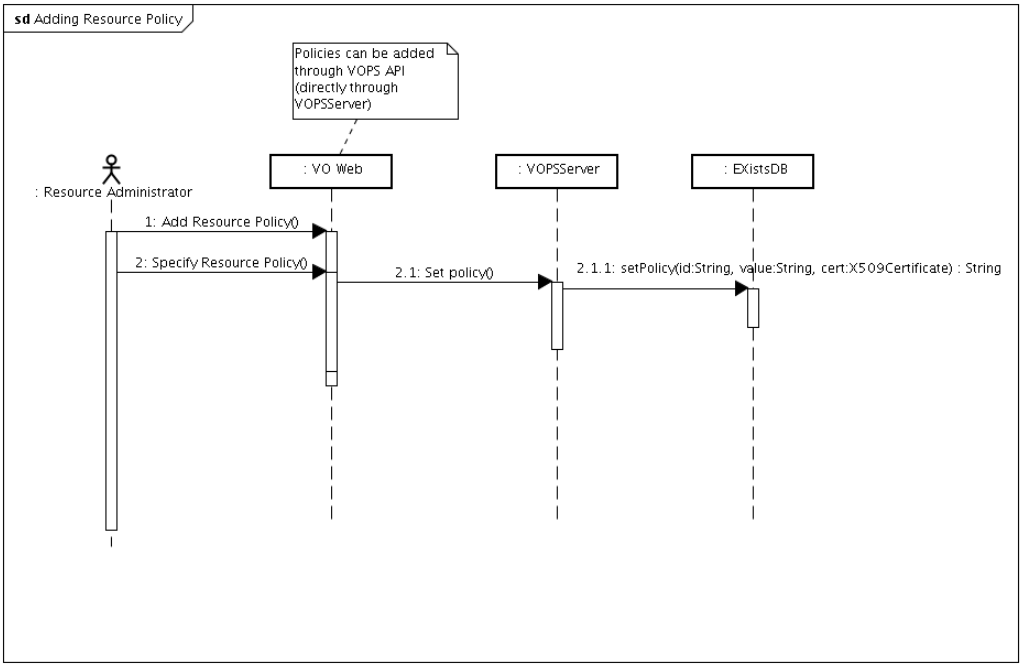


Figure 41: Adding a resource policy into VOPS database.

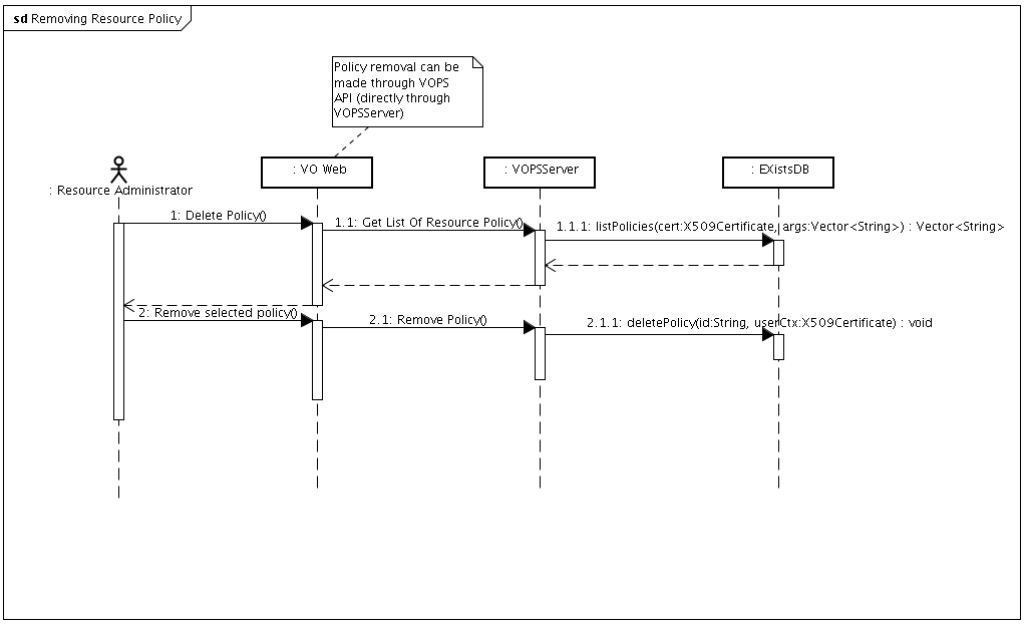


Figure 42: Removing a resource policy from VOPS database.

In figure 43 deletion of the policies belonging to a domain of specific VO is presented. VO administrator chooses which VO policy will be deleted. This kind of policy deletion can be triggered automatically when VO is terminated. First sequence as in figure 40 is made (not presented in the figure below), which gets the policy belonging to a specific VO. Afterwards policy deletion in the XML database is triggered.

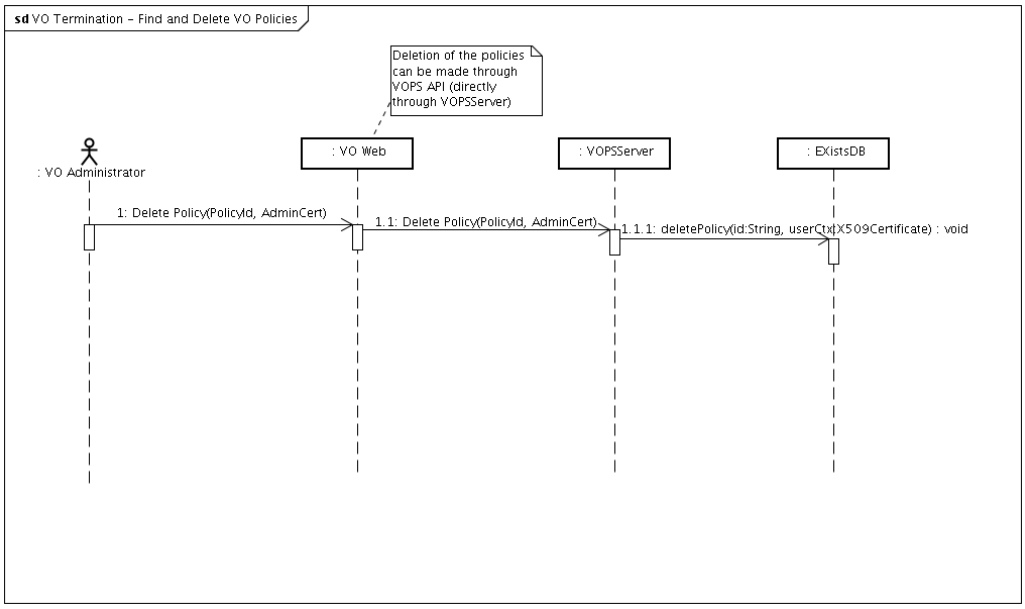


Figure 43: Removing a resource policy from VOPS database.

5.5 Monitoring Service Design

5.5.1 The Monitoring Service Classes

Class shown on figure 44 represents interface to the monitoring service. More detailed description of monitoring manager API is shown in section A.5.

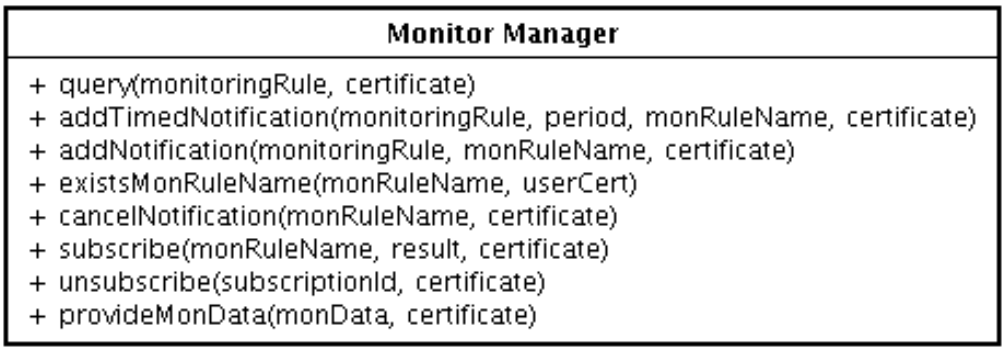


Figure 44: Monitor Manager class diagram.

Monitoring rules. When interested party sends monitoring request Monitoring uses monitoring rule engine to check metric values against collected data from the sources and triggers a callback if conditions are met.

Monitoring rules are represented with a language that contains one or more criteria. Basic expression constructed with single criterion is defined as

```
{<domain>=<domain_reference>}<metric_name><operator><reference_value>
```

and can be further extended.

Domain defines the granulation we are interested in e.g. resource, job, VO.

Metrics are represented as different types that include numeric values and strings. Examples of metrics are CPU utilization, memory usage, job status, service availability.

Operators perform equality and threshold (greater-than, less-than) comparisons. Not every operator is suitable for every metric value type.

To extend expressions several criteria can construct an expression using AND and OR logical operators. Using aggregations is another way of extending expressions which include average, sum and count.

5.5.2 The Monitoring Service Interactions

Diagram shown on figure 45 presents monitoring initialization phase. Interested parties subscribe to different notifications to receive to them important information. At some point administrator adds a new VOPS policy which gets translated to monitoring rule. Monitoring rule is then sent to Monitoring service where monitoring against it is performed.

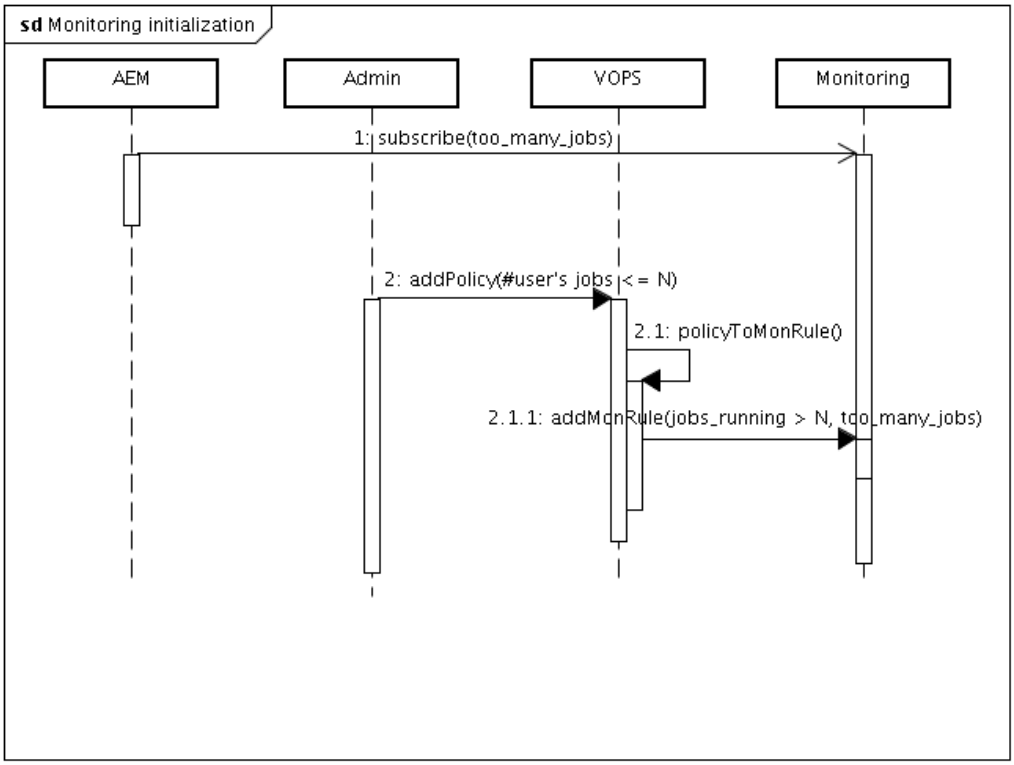


Figure 45: Monitoring initialization.

Diagram shown on figure 46 presents case when monitoring rule conditions are not met when performing monitoring rules check. AEM reports to Monitoring that some job has started (or ended) and Monitoring checks if conditions of any monitoring rules are satisfied. Because they are not, no notification is triggered.

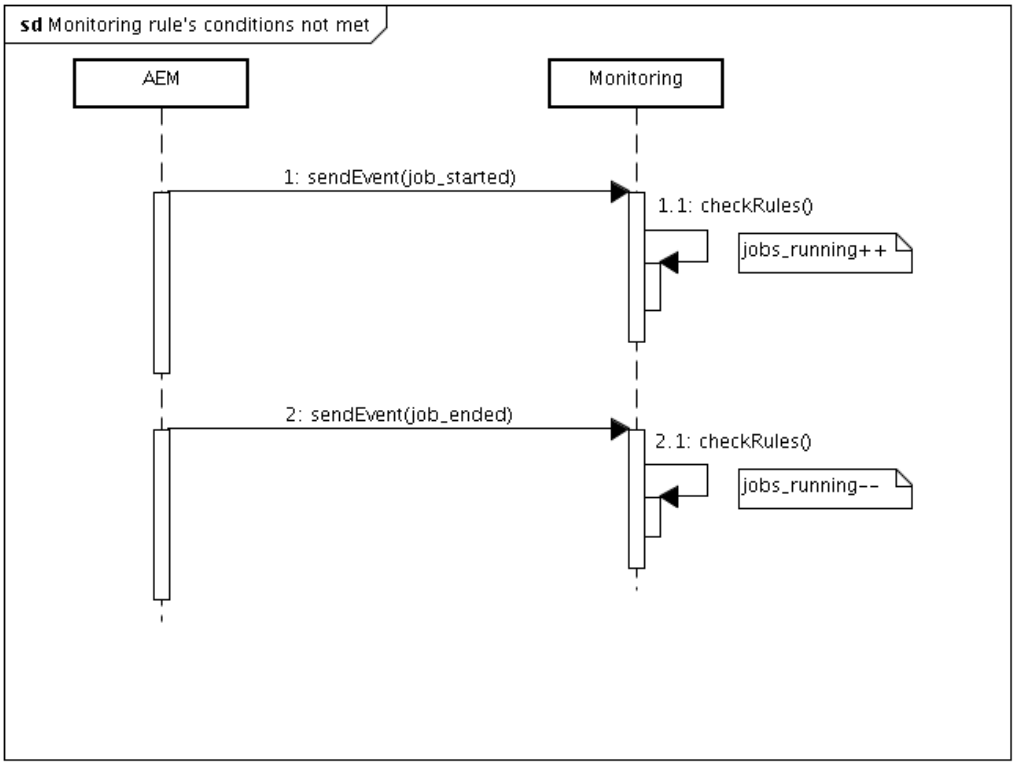


Figure 46: Monitoring rule conditions not met.

Diagram shown on figure 47 presents case when monitoring rule conditions are met when performing monitoring rules check. AEM reports to Monitoring that some job has been started and Monitoring detects too many jobs are running based on particular monitoring rule. Notification is therefore triggered which all subscribers receive.

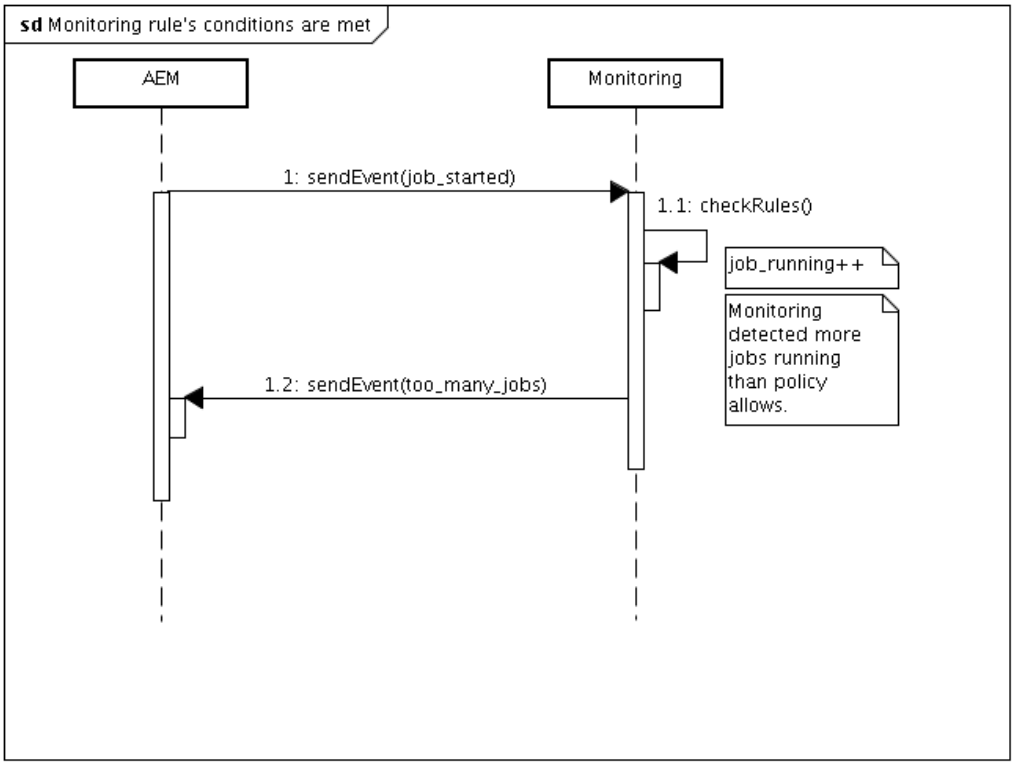


Figure 47: Monitoring rule conditions met.

5.6 Auditing Service Design

5.6.1 The Auditing Service Classes

Class shown on figure 48 represents interface to the auditing service. More detailed description of auditing manager API is shown in section A.6.

5.6.2 The Auditing Service Interactions

Diagram shown on figure 49 presents auditing archiving data. First Auditing subscribes to monitoring to receive monitoring data. When various XtreemOS services report events to Monitoring, Auditing subsequently receives events from Monitoring and stores them into historical database.

Figure 48: Auditing Manager class diagram.

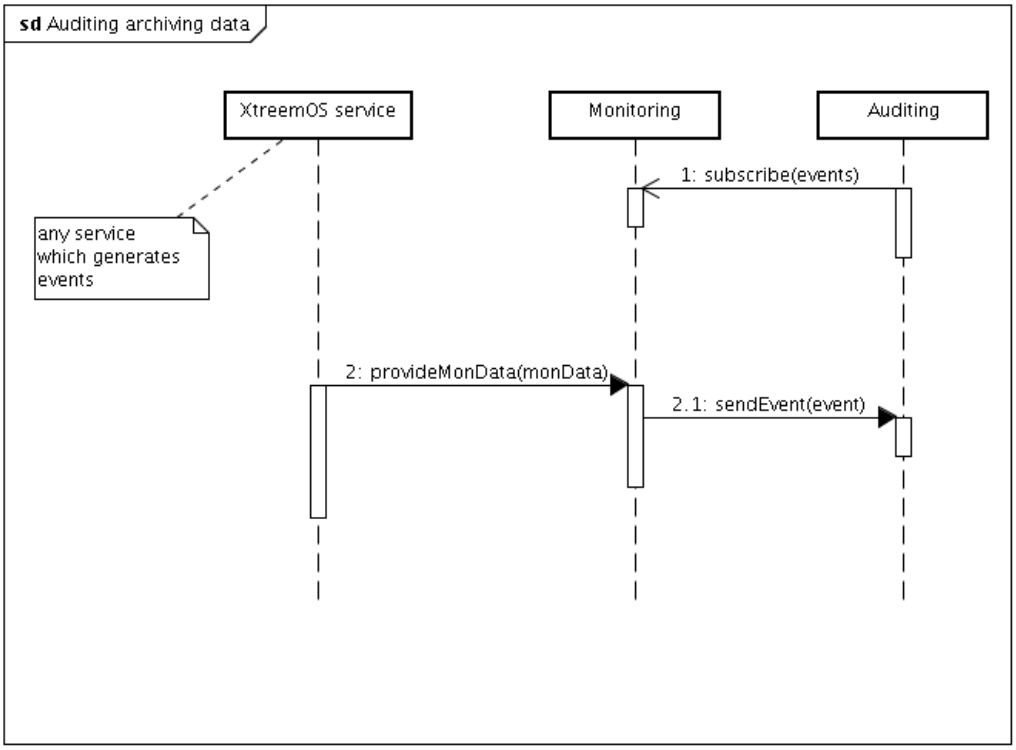


Figure 49: Auditing archiving data.

Diagram shown on figure 50 presents auditing generating report. Administrator requests Auditing to generate report of specified type. Auditing generates the report and sends it to the Administrator.

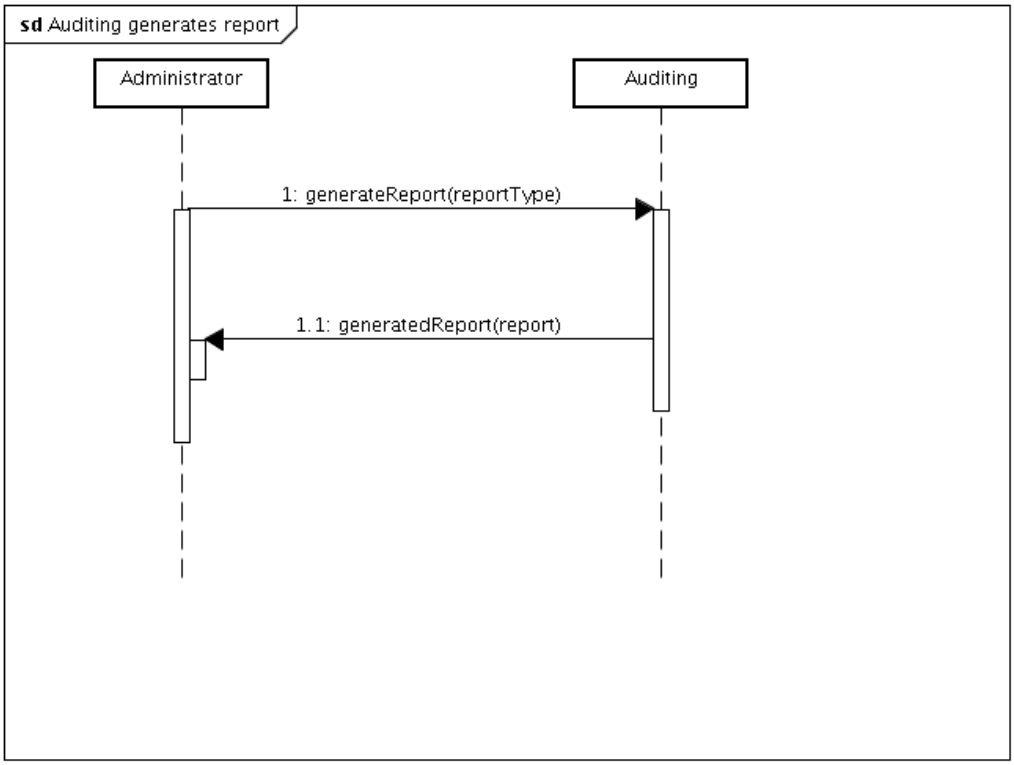


Figure 50: Auditing generates report.

6 Conclusions and Future Work

This deliverable presented the final specification, design and architecture of the security and VO management services in the XtreemOS operating system. The deliverable was based on previous specifications, D3.5.3 [1], D3.5.4 [2] and D3.5.11 [3], completed in the past two years of the lifetime of the project. The new deliverable focused on multi-layer presentation of the services from the perspectives of high-level capabilities, intermediate-level system design and low-level APIs.

The current deliverable defined additional new features of the services, which included:

- VO termination capabilities and VO state dissolution, and
- monitoring and auditing capabilities for supporting billing and non-repudiation applications.

Originally, it was planned to report in this deliverable the output of task T3.5.13 Isolation. However, we considered that the output of such a task could fit better in deliverable D3.5.9, *methodology and design alternatives for trust services in XtreemOS* [4], to be submitted in M45. We will be describing there the output of T3.5.13, where isolation is seen as a mechanism for achieving non-intereference and fine-grained control of resource usage.

References

[1] XtreemOS Consortium. First specification of security services. In *XtreemOS public deliverables - D3.5.3*. Work Package 3.5, May 2007.

[2] XtreemOS Consortium. Second specification of security services. In *XtreemOS public deliverables - D3.5.4*. Work Package 3.5, November 2007.

[3] XtreemOS Consortium. Third specification of security services. In *XtreemOS public deliverables - D3.5.11*. Work Package 3.5, January 2009.

[4] XtreemOS Consortium. Methodology and design alternatives for trust services in xtreemos. In *XtreemOS public deliverables - D3.5.11*. Work Package 3.5, February 2010.

[5] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson. Rfc 3820 - internet x.509 public key infrastructure (pki) proxy certificate profile, June 2004.

[6] Thomas D. Wu. The secure remote password protocol. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, California, USA, 1998. The Internet Society.

Appendix

A Application Programming Interface of the Security and VO Management Services

A.1 The XVOMS API

XVOMS provides two types of interfaces to other XtreemOS components: **UserUtil** - interfaces for managing users and **VOUtil** - interfaces for managing VOs.

A.1.1 User and VO Management Interfaces in XVOMS

Users access all the VO management facilities via the XvomsSession interface:

Name: `XvomsSession()`

Purpose: Empty constructor for object

Overview: This constructor is for testing and should not be called by a user.

Input: None

Output: Returns empty XvomsSession object

Throws: None

Name: `XvomsSession(String username, String passMD5)`

Purpose: Constructor for XVomsSession

Overview: This constructor takes the username and MD5 hash of the users password and checks with the XVOMS database that the user and password are valid. If so the returned object has the property authenticated set as true, otherwise this is false.

Input: `username` - string value of the user name. // `passMD5` - the MD5 hash of the users password.

Output: Returns initialised XvomsSession object

Throws: `XVOMSException` - if user expired or password invalid.
`UserNotFoundException` if user not found in XVOMS.

XvomsSession is associated with a XvomsSessionContext, which are described as follows:

Name: `XvomsSessionContext(XvomsSession xsession)`

Purpose: Constructor for XVomsSessionContext

Overview: If the user in the given session is authenticated, the context (including user, voUtil, and userUtil objects) will be properly populated. Otherwise, all these objects will remain null.

Input: `xsession` - XvomsSession to create context for.

Output: Returns XvomsSessionContext object if user is authenticated.

A.1.2 User Management methods

The **UserUtil** interface allows managing users in XVOMS. The followings are a list of methods that can be used to manage users via XVOMS:

Name: `User addUser(String realname, String username, String pass, String des)`

Purpose: create new user.

Overview: creates a new user in the users table of the XVOMS database.

Input: `realname` - user's full name
`username` - username within the grid
`pass` - password

Output: `user` - User object returned, XVOM database updated.

Throws: `XVOMSException` - if user already exists.

Name: `User addUser(String realname, String username, String pass, String des, VORole vorole, Actor actor)`

Purpose: create new user.

Overview: creates a new user in the users table of the XVOMS database. This version defines the vorole and actor of the new user.

Input: `realname` - user's full name
`username` - username within the grid
`pass` - password
`vorole` - The vorole for the new user
`actor` - The actor for the new user

Output: `user` - User object returned, XVOM database updated.

Throws: `XVOMSException` - if user already exists.

Name: **`User getUser(String username)`**

Purpose: get user from username.

Overview: Look up username in XVOMS database and return user object.

Input: `username` - username to look up.

Output: Returns user object, null if username not found.

Throws: None

Name: **`List getUsers(Actor actor)`**

Purpose: Get users with property actor

Overview: This returns a list of all users with Actor type actor.

Input: `actor` - type of actor

Output: Returns List of user objects

Throws: None

Name: **`List getUsers(VO vo)`**

Purpose: Find users in a VO

Overview: Get a list of user objects who are in the given VO.

Input: `vo` - the VO of interest.

Output: Returns a List of user objects in the given VO.

Throws: None

Name: `List getUsers(VOGroup vogroup)`

Purpose: Find users in a VOGroup

Overview: Get a List of all the users in a VOGroup

Input: `vogroup` - the VOGroup of interest.

Output: Returns a List of user objects in the given VOGroup.

Throws: None

Name: `List getUsers(VORole vorole)`

Purpose: Find users with a particular VORole

Overview: Get a List of all the users with a given VORole

Input: `vorole` - the VORole of interest.

Output: Returns a List of user objects with the given VORole.

Throws: None

Name: `void removeUser(User user,VORole vorole)`

Purpose: Remove user from database

Overview: Remove a user from the XVOMS database, given their VORole

Input: `user` - the user to remove.
`vorole` - the VORole of the user to remove.

Output: None.

Throws: None

Name: `void removeUser(User user)`

Purpose: Remove user from database

Overview: Remove a user from the XVOMS database.

Input: `user` - the user to remove.

Output: None.

Throws: None

Name: `void updateUserPass(User user, String pass)`

Purpose: Update user password

Overview: Change password for the current user, if authenticated.

Input: `user` - the user to update the password for.
`pass` - the new password.

Output: None.

Throws: None

A.1.3 System Management methods

The following methods are defined in the **VOUtil** interface. They allow a system administrator (of XVOMS) to manage user roles (also called actor) in the system. Examples of a user role at the system level are system administrator, resource administrator.

Name: `Actor addActor(String name, String des)`

Purpose: Add actor.

Overview: Add a new actor to the system.

Input: `name` - the name of the new actor.
`des` - a description of the new actor.

Output: returns Actor object.

Throws: None

Name: `void addActor2User(User user, Actor actor)`

Purpose: Add actor role to a user.

Overview: Assign the given actor role to the named user.

Input: `user` - the name of the user.
`Actor` - The actor role to add.

Output: None.

Throws: `XVOMSException` - if either input is null.

Name: `Actor getActor(String name)`

Purpose: Get Actor object from name.

Overview: Look up the name of the Actor in the XVOMS database and return the object, if it exists otherwise throw an exception.

Input: `name` - the name of the Actor.

Output: Returns the associated Actor object.

Throws: `XVOMSException` - if name does not exist.

Name: `void removeActor(Actor actor)`

Purpose: Remove Actor object from database.

Overview: Remove the Actor from the XVOMS database. The Actor should not have any associated users, otherwise an exception is thrown.

Input: `actor` - the Actor object to remove.

Output: None.

Throws: `XVOMSException` - if Actor is still associated with one or more users.

A.1.4 Resource Management methods

The following methods are defined in the **VOUtil** interface. The resource management methods allow administrator (RCA owner and Grid administrator) to manage RCAs in the system.

Name: `RCA addRCA(String name,String desc, User user)`

Purpose: Add an RCA into the XVOMS database.

Overview: The named RCA will be added to the XVOMS database with user as a resource Admin.

Input: `name` - name of the RCA resource to add.
`desc` - Description of the resource.
`user` - user object for resource Admin.

Output: Returns RCA object.

Throws: None.

Name: `RCA getRCA(String name)`

Purpose: Get an RCA from its name.

Overview: Look up the given name and return the RCA object, or null if not found.

Input: `name` - name of the RCA resource to lookup.

Output: Returns RCA object.

Throws: None.

Name: `void updateRCA(String name, String desc)`

Purpose: Update the description of an RCA.

Overview: Look up the given RCA name and update its description.

Input: `name` - name of the RCA resource to update.
`desc` - updated description of RCA.

Output: None.

Throws: None.

Name: `void addRCA2VO(RCA rca, VO vo, User user)`

Purpose: Add rca to vo by user.

Overview: The user must have the VO Admin role for the given vo.

Input: `rca` - RCA resource to add.
`vo` - VO to add resource to.
`user` - User who is VO Admin for vo.

Output: None

Throws: `XVOMSException` - if user is not authorised.

Name: `void removeRCA(RCA rca, User user)`

Purpose: Remove an rca from the XVOMS database by user.

Overview: The user must have the RCA Admin role for the given vo. The RCA must not be in any vo.

Input: `rca` - RCA resource to remove.
`user` - User who is RCA Admin for rca.

Output: None

Throws: `XVOMSException` - if user is not authorised or rca is a member of a vo.

Name: `void removeRCA2VO(RCA rca, VO vo, User user)`

Purpose: Remove rca from vo by user.

Overview: The user must have the VO Admin role for the given vo.

Input: `rca` - RCA resource to remove from vo.
`vo` - VO to remove resource from.
`user` - User who is VO Admin for vo.

Output: None

Throws: `XVOMSException` - if user is not authorised.

A.1.5 VO Management APIs

The following methods are defined in the **VOUtil** interface. These methods can be used to manage VOs via XVOMS:

Name: `Action addAction(String name)`

Purpose: Add action to XVOMS database

Overview: The action name (e.g. ADD,DELETE,GET,...) is added to the database.

Input: `name` - String for the action name.

Output: Action object.

Throws: None.

Name: `void addGroup2User(User user, VOGroup vogroup)`

Purpose: Add user to vogroup

Overview: Adds the user into the existing vogroup.

Input: `vogroup` - The vogroup to add.

Inout: `user` - The user who is updated with the new group.

Output: None.

Throws: None.

Name: `VOGroup addGroup2VO(VO vo, String desc)`

Purpose: Add a vogroup to a vo

Overview: Create a new group described by the desc string and add to given vo.

Input: `vo` - The vo that the new group will be in.

Inout: `desc` - The description of the new group.

Output: Returns a VOGroup object.

Throws: None.

Name: `Request addRequest(String desc,String type, VO targetedVO)`

Purpose: Adds a request to the user in the current context.

Overview: Record a request in the database for the current user.

Input: `desc` - Details of the request.
`type` - type of request.
`targetedVO` - the VO the request relates to.

Output: Returns a Request object.

Throws: None.

Name: `Resource addResource(VO vo, String desc, RCA rca)`

Purpose: Adds a resource to a vo.

Overview: The resource described by desc will be added to the vo from the given rca.

Input: `vo` - The vo to add the resource to.
`desc` - Details of the resource to add.
`rca` - The RCA which the resource is registered with.

Output: Returns a Resource object.

Throws: None.

Name: `VORole addRole2Group(VOGroup group, String desc)`

Purpose: Adds a vorole to a vogroup.

Overview: The vorole described by desc is added to the group.

Input: `group` - The group to add the role to.
`desc` - Description of the role to add.

Output: Returns a VORole object.

Throws: None.

Name: `void addRole2User(User user, VORole vorole)`

Purpose: Adds a vorole to a user.

Overview: The vorole is added to the user.

Input: `user` - The user to add the role to.
`vorole` - The vorole to add.

Output: None.

Throws: None.

Name: **`AccessControlRule addRule(String desc, Actor actor, Action action, String target)`**

Purpose: Adds an access control rule.

Overview: Create a new access control rule associated with a table.

Input: `desc` - Description of the rule.
`actor` - The actor the rule is related to.
`action` - The action the rule is related to.
`target` - The allowed target, defined by enum ALLOWEDTARGET

Output: Returns an accessControlRule.

Throws: `InvalidActorException`,
`InvalidActionException`,
`InvalidTargetException`

Name: **`VO addVO(String des, String name, User owner)`**

Purpose: Add a VO and set the owner.

Overview: The VO is added to the database and user is set as the owner.

Input: `des` - Description of the VO.
`name` - The name of the new VO.
`owner` - The user who will own the new VO.

Output: Returns a new VO object, null if name or owner is null.

Throws: None.

Name: **`void addVO2User(User user, VO vo)`**

Purpose: Add a VO to a user.

Overview: The user becomes a member of the existing VO.

Input: `vo` - The VO to join.

Inout: `user` - The user who will become a member of the VO.

Output: None.

Throws: None.

Name: `List getRequests()`

Purpose: Get a list of requests for the current user.

Overview: Find the set of requests pending for the current user.

Input: None (implicit use of current context).

Output: Returns a List object of current requests.

Throws: None.

Name: `List getRequests(VO vo)`

Purpose: Get a list of requests belonging to a VO.

Overview: Find all pending requests for the named VO as a List.

Input: `vo` - The VO of interest.

Output: Returns a List object of current requests for this VO.

Throws: None.

Name: `Object getObject(String name, Class aClass)`

Purpose: Generic method to get object by name.

Overview: This method is mainly intended for internal use. The object must be cast to the actual class type.

Input: `name` - The object name.
`aclass` - The object class.

Output: Returns an object which must be cast correctly.

Throws: `ObjectNotFoundException`

Name: `VO getVO(String gvid)`

Purpose: Get VO from gvid.

Overview: Find the VO from the global VO ID.

Input: `gvid` - The gvid of interest.

Output: Returns a VO object for the corresponding VO, or null if not found.

Throws: None.

Name: `List getVOAtrributes(String user)`

Purpose: Get a list of VO attributes for user name.

Overview: Returns a List of VOattribute array where each array is an ordered set of VO attributes for a user in their given VO. Arrays are such as [VOGroup, VORole].

Input: `String` - The user of interest.

Output: Returns a List of attribute arrays.

Throws: `UserNotFoundException`

Name: `List getVOAtrributes(User user)`

Purpose: Get a list of VO attributes for user.

Overview: Returns a List of VOattribute array where each array is an ordered set of VO attributes for a user in their given VO. Arrays are such as [VOGroup, VORole].

Input: `user` - The user of interest.

Output: Returns a List of attribute arrays.

Throws: None.

Name: `List getVOAtrributes(User user, VO vo)`

Purpose: Get a list of VO attributes for user in a VO.

Overview: Returns a List of VOattribute arrays as in previous methods, but for the user in the named VO.

Input: `user` - The user of interest.
`vo` - The vo of interest.

Output: Returns a List of attribute arrays.

Throws: None.

Name: `List getVOGroups(VO vo)`

Purpose: Get a list of groups in VO.

Overview: Returns a List of VOGroup for the VO.

Input: `vo` - The VO of interest.

Output: Returns a List of VOGroup.

Throws: None.

Name: `List getVORoles(VOGroup vogroup)`

Purpose: Get a list of VORole in VOGroup.

Overview: Returns a List of VORole for the VOGroup.

Input: `vogroup` - The VOGroup of interest.

Output: Returns a List of VORole.

Throws: None.

Name: `List getVOs()`

Purpose: Get a list of VOs in database.

Overview: Returns a List of all the VOs currently in the XVOMS database.

Output: Returns a List of VOs.

Throws: None.

Name: `void prettyPrint(String username)`

Purpose: Print user's attributes

Overview: Print to stdout.

Input: `username` - The username of interest.

Output: None.

Throws: None.

Name: **`void updateGroup(VOGroup group, String des)`**

Purpose: Update an existing group.

Overview: Updates the description of the group in XVOMS.

Input: `des` - The new group description.

Inout: `group` - The group to update

Output: None.

Throws: None.

Name: **`void updateRole(VORole vorole, String des)`**

Purpose: Update an existing vorole.

Overview: Updates the description of the vorole in XVOMS.

Input: `des` - The new role description.

Inout: `vorole` - The vorole to update

Output: None.

Throws: None.

Name: **`void updateVO(String gvid, String des)`**

Purpose: Update an existing VO.

Overview: Updates the description of the VO in XVOMS for the given gvid.

Input: `des` - The new role description.
`gvid` - The global ID of the VO to update.

Output: None.

Throws: None.

Name: `void removeGroup2User(User user, VOGroup vogroup)`

Purpose: Remove user from VOGroup.

Overview: Updates XVOMS database to remove the membership of user in vogroup.

Input: `user` - The user to remove.
`vogroup` - The vogroup to remove the user from.

Output: None.

Throws: None.

Name: `void removeGroup2VO(VO vo, VOGroup vogroup)`

Purpose: Remove VOGroup from VO.

Overview: Updates XVOMS database to remove vogroup.

Input: `user` - The user to remove.
`vogroup` - The vogroup to remove from the VO.

Output: None.

Throws: None.

Name: `void removeRequest(Request request)`

Purpose: Remove request from database.

Overview: Updates XVOMS database to remove request.

Input: `request` - The request to remove.

Output: None.

Throws: None.

Name: `void removeGroup2VO(VO vo, VOGroup vogroup)`

Purpose: Remove VOGroup from VO.

Overview: Updates XVOMS database to remove vogroup.

Input: `user` - The user to remove.
`vogroup` - The vogroup to remove from the VO.

Output: None.

Throws: None.

Name: `void removeRequest(Request request)`

Purpose: Remove request.

Overview: Updates XVOMS database to remove request.

Input: `request` - The request to remove.

Output: None.

Throws: None.

Name: `void removeResource(VO vo, Resource resource)`

Purpose: Remove resource from VO.

Overview: Remove resource that has been added to a VO.

Input: `vo` - The VO to remove the resource from.
`resource` - The resource to remove.

Output: None.

Throws: None.

Name: `void removeRole2Group(VOGroup group, VORolep vorole)`

Purpose: Remove vorole from vogroup.

Overview: Updates XVOMS database to remove vorole.

Input: `group` - The vogroup.
`vorole` - The vorole to remove from the VOGroup.

Output: None.

Throws: None.

Name: **`void removeRole2User(User user, VORole vorole)`**

Purpose: Remove vorole from user.

Overview: Updates XVOMS database to remove vorole from user.

Input: `user` - The user.
`vorole` - The vorole to remove from the user.

Output: None.

Throws: None.

Name: **`void removeVO(VO vo)`**

Purpose: Remove vo from system.

Overview: Updates XVOMS database to remove the vo. The vo should be empty.

Input: `vo` - The vo to remove.

Output: None.

Throws: None.

Name: **`void removeVO2User(User user, VO vo)`**

Purpose: Remove user from vo.

Overview: Updates XVOMS database to remove the user from the vo.

Input: `user` - The user to remove from the vo.
`vo` - The vo to remove the user from.

Output: None.

Throws: None.

A.2 CDA API

The CDA (Certificate Distribution Authority) client is used to request an XOS-Certificate from the CDA server. It can run on any client node, and does not use any other XtreemOS services. This section describes the API of the CDA client classes (`CdaClient` and `PeerChecker`), and how to use them in a CDA client application.

A.2.1 The CDA Client/Server Protocol

The command-response protocol for obtaining an XOS-Certificate takes two steps:

1. An authentication step sends the username and password to the server.

   ```
   AUTHENTICATE
   username,password
   ```

   If the string received in the response from the server is AUTHOK, the next step, sending the certificate request, is taken. Otherwise, the string FAILED is received from the server, followed by an error message. In this case, the CDA client should present this error message to the user, close the client connection, and exit.

2. The second step in the protocol sends a Certificate Signing Request (CSR) to the server, along with the name of user's primary VO and the name of their primary VO group.

   ```
   CERTREQUEST
   voName, voGroup
   <CSR>
   ```

The returned value is an X.509 v3 certificate containing the user's identity, public key, and VO attributes (their XOS-Certificate).

A.2.2 CDA Client

CdaClient Class The CDA client class is in the package `eu.xtreemos.security.cda.client`. It is used to communicate with the CDA server, authenticating the user and requesting an XOS-Certificate for them.

Name: CdaClient `(String host, int port, X509Certificate trustedCert)`

Purpose: Constructor to create an instance of the CdaClient class.

Overview: Make a connection to the CDA server on **host** at port **port**. Check for the server sending an identity certificate signed by the root certificate for this grid, **trustedCert**.

Input: The constructor takes the following inputs:

host The address of the host to connect to.

port The port number on the host to connect to.

trustedCert The root certificate for this XtreemOS Grid.

Output: A new instance of the CdaClient class.

The `authenticate` method of the CdaClient class sends the AUTHENTICATE command to the CDA service:

Name: `boolean` **authenticate** `(String username, char[] password)`

Purpose: Authenticate the user to the CDA server.

Overview: Sends the user's username and password to the CDA server.

Input: The **authenticate** method takes the following inputs:

username The name of the user.

password The user's password.

Output: A boolean indicating whether the user has supplied the correct details for a valid user account.

Throws: IOException, if the communication with the CDA server fails.

The method `sendCertificateRequest` command sends the CERTREQUEST command to the CDA service:

Name: `X509Certificate` **sendCertificateRequest** `(String voName, String groupName, PKCS10CertificationRequest userRequest)`

Purpose: Request an XOS certificate from the CDA server.

Overview: Sends a Certificate Signing Request for the VO **voName** and the group **groupName**

Input: The **sendCertificateRequest** method takes the following inputs:

voName The name of user's primary VO.

groupName The user's primary group.

userRequest A Certificate Signing Request containing the user's public key, signed with the user's private key.

Output: An XOS-Certificate containing the user's private key, their Global User ID and their VO attributes. This information is signed by the CDA server's private key.

Throws: IOException, if the communication with the CDA server fails.

Throws: IllegalArgumentException, if the user's CSR is malformed, or if the voName or groupName do not exist.

PeerChecker Class The PeerChecker class is in the package `eu.xtreemos.security.cda.client`. This class is used by the CDA client to check the authenticity of the CDA service. The **PeerChecker** class and its methods are used after the CdaClient object is constructed, and before the **CdaClient.authenticate** method is called.

Name: PeerChecker `(String host, PublicKey rootCAPublicKey, TrustManager[] trustManagers, boolean carryOnRegardless)`

Purpose: Set up framework to check the identify of the CDA server.

Overview: The CDA client requests SSL server authentication on the communications channel. This class is used to set up the trusted root which can be used to check the autheticity of the CDA server.

Input: The **PeerChecker** constructor takes the following inputs:

host Name of host that the CDA client is connecting to. The PeerChecker checks that the hostname in the CDA server certificate is the same as this hostname.

rootCAPublicKey The public key of the Root CA for this XtreemOS Grid. Used to check the signature on the host certificate sent by the CDA server.

trustManagers TrustManager objects used to check the identity of the CDA server.

carryOnRegardless If true, gives the option to carry on the protocol even if the CDA server mis-identifies itself.

Output: A PeerChecker object.

The following method of the PeerChecker class is used for checking when the CDA server has been authenticated.

Name: `boolean` **isReady()**

Purpose: Indicate when the CDA server has been authenticated, ensuring that the communications channel between CDA client and CDA server is secure.

Output: A boolean indicating if the CDA server has been authenticated.

To allow the CDA client to control when the handshake on the SSL connection takes place, the `startHandshake()` method of the `CdaClient` class needs to be called. The following code fragment shows the loop needed to wait until the SSL handshake has completed:

```
client.addHandshakeCompletedListener(peerChecker);
client.startHandshake();
while ( !peerChecker.isReady()) {

  try {
    Thread.sleep(1);
  }
  catch (InterruptedException ex) {
    ;
  }

}
```

The 'post-connection' checking needs to performed before any application data is written on the SSL connection. Processing the SSL handshake is done in a separate thread, so the CDA client needs to execute the loop above while the PeerChecker method `handshakeCompleted` authenticates the details sent by the CDA server in its public key certificate.

The method `boolean PeerChecker.isReady()` indicates when the checking the authenticity of the CDA server has been completed. The CDA client code has to loop until `isReady` returns `true`. The CDA client is now ready to start sending commands to the CDA service. It issues the AUTHENTICATE command by calling the method described in A.2.2. If this method returns true, the CDA client can proceed to send the CERTREQUEST command to the CDA service by the method described in A.2.2.

This CDA client application can then verify the certificate with a call to the method `verifyCertificate` in the class `CertificateProcessor`, contained in package `eu.xtreemos.security.cda.util`.

Name: `boolean` **verifyCertificate** `(X509Certificate cert, boolean verbose, PublicKey issuerPublicKey)`

Purpose: Verify an XOS-Certificate to check that it was issued by the holder of the private key corresponding to the **issuerPublicKey**.

Input: The verifyCertificate method has the following inputs:

cert: The XOS-Certificate to verify.

verbose: Flag to set level of reporting messages.

issuerPublicKey: Public key of the issuer of the XOS-Certificate, i.e. the CDA server public key. Obtained by a call to `PeerChecker.getPeerKey()`.

Output: If the XOS-Certificate is verified as authentic and valid, the return value is `true`. In this case, the CDA client application can save the XOS-Certificate via `Utils.writeCertificate` and the associated private key via `Utils.writeKey`.

A.2.3 CDA Server

The core functionality to generate XOS-Certificates is contained in the class `eu.xtreemos.security.cda.server.engine.Engine`. The first step is to use the helper class `eu.xtreemos.security.cda.server.engine.VOService` to authenticate a user before generating an XOS-Certificate.

Constructor for CDA Server Engine

Name: `Engine (PrivateKey cdaKey, X509Certificate cdaCert, X509Certificate rootCACert, String signatureAlgorithm, VOService voService, VerificationServiceInfo revokeInfo, int days, int hours, int minutes)`

Purpose: Create the CDA server engine.

Input: **cdaKey** Private key of the CDA, used to sign the user's XOS-Certificate

cdaCert Public key certificate of the CDA

rootCACert Public key certificate of the Root CA (unused in first release)

signatureAlgorithm Name of the algorithm used to provide a signature on this certificate

voService Provides an interface to the X-VOMS database

verificationServiceInfo Information on service addresses to perform on-line verification of the certificate

days Number of days for the certificate validity

hours Number of hours for the certificate validity

minutes Number of minutes for the certificate validity

Output: An instance of the CDA server engine class.

Authenticating the user The following method is used to authenticate the user with the X-VOMS database:

Name: `public boolean authenticate (String username, char[] password)`

Purpose: Check the user's details in the X-VOMS database.

Input: **username** The name of the user making the request

password The password of the user making the request

Output: The call returns `true` if the username and password are valid, `false` otherwise.

Using the CDA Server Engine to Generate XOS-Certificates The following method generates XOS-Certificates:

Name: tt public X509Certificate **generateXOSCert** (VOUser user, long serial, PublicKey userKey, int days, int hours, int minutes)

Purpose: Create the user's XOS-Certificate.

Input: **user** User object with the requested VO and group as primary VO and primary group

serial A unique serial number for the new XOS-Certificate

userKey The user's public key

days Number of days for the certificate validity

hours Number of hours for the certificate validity

minutes Number of minutes for the certificate validity

Output: The return value is the user's XOS-Certificate, in the form of an X509 v3 certificate containing the user identity (the CN field contains the user's Global User Identifier), public key, and their VO attributes.

A.3 The RCA API

In this section we detail the APIs of the RCA's components, as they are exposed towards the message bus, e.g., for the DIXI services or the XATI clients.

A.3.1 RCA Server

The `RCAServer` class resides in the `eu.xtreemos.xosd.vo.rca.server.RCAServer` package. It implements the service, which has the main purpose of signing the resource's identity certificate public key, and provide a signed attribute certificates to the resource. The service also provides access to the RCA database that keeps the collections of the registered resources.

Name: `ArrayList<ResourceDescriptorRecord>` **getPendingResources**`( X509Certificate userCert )`

Purpose: Returns a list of resource descriptions describing the resources listed in the RCA DB as pending for registration.

Input: **userCert** The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: A collection of resource descriptors of resources applied for registration with the RCA.

Name: `ArrayList<ResourceDescriptorRecord>` **getRegisteredResources** `( X509Certificate userCert )`

Purpose: Returns a list of resource descriptions describing the resources listed in the RCA DB as registered.

Input: **userCert** The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: A collection of resource descriptors of resources registered with the RCA.

Name: `public Integer` **confirmRegistration** `( X509Certificate userCert, ResourceID id )`

Purpose: Confirm the registration of a resource that has previously been applied for the registration using `applyForRegistration`. After this call, the RCA will sign certificates for the registered resource (`requestCertificate`).

Input: **id** The id of the resource record signifying the resource to be confirmed for the registration.

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: 0 if the call was successful.

Name: `Integer` **getResourceStatus** `( ResourceID id, X509Certificate userCert )`

Retrieves the current status of the resource according to the RCA DB.

Input: **id** The identifier of the resource to check the status of.

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: The status of the resource.

Name: `public ResourceDescriptorRecord` **unregisterResource** `( ResourceID id, X509Certificate userCert )`

Purpose: Remove the resource from the list of registered resources. Once the resource has been unregistered, it cannot have the machine certificates signed by the RCA.

Input: **id** The id of the resource record signifying the resource to be removed for the list of registered resources.

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: 0 if the call was successful.

Name: `public Integer` **applyForRegistration** `( ResourceDescriptorRecord resource, X509Certificate userCert )`

Purpose: Put the resource on the list of resources that can be registered, but need to wait for an authorised administrator to confirm the registration using the confirmRegistration call before the resource can have its certificates signed by the RCA.

If the resource is already on any of the lists, then their entry gets replaced with the new value, thus updating the application or the registration.

Input: **resource** The descriptor of the resource applying for the registration.

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: 0 if the call was successful.

Name: `public Integer` **updateResource** `( ResourceDescriptorRecord resource, X509Certificate userCert )` *throws* `Exception`

Purpose: For an already registered resource, update the resource description in the RCA DB.

Input: **resource** The descriptor of the resource with the updated data. The descriptor should contain the proper resource's ID.

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: 0 if the call was successful.

Throws: `ResourceNotRegisteredException` if the resource is not registered.

Name: `RCASignedResponse` **requestCertificate** `( ResourceID id, PKCS10CertificationRequest certRequest, X509Certificate userCert )` *throws* `Exception`

Purpose: Serves the client's request for signing the certificate. The method retrieves the data on the resource from the RCA DB, and uses the descriptor and the data in the config file to set up the attributes of the certificates that will be returned signed. The resource has to be a member of the registered resources, i.e. successful calls to `applyForRegistration` and `confirmRegistration` have to precede this call.

Input: **id** The id of the resource that requests the certificate signature.

certRequest The certificate signature request.

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: The signed certificates: identity certificate, attribute certificate, the type of which depends on the configuration.

Throws: IllegalArgumentException Thrown when the certificate request is invalid.

Throws: `ResourceNotRegisteredException` if the resource is not registered.

Name: requestVOCertificate`( ResourceID id, X509Certificate certificate, String vo, X509Certificate userCert )` *throws* `IllegalArgumentException, BadResourceException`

Purpose: Request the retrieval of the signed resource attribute certificate authorizing the resource for the membership in the given VO.

Input: id The id of the resource that requests the certificate signature.

certificate The public machine certificate of the resource.

vo The ID of the VO the resource is requesting the certificate for.

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: The signed resource certificate containing the attributes authorizing the resource to be used within the VO.

Throws: BadResourceException The resource is not a member of the VO.

IllegalArgumentException Thrown when the certificate request is invalid.

Name: `Integer` **setVOMembership**`( String vo, Boolean setMember, X509Certificate userCert )` *throws* `ResourceNotRegisteredException`

Purpose: Manipulates the membership of this RCA in the VO.

By adding the RCA into a VO, all the registered resources are also notified about being added into the VO. Similarly, when the RCA is removed from a VO, the registered resources are notified as well.

Input: **vo** The ID of the VO that is the object of the call.

setMember True for adding the RCA into the VO, or false otherwise.

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: 0 if the call was successful.

Throws: `ResourceNotRegisteredException` if the resource is not registered.

Name: `Integer` **notifyVOMembershipChange**`( ResourceID id, String vo, Boolean addition, X509Certificate userCert )` *throws* `ResourceNotRegisteredException`

Purpose: Lets manipulate with the resource's membership of a VO. The method can set and unset the VO which the resource belongs to. The resource has to be on a registered resources list.

Input: **id** The identification of the resource we are setting the membership in a VO of.

vo The VO to set the membership in.

addition If true, the membership will be set, and if false, the VO will be removed from the list of VOs the resource is a member of.

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Throws: `ResourceNotRegisteredException` if the resource is not registered.

Output: 0 if the call was successful.

A.3.2 RCA Client

The `RCAClient` class resides in the `eu.xtreemos.xosd.vo.rca.client.RCAClient` package. It implements the service, which runs on each node that is capable of providing services or resources to Virtual Organisations (VO). The service is the node's counterpart of the RCA server, providing a convenient way to store and access the local machine certificates, gather information on the resource (e.g. from the local `ResourceMonitor` service), and it also generates new public/private key pair, the former of which it then sends to the `RCAServer` for signing.

Please note that the purpose of the service is to provide the information on the context of the particular node that hosts the service. This means that the service is not suitable to be run or used from RCA front-ends such as the RCA Web. Services, such as VOPS, can use its API calls to obtain information on the certificates installed on a particular remote node (resource).

If a service or a program (such as RCA Web) is implementing functionality of the client to the RCA Server service that will be deployed outside the target nodes, it is better to use the `RCAClientProcessor` class functionality, explained in A.3.3.

Name: `public Integer` **applyForRegistration** `( X509Certificate userCert )`

Purpose: Collects the node's resource data and submits them to RCA server to apply for registration.

Output: 0 if the call was successful.

Input: **userCert** he user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Name: `public Boolean` **requestNewCertificate** `( X509Certificate userCert ) throws Exception`

Purpose: The method generates a new public and private key pair, then sends the public key for certification to the RCA server and obtains the signed attribute certificate. It stores the certificates into the local truststore.

Input: **userCert** The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: True if the request was successful, or false otherwise.

Input: **userCert** he user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Name: `public Boolean` **requestAttributeCertificate** `( String vo, X509Certific userCert )` *throws* `Exception`

Purpose: Requests the resource's attribute certificate providing credentials for the given VO from the RCA server. It also installs the new certificate if the request succeeds.

Output: True if the request succeeded by obtaining the certificate from the RCA server, and installing it.

Input: **vo** The ID of the VO the resource is requesting the certificate for.

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Name: `X509Certificate` **getMachineIdentityCertificate** `( X509Certificate userCert )`

Purpose: Retrieve the node's machine identity certificate.

Input: **userCert** The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: The certificate currently stored on the node and used by the machine to represent its identity.

Name: `RCASignedResponse` **getMachineAttributeCertificate** `( String vo, X509Certificate userCert )`

Purpose: Retrieve the machine's attribute certificate. The caller can select whether the attribute certificate to be retrieved is a general (non-VO) attribute certificate, or an attribute certificate related to the VO.

Input: **vo** The ID of the VO the call is to retrieve the attribute certificate of. Use null or an empty string to obtain the non-VO certificate.

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: The storage of attribute certificates. It contains only one of the attribute certificates, depending on which one is currently installed on the node. If both types are installed, then the method selects the one that expires later. Returns null if the certificate is not available.

Name: `Integer` **pushVOAttributeCertificate**`( RCASignedResponse certResponse, X509Certificate userCert )`

Purpose: Lets the RCA Server service push one or more machine's VO attribute certificates that can be installed and used on the local node.

Input: **certResponse** The object containing the pushed certificate(s).

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: 0 if the call was successful.

Name: `Integer` **removeVOAttributeCertificate**`( String vo, X509Certificate userCert )`

Purpose: Lets the RCA Server remove an attribute VO certificate, notifying the client about removal from the VO.

Input: **vo** The name of a VO that the resource has been removed from.

userCert The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: 0 if the call was successful.

Name: `public String` **getMachineCertificateDetails** `( X509Certificate userCert )` *throws* `Exception`

Purpose: Opens the certificates currently stored locally and signed by RCA, and returns a string containing the details on the certificate.

Input: **userCert** The user's credential, used for limiting the access control to the administrators. Clients such as the web interface server code should use their appropriate server certificates.

Output: The details on the currently stored and used certificate.

A.3.3 RCA Client Processor

Name: **RCAClientProcessor** `( String serviceCertFile )` *throws* `IOException, FileNotFoundException`

Purpose: Class constructor. It initialises the class.

Input: **serviceCertFile** The path to the file containing the RCA service certificate.

Output:

Name: `void init(String serviceCertFile)` *throws* `IOException, FileNotFoundException`

Purpose: Initialises the class using the new service certificate. Not required if the class is initialised with the constructor with the valid service certificate file in the parameter.

Input: **serviceCertFile** The path to the file containing the RCA service certificate.

Name: `KeyPair` **generateKeyPair**`( int keylen )` *throws* `NoSuchAlgorithmException`

Purpose: Generates a new pair of private key and public key.

Input: **keylen** The length of the key (e.g., 1024).

Output: The representations of the newly generated key pair.

vspace1cm

Name: `void` **savePrivateKey**`( PrivateKey key, char[] passphrase, String keyFileName )` *throws* `IOException, FileNotFoundException`

Purpose: Saves the contents of the private key to a file.

Input: **key** the keypair to save.

passphrase the password to use when encrypting the private key.

keyFileName the name of the target file.

vspace1cm

Name: `PKCS10CertificationRequest` **createCertRequest**`( KeyPair keyPair )` *throws* `InvalidKeyException, SignatureException, Exception`

Purpose: Create a request for signing the certificate containing the provided public key.

Input: **keyPair** the key pair containing the public key that is to be used in the certificate, and the private key the request will be signed with.

Output: The representation of the certification request.

vspace1cm

Name: `boolean` **verifyCertificate**`( X509Certificate cert )`

Purpose: Verify the certificate.

Input: **cert** the certificate to be verified.

Output: If the certificate verifies successfully, the method returns `true`, otherwise it returns `false`

vspace1cm

Name: `void saveCertificate(X509Certificate cert, String fileName)` *throws* `FileNotFoundException, IOException`

Purpose: Saves the certificate into a file on disk. Can be used for machine identity certificates and attribute certificates which use the v2 certificate, which carries the attribute values in the certificate extensions.

Input: **cert** the certificate to be saved into the file.

fileName the name of the file that will contain the certificate.

vspace1cm

Name: `void saveCertificate(X509AttributeCertificate cert, String fileName)` *throws* `FileNotFoundException, IOException`

Purpose: Saves the v3 attribute certificate into a file on disk.

Input: **cert** the attribute certificate to be saved into the file.

fileName the name of the file that will contain the certificate.

vspace1cm

Name: `void` **saveVOCertificate** `( X509Certificate certificate, String path )` *throws* `FileNotFoundException, IOException, BadAttributeC`

Purpose: Saves the VO attribute certificate into the given folder. The file name of the certificate file is generated depending on the VO information stored within the certificate.

Input: **certificate** The certificate containing the attributes of the VO.

path The path to store the certificate into. The path needs to exist before calling the method.

vspace1cm

Name: `void` **saveServiceCertificate** `( X509Certificate certificate, String path )` *throws* `BadAttributeCertificateException, FileNotFoundException, IOException`

Purpose: Save the service certificate.

Input: **certificate** The certificate to be saved

path Certificate's target path.

vspace1cm

Name: `String` **generateVOCertFileName**`(String vo, boolean v3cert)`

Purpose: A helper method for generating the file name of the VO certificate.

Input: **vo** The ID of the VO.

v3cert Indicates whether the certificate to be saved is formatted as a v3 attribute certificate (`true`) or a v2 certificate with attributes in extensions (`false`).

vspace1cm

A.4 The VOPS API

VO Policy Service executes policy decisions on two levels: *core site* and *resource site*. On core site, VOPS serves to requests from other core level services, meanwhile on resource site it is used to serve request to resource selection mechanism running on resource nodes.

VOPS API can be viewed also as three type of access points providing different functionalitites: *Policy Administration Point* (PAP), *Policy Decision Point* (PDP) and *Policy Information Point* (PIP):

- PAP comprises of *setPolicy()*, d*elete()*,
- PDP comprises of *accessRequest()* methods on core site and resource site,
- PIP comprises of *listPolicies()*, *getPolicy()*, *getFilterPolicy()*.

A.4.1 Core site

List policy Lists all policies in the context of a VO user or VO administrator. This method is called when specific user wants to list her policies, policies based on GUID from userCtx are filtered from the database and only those are returned.

```
Vector<String> listPolicies(X509Certificate cert,
    Vector<String> args) throws XMLDBException
```

Input parameters

cert user or VO administrator certificate

args which parameters will be in the XML returned as a result. Arguments are passed as strings presenting xpaths to the fields in XACMLs, e.g. */Policy/Description*.

Returns A vector of policies in XML format.
Throws Instance of XMLDBException

Modify policy Modifying policies consists of combination of *getPolicy()* and *setPolicy()* methods. To modify policy, first *get()* method should be used to obtain policy in consideration. After the policy is changed locally (with API provided by class in package *eu.xtreemos.security.vops.xacml.policy*), *setPolicy()* method should be used to update the changes.

Remove policy Remove policy from storage with **id** given as parameter.

```
void delete(String id) throws XMLDBException
```

Input parameters

id policy id obtained with *listPolicies()* command

Throws Instance of XMLDBException

Set policy and create new policy Replace policy with the new value. In case if **id** equals to *null*, new policy in the database is created.
Throws Instance of XMLDBException

```
String setPolicy(String id,String value,X509Certificate cert)
    throws XMLDBException
```

Input parameters

id policy id or *null*

value policy in XACML format

userCtx subject's certificate

Throws Instance of XMLDBException.

Get Policy Return complete policy in XACML format. Gets policy from eXist's policy DB. Policy is defined by **id** which is unique in the entire set of policies. And obtained with **listPolicies()** method.

```
String getPolicy (String id) throws XMLDBException
```

Input parameters

id policyId

Returns The policy in xacml format.
Throws Instance of XMLDBException.

Get Filter Policy Returns filter policy which can be simply used in a local decision point (e.g. on a node). Policy filters can be cached (history of policies used) to speed up decision process (feature not yet implemented). The search through the database depends on **type** input parameter. If **type** equals to *VOPSConstants.PolicyType.anyPolicyType*, both user policy database and VO policy database are searched for final filter policy. If **type** equals to *VOPSConstants.PolicyType.voPolicyType* only VO policy database is searched for final policy. If **type** equals to *VOPSConstants.PolicyType.userPolicyType* only user policy database is searched for final policy.

```
String getFilterPolicy(String jsdl, X509Certificate subjectCtx,
VOPSConstants.PolicyType type)
```

Input parameters

jsdl job description

subjectCtx subject's certificate (VO user or VO admin)

type defines the type of the search through the database

Returns The filtered policy in xacml format.

Request policy decision Used by AEM framework to check if resources listed in comply with policies stored in VO policy storage. This method can also be used to check the access request based only on filtered policy passed as parameter.

```
ResponseCtx accessRequest(String jsdl, X509Certificate userCtx,
    X509Certificate resourceCtx, String filteredPolicy)
```

Input parameters

jsdl job description

userCtx subject's certificate (VO user or VO admin)

resourceCtx defines the type of the search through the database

filteredPolicy if the content is null, policies in the eXist policy storage is used to check the access request.

Returns Instance of *ResponseCtx* which carries the information if the access request is allowed or denied.

Throws Instance of XMLDBException.

A.4.2 Resource site

Request for access to specific node.

```
Result accessRequest(String jsdl, X509Certificate subjectCtx,
    X509Certificate resourceCtx, ActionConstants action,
    String filteredPolicy, boolean filteredPolicyOnly)
    throws NodePDPException
```

Input parameters

jsdl job description

subjectCtx subject's certificate (VO user or VO admin)

resourceCtx defines the type of the search through the database

action action which is taken by the user (e.g. submission). Can be one of *AEM-Create*, *JobAEMExit*, or *JobAEMSubmitJob*

filteredPolicy filtered policy used in checking the access request, provided by the VOPS.

filteredPolicyOnly true if usage of only filtered policy. When set to false, also local storage is taken into account. Local storage resides where configuration entry *rsspdp.storage* is set to.

Returns Sun's implementation of the result when using Sun's XACML PDP. To obtain decision value, use *result.getDecision()*. It returns a value denoting the decisions: *Result.DECISION_PERMIT*,*Result.DECISION_DENY*. It carries the information if the access request is allowed or denied.

Throws Instance of XMLDBException.

A.5 The Monitoring Service API

Name: query

Purpose: Queries monitoring to retrieve various data about the system.

Overview: Hierarchy of nodes which represents monitoring rule is built and query is performed on that tree of nodes.

Input: String monitoringRule - Monitoring rule describing what data to retrieve.

Input: X509Certificate userCert - The user's credential.

Returns: Object - object which represents result of the query. Type of the result depends on the monitoring rule which can be in various types e.g. float, integer, string, etc.

Throws: Exception in case monitoring rule syntax is not correct.

Name: addTimedNotification

Purpose: Monitoring rule describing what data to retrieve.

Overview: CronDaemon is used to periodically perform querying.

Input: String monitoringRule - Monitoring rule describing what data to retrieve.

Input: Integer period - Delay between two queries in milliseconds.

Input: String monRuleName - Name of the monitoring rule which can be later subscribed to for receiving notifications.

Input: X509Certificate userCert - The user's credential.

Throws: Exception in case monitoring rule syntax is not correct.

Name: addNotification

Purpose: Adds monitoring rule. Callback is triggered when conditions in monitoring rule are met. Monitoring rule must therefore be made up of condition(s) in order for monitoring rule to be accepted.

Overview: Hierarchy of nodes is built and notification is triggered when conditions are met.

Input: String monitoringRule - Monitoring rule containing conditions when to trigger callback.

Input: String monRuleName - Name of the monitoring rule which can be later subscribed to for receiving notifications.

Input: X509Certificate userCert - The user's credential.

Throws: Exception in case monitoring rule syntax is not correct or monitoring rule is not made up of condition(s).

Name: existsMonRuleName

Purpose: Checks if monitoring rule name exists.

Overview: Searches for specified monitoring rule name.

Input: String monRuleName - Monitoring rule name

Input: X509Certificate userCert - The user's credential.

Returns: Boolean - true if monitoring rule exists; otherwise false.

Name: cancelNotification

Purpose: Cancels notification.

Overview: Searches for specified notification identified by specified monitoring rule name and removes it from the list.

Input: String monRuleName - Monitoring rule name identifying notification to be canceled.

Input: X509Certificate userCert - The user's credential.

Throws: Exception in case notification ID does not exist.

Name: subscribe

Purpose: Used for subscribing to the named monitoring rule to receive notifications.

Overview: Adds callback to the list of callbacks that are associated with specified monitoring rule name.

Input: String monRuleName - Name of the monitoring rule to subscribe to and receive notifications.

Input: ICallback result - Callback that is triggered when message to channel is sent.

Input: X509Certificate userCert - The user's credential.

Returns: String - ID of the subscription which can be later used to unsubscirbe.

Name: unsubscribe

Purpose: Used for unsubscribing from named monitoring rule to stop receiving notifications.

Overview: Searches for specified subscription and removes it form the list.

Input: String subscriptionId - ID of the subscription to unsubscribe from.

Input: X509Certificate userCert - The user's credential.

Throws: Exception in case subscription ID does not exist.

Name: provideMonData

Purpose: Used to send monitoring data to the monitoring. Used by other services to provide monitoring with monitoring data.

Overview: Stores provided monitoring data to internal database.

Input: MonData monData - Object that represents monitoring data.

Input: X509Certificate userCert - The user's credential.

A.6 The Auditing Service API

Name: addArchiveRule

Purpose: Adds archive rule which describes what and when to archive. To describe what to archive, monitoring rule is used. Specified monitoring rule must be made up of condition(s) in order for monitoring rule to be accepted. Data is archived when conditions in monitoring rule are met.

Overview: Creates hierarchy of nodes which represent monitoring rule and archives specified metrics to the historical database.

Input: String monitoringRule - Monitoring rule describing what data to archive.

Input: List<Metric> metricsToArchive - List of metrics to be archived.

Input: X509Certificate userCert - The user's credential.

Returns: String - archive rule ID which can be later used for archive rule cancellation.

Throws: Exception in case monitoring rule syntax is not correct.

Name: addTimedArchiveRule

Purpose: Adds archive rule which describes what to archive in timed intervals.

Overview: CronDaemon is used to periodically perform archiving.

Input: String monitoringRule - Monitoring rule describing what data to archive.

Input: Integer period - Delay between two archiving occurrences.

Input: List<Metric> metricsToArchive - List of metrics to be archived.

Input: X509Certificate userCert - The user's credential.

Returns: String - archive rule ID which can be later used for timed archive rule cancellation.

Throws: Exception in case monitoring rule syntax is not correct.

Name: cancelArchiveRule

Purpose: Cancels archive rule meaning it is not being used anymore.

Overview: Searches for specified archive rule and removes it form the list.

Input: String archiveRuleId - ID of the archive rule.

Input: X509Certificate userCert - The user's credential.

Throws: Exception is case archive rule ID does not exist.

Name: query

Purpose: Gets a list of records containing data archived by specified archive rule.

Overview: Queries historical database and retrieves records.

Input: String archiveRuleId - Archive rule ID to get database records from.

Input: X509Certificate userCert - The user's credential.

Returns: List<Record> - list of database records containing archived data.

Throws: Exception in case archive rule ID does not exist.

Name: queryByTime

Purpose: Gets a list of records containing data archived by specified archive rule in timed intervals.

Overview: Queries historical database and retrieves records.

Input: String archiveRuleId - Archive rule ID to get database records from.

Input: TimeInterval timeInterval - Time interval determining time scope of returned records.

Input: X509Certificate userCert - The user's credential.

Returns: List<Record> - list of database records containing archived data.

Throws: Exception in case archive rule ID does not exist.

Name: generateReport

Purpose: Generated report from archived data.

Overview: Queries historical database and generates report.

Input: ReportType reportType - Type of the report to be generated.

Input: TimeInterval timeInterval - Time interval determining time scope of the report.

Input: X509Certificate userCert - The user's credential.

Returns: String - generated report.

A.7 The VOWeb and RCAWeb Front-end Interfaces

A.7.1 User login

Description User logins with username and password before use the Web Front-end

Input parameters user name, password

Returns: Print successful message if user is valid. Print failure message if user has not been approved or expired.

Throws: VOException.

Dependent APIs:

- `Database SQL`: Check username and password from xvoms DB
- `XVOMS.getActor`: Retrieve the actor for the given user

A.7.2 Create an account

Description Create a new account in VOLife

Input parameters: user name, password, re-type password, first name, last name, organization, email

Returns: Print successful message and a mail will be sent to the email address given by user if user's information has been checked and approved. Print failure message if user's information is wrong.

Throws: Exception.

Dependent APIs:

- `XVOMS.addUser(realname,username,...)`: Add user to XVOMS

A.7.3 Create a VO

Description Create a VO

Input parameters: VO name, VO description

Returns: Print successful message if VO is created. Print failure message if VO is not created.

Throws: VOException.

Dependent APIs:

- `XVOMSUtil.createVO(name,desc,owner)`: create a VO to XVOMS
- `XVOMSUtil.addUser(user,vo)`: add the user owner to the created VO

A.7.4 Join/Leave a VO

Description create a vo join/leave request

Input parameters: VO selected by the user

Returns: Print successful message if user creates the join/leave request to the VO successfully. Print failure message if user fails.

Throws: VOException.

Dependent APIs:

- `XVOMS.addRequest(desc,type,vo)`: create a request in the given VO

A.7.5 My Pending Requests

Description Listing all the requests in the vos owned by the current user in the session.

Input parameters: N/A
Returns: Show the user's requests in web interface
Throws: VOException.
Dependent APIs:

- `XVOMS.getRequests()`: gets a list of requests belonging to the current user in the session

A.7.6 Approve/Decline A Request

Description Approve/Decline a pending request belonging to the current user in the session.

Input parameters: reqid- the Id of the pending request
Returns: Refresh the list of pending requests
Throws: VOException.
Dependent APIs:

- `XVOMS.approveRequests(request)`: Approve a pending request
- `XVOMS.declineRequests(request)`: Decline a pending request
- `XVOMS.removeRequests(request)`: remove a pending request

A.7.7 Get an XOS-Cert

Description Download the user certificate

Input parameters: N/A
Returns: Print successful message if XOS-Cert is downloaded successfully. Print failure message if operation fails.
Throws: VOException.
Dependent APIs:

- `N/A`

A.7.8 Generate new Keypair

Description: Generate the XOS certificate

Input parameters: passphrase, retype-passphrase
Returns: Print successful message if new Keypair is generated successfully. Print failure message if operation fails.

Throws: VOException.
Dependent APIs:

- `CDA.newKey(file,passphrase)`:Create a new key pair and save it in file for current user

A.7.9 About me

Description: Show basic information about the current user in the session.
Input parameters: N/A
Returns: Show username, guid, and volume information of the current user in the session.
Throws: VOException.
Dependent APIs:

- `XVOMS.getGUid(user)`: Get the global uid for the given user
- `XVOMS.getUservolume(user)`: Get the xtreemfs volume information for the user

A.7.10 Change Password

Description: Change the password of the current user in the session
Input parameters: old password, new password, confirmed password
Returns: Print successful message if password is changed successfully. Print failure message if fail or old password is not correct or new password is not match with confirmed password.
Throws: VOException.
Dependent APIs:

- `User.setPassword(md5passwd)`: set the new password for the current user
- `XVOMS.getUser(user, passwd)`: return User object by giving the username and password

A.7.11 Logout

Description: Logout volife and terminate the user session.
Input parameters: N/A
Returns: Set username and guid to null, return to main interface of VOlife
Throws: N/A
Dependent APIs:

- `N/A`

A.7.12 My Owned VOs

Description List all the vos owned by the current user in the session.

Input parameters: search keywords

Returns: List VOs which names contain search keyword. If no keywords, then list all VOs owned by the current user.

Throws: VOException

Dependent APIs:

- `XVOMS.getOwnedVOs(user)`: get all the VOS owned by the given user

A.7.13 Delete a VO

Description: Terminate a VO including the requests, resources, vogroups, users.

Input parameters: VO selected in the list of the user's owned VOs.

Returns: Print successful message if selected VO is deleted successfully. Print failure message if operation fails.

Throws: VOException.

Dependent APIs:

- `N/A`

A.7.14 Manage groups/roles

Description: Add/Delete VO groups and roles

Input parameters: vo– the vo to/from which the group/role is added/deleted group – the group to add/del role – the role to add/del

Returns: Show the user's VO list with group information and role information

Throws: VOException.

Dependent APIs:

- `XVOMS.addGroup2User(user,vogroup)`: gives a user a particular vogroup.
- `XVOMS.addGroup2VO(vo, des)`: add a vogroup to a vo.
- `XVOMS.addRole2Group(group, des, role)`: add a vorole to a vogroup.
- `XVOMS.addRole2User(user, vorole)`: gives a user a particular voroleaddGroup2VO(vo, des).
- `XVOMS.removeGroup2User(user, vogroup)`: removes user from a particular vogroup.

- `XVOMS.removeGroup2VO(vo, vogroup)`: removes vogroup from vo.
- `XVOMS.removeRequest(request)`: removes request from the system.
- `XVOMS.removeResource(vo,resource)`: removes resources from an existing vo.
- `XVOMS.removeRole2Group(group, vorole)`: removes vorole from group.
- `XVOMS.removeRole2User(user, vorole)`: removes vorole for a particular user.

A.7.15 Add a RCA

Description Register a RCA server.

Input parameters: RCA name, RCA description, DIXI host, DIXI port

Returns: Print successful message if RCA is added successfully. Print failure message if operation fails.

Throws: VOException.

Dependent APIs:

- `XRCAServer.setVOMembership`: Add a RCA to VO

A.7.16 Delete a RCA

Description Delete a RCA server.

Input parameters: the RCA server to delete

Returns: Print successful message if RCA is deleted successfully. Print failure message if operation fails.

Throws: VOException.

Dependent APIs:

- `XRCAServer.setVOMembership`: Delete a RCA from VO

A.7.17 Add a resource to RCA

Description Add a resource to the RCA.

Input parameters: user guid, resource name, RCA, host, port, architecture, os, cpus, ram, speed

Returns: Print successful message if resource is added successfully. Print failure message if operation fails.

Throws: VOException.
Dependent APIs:

- `XRCAServer.getPendingResources`: Get a list of resources

A.7.18 Delete a resource

Description Remove a resource from the RCA.
Input parameters: the resource to delete
Returns: Print successful message if resource is deleted successfully. Print failure message if operation fails.
Throws: VOException.
Dependent APIs:

- `XRCAServer.getPendingResources`: Get a list of resources.
- `XRCAServer.unregisterResource`: Remove a resource from RCA.

A.7.19 Add a resource to VO

Description Request to add a resource to a VO.
Input parameters: the resource to add
Returns: Print successful message if request is generated successfully. Print failure message if operation fails.
Throws: VOException.
Dependent APIs:

- `N/A`

A.7.20 Approve a resource

Description Approve the request of a resource to add to a VO.
Input parameters: the resource request
Returns: Print successful message if request is approved successfully. Print failure message if operation fails.
Throws: VOException.
Dependent APIs:

- `XRCAServer.getPendingResources`: Get a list of resources
- `XRCAServer.confirmRegistration`: Confirm the resource request

A.7.21 Decline a resource

Description Decline the request of a resource to add to a VO.

Input parameters: the resource request

Returns: Print successful message if the request is declined successfully. Print failure message if the operation fails.

Throws: VOException.

Dependent APIs:

- `XRCAServer.getPendingResources`: Get a list of resources
- `XRCAServer.unregisterResource`: Reject the resource

A.7.22 Get Machine Certificates

Description Retrieve the node's machine identity certificate.

Input parameters: userCert- The user's credential, used for limiting the access control to the administrators

Returns: The certificate currently stored on the node and used by the machine to represent its identity

Throws: N/A

Dependent APIs:

- `RCA.getMachineIdentityCertificate(userCert)`:Retrieve the node's machine identity certificate