



UVOS MANUAL

UNICORE Team

Document Version:	1.4.1
Component Version:	1.4.1
Date:	01 05 2011

This work is co-funded by the EC EMI project under the FP7 Collaborative Projects Grant Agreement Nr. INFSO-RI-261611.

This work was co-funded by the EC Chemomentum project under the FP6 Grant Agreement Nr. IST-033437.

Contents

1	Introduction	1
2	UVOS Overview	2
2.1	Example	2
3	Entities	3
3.1	Which type of identities shall be used?	3
4	Attributes	4
5	Authentication	5
6	UVOS access authorization	5
6.1	Authorization overview	6
6.2	The simple (default) authorization scheme	8
7	VO registrations (applications)	9
8	Email notification	9
9	UVOS usage scenarios	10
9.1	PULL authorization	10
9.2	PUSH authorization	11
9.3	Web portal authentication	12
10	Installation	12
10.1	Installation from the archive	13
10.2	Installation from the RPM package (RedHat distributions)	13
10.3	Database installation	13
11	Configuration	14
11.1	Database configuration	14
11.2	Server configuration	15
11.3	Mail notification configuration	21
11.4	Logging configuration	25
11.5	Defining attribute types	26

12 Server operation	26
13 Upgrading UVOS server	27
13.1 Version specific notes	27
13.2 Generic upgrade instructions	28
14 APPENDIX - permission requirements	29

UNICORE VO Service (UVOS) is a client-server system, developed to be used as an additional tool for large distributed systems, providing a solution for grid users management. Grid systems, especially UNICORE grid middleware, are the mainspring of the UVOS system. UVOS can be used with different systems, however is designed primarily to support UNICORE grid middleware.

For more information about UVOS visit <http://uvos.chemomentum.org>.

1 Introduction

UNICORE VO Service (UVOS) is a client-server system, developed to be used as an additional tool for large distributed systems. Grid systems, especially UNICORE grid middleware, are the mainspring of the UVOS system. Although UVOS can be used with different systems, for the purpose of this document we will use the term *grid system* to refer to supported systems.

The fundamental UVOS features are:

- storing identities of grid users and other identifiable components (for example servers),
- organising identities in hierarchical groups,
- assigning arbitrary attributes to users in various ways,
- registrations requests (or *VO applications*) support. UVOS exposes those features as remotely accessible operations through the web service mechanism. This provides internal system access authorisation and authentication.

Typical usage patterns of the UVOS system include:

- grid node access authorization support, which enables granting access to members of a particular group or owners of selected attributes
- mapping grid user identity onto another one (usually in different format),
- storing dynamic and static information about grid entities. For more detailed information about possible ways UVOS can be deployed see [usage scenarios description](#) Section 9.

The UVOS system is build upon well established standards. For instance, all query operations used by clients are available through the SAML 2 protocol. Moreover, the following optional SAML profiles are implemented to ensure interoperability:

- SAML Attribute Query Deployment Profile for X.509 Subjects,
- SAML Attribute Self-Query Deployment Profile for X.509 Subjects,
- OGSA Attribute Exchange Profile Version 1.2,
- XACML Attribute Profile.

2 UVOS Overview

UVOS is a component that acts as an information point and organises entities within a hierarchical group structure. Top level groups of this structure are called virtual organisations. Each *entity* is assigned a list of group membership and a set of attributes. An attribute is composed of a name and a set of values, which can be empty. In addition, a single entity can possess multiple representations, for example in two different formats. These equivalent incarnations of the same entity are called *identities*, and are usually invisible for an outside user.

2.1 Example

What follows is a comprehensive example of the UVOS database. It serves as an illustration of various concepts which are presented in more detail later on. Please note that you can find a script in the UVOS server distribution which creates the following example, so you can easily experiment with it.

Groups hierarchy:

```
[UVOS root]
|-Math-VO
|   |-Staff
|   |   |-Admins
|   |   |   |-u:UNICORE example user (DN)
|   |   |   |-u:Eve (email)
|   |   |   |-u:Amy (email)
|   |   |   |-u:Ben (email)
|   |   |-Scientists
|   |   |   |-u:Ben (email)
|   |   |   |-u:Andrew (DN)
|   |   |   |-u:Chris (email)
|   |-UADB
|   |   |-SiteA
|   |   |-SiteB
|-QSAR-VO
|   |-u:Ben2 (DN)
|   |-u:Tom (DN)
|   |-u:UNICORE example user (X509Cert)
```

Two top level groups, called VOs, are defined in this example. The first one (`Math-VO`) has a complicated structure with subgroups, while the second (`QSAR-VO`) is very simple, with no subgroups. The users are presented with `u:` prefix, along with their identity type in brackets. In this example there are two equivalent identities (in other words: entity with two representations) of "UNICORE example user": of DN type and of X509Certificate type. "UNICORE example user" is the identity taken from the demo certificate which is distributed with UNICORE 6 quickstart package.

3 Entities

Distinct members of the UVOS system are called entities. Every entity has a unique label and usually one token that defines it. The token must be in one of the supported formats, which are:

- X.509 certificate,
- DN - distinguished name,
- Email address.

A token along with its type is called an identity. As explained an entity typically possesses one identity, but it can also have more, even if they are of the same type. For instance, an entity with the label "Jimmy Page" can have three identities: X.509 certificate issued by VeriSign, another X.509 certificate issued by ICM Warsaw University and an Email address jpage@example.com.

It is worth pointing out that all of the identities that compose an entity share the same characteristics (attributes, group membership, permissions, etc.). The UVOS works using entities, so that any of its identities can be given as a representation.

3.1 Which type of identities shall be used?

There are several things that influence the answer to the above question. We will give some simple rules which apply in typical UNICORE (or in general grid) situation.

First of all we can observe that DN and X.509 certificate are quite similar, namely certificate contains (in a certificate subject field) a user's DN. Also the fact is that grid sites ask for DN type identity when authenticating or authorizing users.

So if it is enough administrator can use DN-type identities and forget about certificates. However it is often more comfortable for VO administrators to have a full entity certificate as it carries more data about the user. So in UVOS prior to 1.3 version, administrator usually created both identities - DN and X.509 - under a single entity. From the version 1.3 on it is not needed, as server was extended to use certificate also when asked for DN type identity (and obviously when there is no such a DN-type identity already defined). So when adding X.509 certificate identity is usually sufficient and it is not required to create DN-type identity.

The usage of email type identity is completely another story. It is used for two purposes:

- as a simple way to authenticate UVOS administrators
- to authenticate grid users to the web portals with a password.

So mail identity is required for regular grid users only as an additional identity (to the "base" DN or certificate type identity) when this user will access the grid through the WWW.

4 Attributes

Attributes are composed of a name and a list of values. A name is a URN, and values are arbitrary strings. The value list can be empty.

The administrator can assign attributes to entities. There are three methods of doing this: :

- using **global attributes**: an entity can have an attribute assigned globally. Such an attribute is valid always and in every context,
- using **group-assigned attributes**: an attribute can be assigned to a group, in which case all members of this group automatically hold this attribute (no matter if they were added later or prior to the creation of the group-assigned attribute). It is worth pointing out that this attribute is valid only in the scope of this group,
- using **group-scoped entity attributes**: those attributes are assigned to the entity, just like global attributes, but have an additional group restriction and are valid only in the scope of the group.

The last two methods introduce a "group-scoped validity" of attributes, which requires a further explanation. From the technical point of view the requester can ask for the entity's attribute in a specified group. Such a query will return all entity's global attributes and all group-scoped attributes valid within this group. Considering the example situation shown above, the user Eve can have the "administrator" attribute in the scope of Math-VO (remember that a VO is just a normal group), but does not have it in VO QSAR-VO, where she is a regular user.

There is also another distinction between attributes, which is important only for query purposes:

- **effective attributes** are those that VO service consumer (e.g. Policy Decision Point) is interested in. SAML queries always return effective attributes. When querying without defining a group scope, all global attributes of the entity will be considered effective attributes. On the other hand, when querying an entity in the scope of a particular group returned attributes contain both global and group-scoped attributes. Note that attributes can be inherited – all attributes valid in the scope of the subgroup are also valid in the scope of the parent group. **IMPORTANT NOTE:** From the version 1.2 of UVOS server, group-scoped entity attributes override group attributes. Consider user U, who is a member of group G and holds the attribute A with value VAL1 in the G group scope, but at the same time group G has an attribute A defined with value VAL2. The previous server versions returned both values (i.e. A with values VAL1, VAL2), while the current server release will return A with value VAL1 only.
- **exact attributes** have the the same functionality as effective attributes when considering global and group-assigned attributes. The difference lies in group-scoped entity attributes. In such a case exact attributes assigned to ID1 in group G are simply those directly assigned to ID1 in the scope of group G. They, for example, do not include attributes which are global or assigned to any of G subgroups (which are only considered effective attributes). Exact attributes are used in VO managing (administrator assigns exact attributes by definition), and **SHOULD NOT** be used for authorization purposes.

Note: it is possible to assign group-scoped entity attribute even when the identity isn't a member of the group. Consequently, this attribute will be visible as an exact attribute only, and not as an effective attribute.

5 Authentication

Every request coming to the server is a subject of an authentication process. The authentication result (whatever it is) is mapped to one of the identities available in service's database - in other words there is no extra database with users of the VO service.

There are several issues here: to what type of identity requester should be mapped, and what authentication mechanism should be used?

First of all you can enable different authentication data sources:

- **TLS** authenticated TLS session peer is mapped to an identity of X509 certificate type or DN type. The additional `uvos.server.authn.mapTLSCertToDNFirst` property (boolean) controls which of those types is tried in the first place.
- **HTTP** an email type identity is created as obtained from HTTP BASIC authentication header and verified using a password, which is also set in the header.

Those options are tried in order, determined by configuration file parameter with name `uvos.server.authn.order`. Administrator need not to enable both of them. Authentication options which are used must be separated with a space character. The first identity in order that is successfully verified (and present in database) is used. If there is identity found which is invalid or not present in database, the authentication process can either continue checking the next possibility or fail. This is controlled by configuration option `uvos.server.authn.failOnError`. Note that it effectively makes sense only when you have both options enabled.

Note that this form of authentication was introduced in the version 1.3.2, earlier versions used a more complicated one.

6 UVOS access authorization

UVOS access is restricted by it's own authorization stack. No external components/services are used to perform authorization. The first part of this section describes in detail the whole authorization process. The system is flexible however quite complicated too. Therefore 2nd part shows simple set of rules (also employed in default configuration) that allow for easy configuration of secure UVOS access. It should be enough for the most of applications. Readers are encouraged to at least briefly scan the initial paragraphs of the next section before proceeding to the second one.

6.1 Authorization overview

While accessing an operation requires the accessing entity to possess zero or more Permissions, in most cases at least one is needed. The following permissions are defined:

- **read (r)** - this permission is needed to perform various operations that read current VO contents.
- **fullRead (f)** - this permission is needed to read special VO contents like historical data or hidden attributes.
- **identityCtl (i)** - this permission is needed by operations that are used to manage identities (add/remove).
- **write (w)** - this permission allows for changing VO contents and UVOS authorization configuration.

[Appendix Section 14](#) defines precisely what permissions are required by available operations. Permissions may be granted as global or as group-scoped, i.e. valid only in the scope of a specified group.

Permissions are assigned to an entity on the basis of three conditions (i.e. if an entity meets required conditions it receives permissions). The specification of those conditions is called a policy. Every group can have its own policy and there is also one global policy. A policy is a set of pairs (condition, permissions). Possible conditions can be defined as:

- **an attribute** - the most common way of defining a condition is to use an attribute. In this case the permission is granted to individuals who possess the specified attribute. **IMPORTANT (1):** if the attribute contains values, then the permission is granted to everybody who possesses this attribute with at least one of the possible values. If the attribute has no value, the permission is granted to everybody who has an attribute with the same name (with or without values - it is not relevant in this case). **IMPORTANT (2):** for global policies (see below) only global attributes are used to evaluate the condition.
- **a member** - the member condition grants the permission to every member of the group that the policy is assigned to. This condition will never be met when the accessed operation requires a global permission.
- **an owner** - the owner condition is the trickiest one. The permission is granted when an entity that tries to perform an operation is also the subject of this operation. This is only possible in a limited number of operations, e.g. the caller of a `isMember(subject, group)` method can meet this condition if he/she is also the "subject" in parameters list.

A group's policy is established in the following way:

- if the group has a policy set, then it is used.
- if not then the parent group's policy is used (of course this is a recursive behaviour).

- if the group is a top-level group (i.e. no parent) then the global policy is used as the group's policy.

In short, group policies are inherited from parent groups and are not merged in any way (the first found is used). When a UVOS operation is invoked it can be authorized globally or, when the operation affects only a particular group, in the scope of the given group. In the first case only the global policy is used. In the second case the both a global policy is used together with the group's policy. There are no conflicts between global policy and the group's policy as the resulting permissions are always either the same or better as those coming from the individual ones.

Note

even with group-scoped access, global attributes are still needed to get permissions that are granted by a global policy.

Example 6.1 Example:

Let's assume the following policies are defined:

- **Global Policy:** (member \rightarrow r), (owner \rightarrow rf), (Attribute *superuser* \rightarrow rfiw)
- **Group /Math-VO Policy:** (member \rightarrow rf), (Attribute *mathmanager* \rightarrow rfiw)
- **Group /Math-VO/Staff/Admins Policy:** (member \rightarrow rfiw)

With the above assumptions effective policies for the groups are as follows:

- **Group /Math-VO:** (member \rightarrow rf), (Attribute *mathmanager* \rightarrow rfiw)
- **Group /Math-VO/Staff:** as above
- **Group /Math-VO/Staff/Admins:** (member \rightarrow rfiw)
- **Group /QSAR-VO:** (member \rightarrow r), (owner \rightarrow rf), (Attribute *superuser* \rightarrow rfiw)

and those groups' policies are evaluated always together with global policy to establish what are caller permissions.

Now we can present the detailed algorithm for making an authorization decision:

1. Let *PI* be an empty policy and *EP* be an empty permission set.
2. If an accessed operation is in the scope of a group then set the policy *PI* to the policy for this group. If needed inherit the policy from parent groups, remembering that the first parent's policy should be used. If no parent policy is set then use *PI* = *Global Policy*.
3. If the accessed operation is in a group scope check if caller is a member of this group. If so add all member permissions from *PI* and global policy to *EP*.

4. If the accessed operation can be self (or owner) accessible, check if caller is accessing herself. If this is true add all owner permissions from *PI* and global policy to *EP*.
5. Retrieve all global attributes of the caller and add them into a *GlobA* set. If the accessed operation is in a group scope then retrieve all group scoped effective attributes of the caller and add it to the *GroupA* set.
6. If the accessed operation lies in the scope of a group then find all permissions, which are either present in *PI* or result from *GroupA* attributes and add them to *EP*.
7. Find all permissions resulting from *GlobA* attributes and present in *Global Policy* and add them to *EP*.
8. Check if all permissions, required for the invocation are present in *EP*. If yes then grant access, deny otherwise.

6.2 The simple (default) authorization scheme

The above description shows that the authorization scheme in UVOS is powerful, but also complicated. This section presents a simple usage pattern, that should be sufficient for most situations. UVOS administrator can, of course, deploy a modified version of the pattern or even mix various authorization schemes on a per VO basis.

The fundamental idea is to use a separate special attribute that grants UVOS access permissions (and is not used for external purposes) with a fixed authorization policy for all groups.

This special attribute's name is `urn:authz:intervo:vo` and following values are meaningful for it:

- **read**: grants **r** permission,
- **fullRead**: grants **rf** permissions,
- **identityCtl**: grants **rfi** permissions,
- **write**: grants **rfiw** permissions (i. e. all permissions).

Only one (fixed) global authorization policy is used. It assigns the permissions defined above to the holder of corresponding attribute's value(s). Note that this policy becomes the policy for every group (by inheritance).

With the above rules authorization is managed in a simple way. It is controlled by assigning the `urn:authz:intervo:vo` attribute with proper values to the users. If there is a need to give permission to all members of a group then the authorization attribute should be set as the group's attribute.

Additionally, the default global authorization policy grants **r** permission for any **group members** in that group scope and **rfi** permissions for **owner access**. This allows members of a group to access it and always allows to access self data.

The special care should be taken when assigning `urn:authz:intervo:vo` attribute as a global attribute - it will result in granting of corresponding permissions in all contexts (i.e. for all groups and as global permissions).

7 VO registrations (applications)

The UVOS system contains an interface which allows for storing and processing VO applications. The system is organised as follows:

- **VO application form** is used to specify overall rules that its applications must obey. It also contains additional information about data presented to the applying user. Examples of included information are: a description, the group to which the application is connected etc.
- **VO application** is issued by the user who has already filled the form.
- **VO form administrator** (every application form can have its own administrator) processes and accepts or rejects the application.

The UVOS interface provides a possibility to store and modify both forms and applications. In addition, application processing is possible. It must be noted, however, that form rendering and user's input processing is not available as a server's functionality. When needed, it can be achieved by using an additional component. Currently only one, **uvos-webapp** is available. It provides a web interface which displays application forms.

8 Email notification

The 1.0 and 1.1 releases of the UVOS were capable only of sending simple email notifications when VO application was submitted (to the application form administrator) and when VO application was processed (to the requester).

Since the release of version 1.2 this functionality was greatly enhanced:

- It is possible to subscribe for notifications dynamically at runtime.
- Notifications can be sent as an effect of almost all management operations, which include adding a new group member, deleting an identity or even changing an attribute.
- Notifications can be group scoped, i.e. sent only if the event occurred in a scope of a particular group. E.g. VO administrator can register herself to get notifications when new identities are added to a specified group/VO.
- There are no limits on the number and configuration options and notifications (notifications with many recipients are also supported).

The following operations issue notifications:

1. addGroup
2. removeGroup

3. copyGroup
4. addIdentity
5. addEquivalentIdentity
6. removeIdentity
7. setAttribute
8. removeAttribute
9. addToGroup
10. removeFromGroup
11. setIdentityStatus
12. purgeHistoricalData

9 UVOS usage scenarios

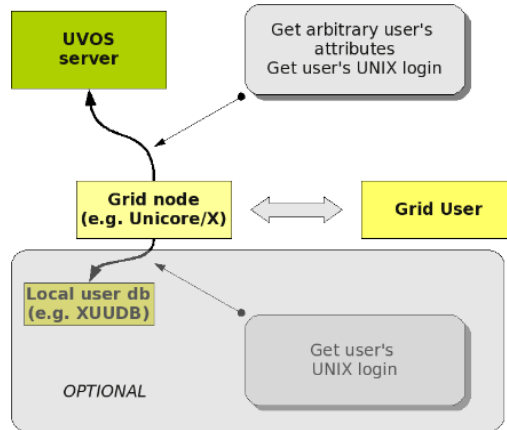
There are several typical deployments in which UVOS can be used. We present them below.

9.1 PULL authorization

In the so called "pull mode" service (e.g. grid execution server, Unicore/X in case of UNICORE middleware) contacts UVOS server to obtain the attributes of a user which tries to use one of its services.

The attributes received from UVOS server can be used for authorization (e.g. server's policy may permit only those users which are in a certain UVOS group or possess some attributes). Also service may use received attributes for other purposes; for instance UNICORE can be configured to use a predefined (scoped) UVOS attribute as an information about local UNIX account of the requester. Attribute scope is used to distinguish mappings for multiple servers.

The PULL mode is depicted on the picture below:

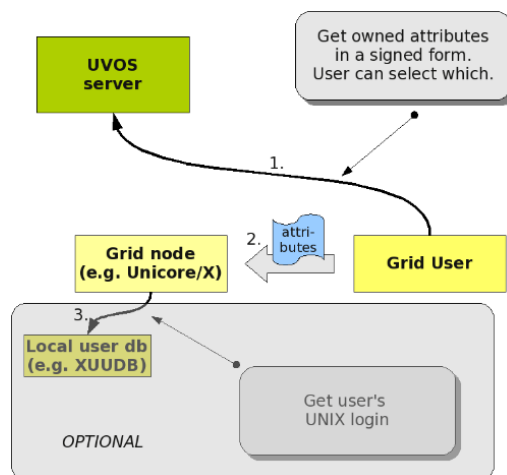


Pull mode is transparent for grid users. However is more difficult for grid administrators to set up: every grid site must be correctly configured to use UVOS.

9.2 PUSH authorization

In so called "push mode" user first contacts UVOS server on her/his own and receives the list of possessed attributes in a signed assertion. Later this assertion can be attached to the requests send to grid services. If the service trusts assertion issuer (i.e. UVOS server which issued it) then it can use the attributes for authorization.

Note that user can ask UVOS server only for subset of owned attributes. In such a case user can hide part of her/his identity or alter the execution (e.g. choosing her/his the role to be used). The PUSH mode is presented on the picture below:



Pull mode is more scalable in terms of server administration and easier to set up. However it requires more user interaction and thus is more suitable for advanced grid users.

9.3 Web portal authentication

UVOS can be used to authenticate web browser users. SAML 2.0 standard is used to achieve this functionality. To enable it you will need additional web application which provides a WWW login page - it is called **uvos-webauthn** and is available in UVOS distribution.

Details of this deployment can be reviewed in many places. E.g. see Wikipedia article http://en.wikipedia.org/wiki/SAML_2.0, section on Web Browser SSO Profile. UVOS uses POST binding. For more detailed, technical description see SAML 2.0 core specification, SAML 2.0 profiles and SAML 2.0 bindings documents. References can be found on the aforementioned Wikipedia page. Also it is the same style as Shibboleth 2.0 works (it was not tested but in principle it should be possible to use Shibboleth SP with UVOS).

10 Installation

Only UNIX systems are supported for installation of the UVOS server and UVOS client tools. Manual installation on Windows is possible assuming that start scripts are converted to Windows BAT files.

UVOS is distributed in the following formats:

1. As a platform independent installation archive.
2. As an RPM available for Scientific Linux 5 platform. This RPM is tested on Scientific Linux, but should work without any problems with recent version of Centos (e.g. Centos 5.5), Fedora (e.g. Fedora 13, 14) and other Red Hat derivatives.

IMPORTANT NOTE ON PATHS

UVOS is distributed either as an platform independent and portable archive or as an installable, platform dependent package such as RPM. After installation paths to files are different depending on installation source used. If installing using distribution-specific package the following paths are used:

```
CONF=/etc/unicore/uvos-server
BIN=/usr/sbin
LOG=/var/log/unicore/uvos-server
```

If installing using portable archive all UVOS files are installed under a single directory. Path prefixes used then are as follows, where INST is a directory where UVOS was installed:

```
CONF=INST/conf
BIN=INST/bin
LOG=INST/log
```

The above variables (CONF, BIN and LOG) are used throughout the rest of this manual.

10.1 Installation from the archive

Download the UVOS server archive from the UNICORE project website. It is enough to unpack the contents of the archive into a final folder. No further actions are required.

10.2 Installation from the RPM package (RedHat distributions)

Download RPM package from UNICORE download site and install it using the rpm command as root user:

```
$ rpm -i unicore-uvos-server-1.4.1.noarch.rpm
```

You can also (what is suggested) use yum to install and subsequently update the UVOS server. The yum installation may be performed as follows (note that first command is needed only if you have not yet installed the EMI yum repository):

```
$ wget --no-check-certificate \
https://twiki.cern.ch/twiki/pub/EMI/EMI-1/rc1.repo \
-O /etc/yum.repos.d/emi-1.repo
$ yum install unicore-uvos-server
```

10.3 Database installation

If you wish to do a quick setup (for small or medium installation) you can use an embedded database, which is supplied within distribution. In this case you can skip to the [Configuration Section 11](#) section as the embedded DB it is installed by default and you need only to invoke initialization script. Otherwise you should follow this section to install and configure a standalone DBMS.

10.3.1 PostgreSQL

Ensure that PostgreSQL database is installed on your system. Login as postgresql user, add a password protected user for the UVOS server and then create a database:

```
$ createuser <USERNAME> -P
$ createdb uvosdb -O <USERNAME>
```

Finally, verify if everything is correct, by manually logging to the newly created database:

```
$ psql -h localhost -U <USERNAME> uvosdb
```

If there is connection problem verify the PostgreSQL configuration in the pg_hba.conf file. Check, if there are correct settings allowing for local connections, e.g. like those:

```
# IPv4 local connections:
host    uvosdb          all          127.0.0.1/32          md5
# IPv6 local connections:
host    uvosdb          all          ::1/128               md5
```

Order of lines in this file is important so in general the above lines should be at the beginning.

Note

From version 1.2 the UVOS distribution contains a PostgreSQL driver so there is no need to set it up manually.

10.3.2 Other DBMSes

Although currently no other DBMSes are supported, it is fairly easy to set it up. It requires porting the SQL database structure creation script to the chosen DBMS SQL flavour. If you want to do this feel free to contact us.

11 Configuration

11.1 Database configuration

The database configuration is done in properties file `CONF/datamap.properties`. You should uncomment and then edit the section which matches your DBMS. For a correct setup, you need to specify a username, password and a connection URL. The last change is required only when your database name is different than "uvosdb".

Note

When using an embedded DB the default `datamap.properties` file need not to be modified.

Final step (required also for the embedded DB) is to initialize DB contents. This is done by invoking a script:

```
$ BIN/initdb.sh
```

or in case of distribution specific package:

```
$ BIN/unicore-uvos-server-initdb
```

11.2 Server configuration

Basic server configuration (security, network addresses, authentication options, etc.) is done in the `CONF/uvosServer.conf` file. The file is well commented, with an explanation for all options and has to be reviewed by every administrator.

After installation vast majority of options have reasonable values. However the following options require a review and often an update:

- Security settings, i.e. all settings with keystore or truststore in name. By default after installing with distribution package there are no certificates provided. The portable archive bundle provides example, insecure certificates and keys (don't use them in production, never!),
- network address and ports to use.

All the existing options are:

Property name	Type	Default value	Description
<i>Server general settings</i>			
<code>uvos.server.mailConf</code>	file path	<code>CONF/mail.properties</code>	A configuration file for the mail notification subsystem.
<code>uvos.server.mailTemp</code>	file path	<code>CONF/mailTemplates</code>	A file containing templates of email notification messages sent when various UVOS events occur.
<code>uvos.server.attributeTypes</code>	file paths separated by ','	<code>CONF/attributeTypes</code> <code>CONF/attributeTypes</code>	A specification of files with definitions of attribute types that the server should load on startup. Comment it out if you don't want this behaviour. Note that, you can specify multiple files separated by ',' (comma and space).
<code>uvos.server.attributeTypes.update</code>	true or false	false	If set to false then only new attribute types present in file will be added to DB. Otherwise (update=true) the types that exist in DB will have their descriptions updated.

Property name	Type	Default value	Description
uvos.server.useExternalRegistry	boolean	false	If set to true server will register its SAMLAttributeQuery service in one or more UNICORE registries (defined by properties uvos.server.externalRegistryUrl
uvos.server.externalRegistryUrl.NUM	URL	empty	Registry address. You can define more than one. All those properties must end in consecutive registry numbers, starting from 1.
<i>Server identity</i>			
uvos.server.keystore	file path	CONF/certs/dummyServer	A keystore file containing server's private key and its certificate.
uvos.server.keystorePassword	string	the!server	The keystore password.
uvos.server.keystoreType	JKS or PKCS12	JKS	The keystore type.
uvos.server.keyStorePassword	string	as uvos.server.keystore	The private key keystore password.
uvos.server.keystoreAlias	string	mykey	The Alias of the keystore entry containing server's private key.
<i>HTTPS configuration</i>			
uvos.server.https.enabled	boolean	true	Enables or disables using the HTTPS port. Note that HTTPS is the recommended transport mechanism.
uvos.server.https.port	int	2443	The HTTPS port to be used.
uvos.server.https.hostname	hostname or IP addr.	localhost	The hostname or IP address for HTTPS connections.
uvos.server.https.allowAnonymous	boolean	true	If set to true then unauthenticated client connection via HTTPS will be accepted (and, if possible, authenticated using other means like HTTP BASIC auth).

Property name	Type	Default value	Description
uvos.server.https.truststore	file path	CONF/certs/dummyServer	The location of the jks truststore with trusted CAs certificates. If allowAnonymous is turned off then only clients that possess certificates issued by the truststore CAs will be accepted. In any case, only trusted certificates can be used to authenticate the client.
uvos.server.https.truststorePasswd	string	the!server	The truststore password.
uvos.server.https.truststoreType	JKS or PKCS12	JKS	The truststore type.
<i>HTTP configuration</i>			
uvos.server.http.enabled	true or false	true	Enables or disables the HTTPS port. It is NOT recommended to use this transport mechanism as it doesn't provide encryption.
uvos.server.http.port	int	65535	The HTTP port to be used.
uvos.server.http.hostname	hostname or IP addr.	localhost	The hostname or IP address for HTTP connections.
<i>Authentication</i>			
uvos.server.authn.order	List of the strings: <i>TLS</i> <i>HTTP</i>	TLS HTTP	Defines the order in which authentication sources should be used. See [3.2.1] for the further explanation.
uvos.server.authn.failOnFailure	true or false	true	If set to true then authentication will fail if validation of one of the PRESENT authentication data fails. Otherwise the authN process will continue, checking the next possible authentication data source.

Property name	Type	Default value	Description
uvos.server.authn.mapToDNFirst	boolean	false	If set to true and validation of some of PRESENT authentication data fails then authentication will fail. Otherwise authN process will be continued, checking the next possible authentication data source
uvos.server.authn.enableExplicitTrustDelegation	boolean	true	Whether to enable Explicit Trust Delegation
<i>SAML service settings</i>			
uvos.server.saml.allowCertificateAsDN	boolean	true	Enables or disables mapping of X509-type identities to DN-type identities. See [3.2.2] for further explanation.
uvos.server.saml.issueURI	URI or empty	empty	This property controls the server's URI which is inserted into SAML responses (the Issuer field). It should be a unique URI which identifies the server. The best approach is to use the server's URL . If absent the server will try to autogenerate one.
uvos.server.saml.signResponses	always, asRequest or never	asRequest	Defines when SAML responses should be signed. Note that it is not related to signing SAML assertions which are included in response. <i>asRequest</i> setting will result in signing only those responses for which the corresponding request was signed.

Property name	Type	Default value	Description
<code>uvos.server.saml.signAlways</code>	<code>boolean</code> or <code>Request</code> or <code>ifResponseUnsigned</code> or <code>never</code>	always	Defines when SAML assertions, that are put in responses, should be signed. The <code>ifResponseUnsigned</code> will result in signing only those assertions which are sent in an unsigned response. Note that several SAML profiles mandates signing assertions so it is best to set it to <code>always</code> .
<code>uvos.server.saml.validityPeriod</code>	positive integer number	14400	Controls the maximum validity period of an attribute assertion returned to client (in seconds). It is inserted whenever query is compliant with "SAML V2.0 Deployment Profiles for X.509 Subjects", what is usually the case
<code>uvos.server.saml.requestValidityPeriod</code>	positive integer number	120	Defines maximum validity period (in seconds) of a SAML request. Requests older than this value are denied. It also controls the validity of an authentication assertion.

11.2.1 Authentication

The UVOS server provides several ways of authenticating incoming requests. All authentication options are in fact a result of an authentication identity format and authentication source combination, i.e. security material used to establish the identity. The authenticated requester must be known to the VO service (but does not need to be a member of any VO/group). The possible options are listed below.

- **TLS** - it is possible when the request uses a HTTPS connection AND (!) the client was successfully authenticated. Note that when `allowAnonymous` is true then the request may arrive through a HTTPS channel, but still be unauthenticated. In such a case this authentication method will not succeed. The resulting identity format is either a X.509 certificate or DN of the HTTPS client. The order in which formats are tried (DN or certificate) can be configured.

- **HTTP** - possible when the requester used HTTP simple authentication (i.e. login and password encoded in HTTP header). The resulting identity format is an email address. The user is authenticated when the UVOS password check is successful.

The order in which the above options are tried can be configured by the server administrator. It is possible to set which authentication data source is tried first (TLS or HTTP). Whenever there is no authentication material the server jumps to the next available option. It might happen that authentication material, for a specified option, is present but is invalid. In such a case the server can either fail the authentication immediately or skip to the next option - this behaviour is also configurable.

Example 11.1 Example:

Let's assume that HTTPS is the only transport enabled with `uvos.server.https.allowAnonymous` parameter set to true, the authentication order is as follows: "HTTP TLS". Moreover, let's assume that the incoming request is arriving through a client-authenticated TLS session, but has incorrect HTTP auth data in the header (user is not registered in VO DB).

- *Case 1: the server doesn't trust the client's certificate and `uvos.server.authn.failOnError` is true.* The `uvos.server.https.allowAnonymous` setting will cause the request not to be denied at transport level. What follows is the authentication step. TLS will be skipped (no input data). As a result, the request will be processed by HTTP module which will fail and cause authentication process to stop.
 - *Case 2: the server doesn't trust the client's certificate and `uvos.server.authn.failOnError` is false.* The behaviour will be the same as above.
 - *Case 3: the server trusts the client's certificate and `uvos.server.authn.failOnError` is true.* The first TLS will succeed if the DN from certificate is registered in the VO database. Otherwise authN will fail immediately.
 - *Case 4: the server trusts the client's certificate and `uvos.server.authn.failOnError` is false.* The TLS will succeed if the DN from certificate is registered in the VO database. Otherwise authN will be continued. HTTP will fail (incorrect data).
-

11.2.2 Using X509 certificates as DNs

Starting from the release 1.3 server allows to use identities of X.509 certificate type as DN type identities. This feature is turned on by default. It works only for SAML attribute queries (so normal queries made for instance by Unicore server or clients) and in SAML authentication protocol (but this is rarely used if any so you can ignore this fact). In case of attribute query this functionality is activated when:

1. the query subject is of DN type identity (note that it is a standard case),
2. there is no identity in the database which is equal to the query subject and which has a DN type,
3. there is an identity A in DB which is of X.509 certificate type and the subject of this certificate is the same as the query subject.

When all above conditions are met then server will return attributes of A. If the certificate mapping feature is turned off then in such a case server will respond with error saying that the query subject is unknown.

11.3 Mail notification configuration

One of the features of the UVOS server is to collect VO applications (or registration requests). It is possible to use an email notification mechanism along with the application process. The notification can be generated in two specific cases:

- whenever a new application arrives (notification is sent to the VO administrator).
- whenever a application is processed (notification is sent to its owner).

Moreover, from version 1.2 upwards, it is possible to configure UVOS to send notifications as a result of nearly every management operation. There are two configuration files that control email notifications: one containing general configuration and the another containing templates of messages to be sent.

The location of the basic mail configuration file is specified in the main configuration file. The default location is `CONF/mail.properties`. The table below shows the configuration options along with the type, default value and the description.

Property name	Type	Default value	Description
<code>mailx.enable</code>	<i>true or false</i>	false	Set it to true to enable email notification sending. The rest of the mail configuration is ignored if the value is set to false.
<code>mailx.sendTestMessageTo</code>	email address	unset	Use this property only if you want to debug the email configuration. When set, the server will send a test message to the specified address upon every startup.

Property name	Type	Default value	Description
mail.from	email address	root@localhost	User name which will be used for the From: field of the email. It is also used as a SMTP envelope return address if it is not overridden below.
mail.smtp.host	host address	localhost	The SMTP server to connect to.
mail.smtp.starttls.enable	boolean	false	If true it enables the use of the STARTTLS command (if supported by the server) to switch the connection to a TLS-protected connection before issuing any login commands. IMPORTANT! SMTP server's certificate must be trusted to establish the connection. The software will use the same truststore that is defined in main configuration of the server. It is therefore important to add the SMTP server's CA certificate to the main truststore
mail.smtp.from	email address	as in mail.from	Email address to use for the SMTP MAIL command. Is also sets the envelope return address.
mail.smtp.auth	boolean	false	If true, attempt to authenticate the user using the AUTH command.
mailx.smtp.auth.username	string	unset	The username used when authentication is enabled by mail.smtp.auth.
mailx.smtp.auth.password	string	unset	The password used when authentication is enabled by mail.smtp.auth.
mail.smtp.timeoutSocket	integer number	infinite	I/O timeout value in milliseconds.
mail.smtp.connectionTimeout	integer number	infinite	Socket connection timeout value in milliseconds.

Property name	Type	Default value	Description
mail.smtp.port	1-65535	25	The SMTP server port to connect to.
mail.debug	<i>true or false</i>	false	Set this property to true if you want to see debug messages (are printed to the standard error, not logged!).
OTHER OPTIONS	-	-	For other options see the SUN Java Mail documentation. http://java.sun.com/products/javamail/javadocs/com/sun/mail/smtp/package-summary.html

The location of the template configuration file is defined in the main UVOS configuration file and is set to `CONF/mailTemplates.properties` by default. In this file you can specify the subject and the body of all notifications. There are also special entries for VO applications. If a customised notification template is not specified then the default template will be used. It is possible to create dynamic emails by using variables denoted with `${...}`. If such variables are used the server will replace the variable with it's actual value e.g. for

The table below shows the properties that can be defined in the template configuration file along with their description.

Property name	Type	Default value	Description
mailtemplate.newApplication.subject	string	give a try to see	Defines the subject of a message sent to the VO admin (who is defined in an application form) when a new application is submitted. The runtime variables, available for this message are: <code>\${FORM_NAME}</code> - application's form name, <code>\${FORM_ID}</code> - application's form id, <code>\${FORM_GROUP}</code> - application's base group

Property name	Type	Default value	Description
mailtemplate.newApplication.body	string	give a try to see	Defines the body of the message sent to the VO admin (who is defined in applications form) when a new application is submitted. Available runtime variables as the same as above.
mailtemplate.applicationProcessed.subject	string	give a try to see	Defines the subject of the message sent after an application is processed (but not REMOVED). The runtime variables, available for this message are: <code>\${APP_ID}</code> - ID of application, <code>\${APP_ADMIN_NOTES}</code> - application's notes as appended by admin, <code>\${APP_STATUS}</code> - new application's status (e.g. REJECTED or ACCEPTED), <code>\${FORM_NAME}</code> - application's form name, <code>\${FORM_GROUP}</code> - application's base group
mailtemplate.applicationProcessed.body	string	give a try to see	Defines the body of the message sent after an application is processed (but not REMOVED). Available runtime variables as the same as above.
mailtemplate.subject	string	give a try to see	Defines the default subject for messages sent as a result of management events.
mailtemplate.body	string	give a try to see	Defines the default body for messages sent as a result of management events.

Property name	Type	Default value	Description
mailtemplate.<ACTION>	String	unset	Redefines the subject of messages sent as a result of an event with the name specified by the <ACTION> param.
mailtemplate.<ACTION>	String	unset	Redefines the body of messages sent as a result of an event with the name specified by the <ACTION> param.

Although the available runtime variables will usually depend on the action, there are two common variables:

- `${ACTION}` - the name of the action, e.g. `addGroup`
- `${CALLER}` - the person that performed the action.

The following variables are action-dependant:

- `${SUBJECT}` - the value of the identity (e.g. `which was added/removed...`) or a group scoped identity (e.g. `whose group scoped attribute was added`)
- `${GROUP}` - the name of the affected group (e.g. `created/removed`)
- `${TARGET}` - (for `copyGroup`) the new name of a copied group
- `${MOVED}` - (for `copyGroup`) simple *moved* or *copied* string
- `${EQUIVALENT}` - (for `addEquivalentIdentity`) the name of an already existing, equivalent identity
- `${ATTRIBUTE}` - the attribute, which was either set or removed
- `${STATUS}` - (for `setIdentityStatus`) *disabled* or *enabled*
- `${DATE}` - (for `purgeHistoricalData`) the date up to this date historical data was cleared.

11.4 Logging configuration

Logging is done through the LOG4J logging facility, which is configured in the `CONF/log4j.properties` file. For any further information, please refer to the LOG4J documentation. By default, the log output is put into the `LOG/uvos.log`, file which is rotated every day. The server's standard output and error is logged to the `LOG/startup.log` file. It is worth noting that the number of information that can be found there is limited.

11.5 Defining attribute types

Although you can add, delete or modify attribute types using client tools when server is running, it is convenient to load a predefined set of attribute types with descriptions. It is a possible to define attribute types in a files, which are read on server startup. The server can either add new or even update existing attribute types with the data from these files. See `uvos.server.attributeType...` configuration options for further details. The standard location of the files containing attribute types is under the `CONF/attributeTypes/` directory.

The format of the attribute definition files of the UVOS system is extremely simple. It is line based and obeys the following rules:

- Every line that begins with a `#` is ignored.
- Every AT is defined by a block of consecutive lines.
- Every block is separated from other blocks by at least one empty line.

A block contains 4 lines:

1. `AT` (literal),
2. attribute name (key),
3. attribute short description or full name,
4. attribute full description.

Moreover, the UVOS distribution contains a tool to translate LDAP schema files to the above format. The `BIN/convertLDAPSchema.sh` (or `BIN/unicore-uvos-server-convertLDAPSchema`) invokes a converter that translates its standard input in LDAP schema format to the UVOS format. E.g.:

```
# BIN/convertLDAPSchema.sh <someLDAP.schema >additionalUVOSats.at
```

The default UVOS distribution contains two examples of files with attribute type definitions. One contains core UNICORE authorization attributes (e.g. `xlogin` attribute) while the other holds a set of common LDAP attributes.

12 Server operation

The server management scripts can be found in the `BIN` directory. Their names along with their descriptions are listed below (in brackets name used by distribution specific package as RPM is provided):

- `initdb.sh` (`unicore-uvos-server-initdb`) - initializes the db. This script can also be used to clean an existing database, and therefore it should be used with an extreme caution!
- `startServer.sh` (`unicore-uvos-server-startServer`) - starts a server in the background.
- `stopServer.sh` (`unicore-uvos-server-stopServer`) - stops a running server.
- `createExampleContents.sh` (`unicore-uvos-server-createExampleContents`) - creates an example contents of the service, as it is presented in the accompanying overview document. You should use it just after initializing the database (using the `initdb.sh`) and before starting the server.

13 Upgrading UVOS server

In most cases UVOS can be upgraded quite easily without changing database schema. This section provides information on upgrade in general and also contains notes on upgrades between specific versions.

13.1 Version specific notes

Version 1.4.1 - It is enough to update libraries in the `lib/` folder.

Version 1.4.0 - It is required to update libraries in `lib/` folder (`lib/endorsed` directory must be added too) and additionally `bin/_setenv.sh` script must be updated. Also `crackcheck.properties` file must be added to `conf/` directory. Of course it is suggested to set it up.

Version 1.3.3 - It is only needed to update libraries in `lib/` folder.

Version 1.3.2 - Upgrade from 1.3.1 is quite easy: there is no need to update database (the format was not altered). You must update: * libraries, * start and stop scripts, * configuration file (see documentation for details): * possible values of property `uvos.server.authn.order` are now: `TLS` and `HTTP` in any order. * new boolean property should be defined (default is false): `uvos.server.authn.mapTLSCertToDNFirst`

Version 1.3.1 - It is only needed to update libraries in `lib/` folder.

Version 1.3 - Upgrade from 1.2 is easy. There is no need to update database (the format was not altered). Configuration files are mostly unchanged: only the new options were added for UNICORE registration support and for enabling/disabling usage of X509 cert identities as DN-type identities.

Version 1.2 - Use generic upgrade instructions (below)

Version 1.1 - Use generic upgrade instructions (below)

Version 1.0 - it was an initial release so there is no upgrade possibility.

13.2 Generic upgrade instructions

13.2.1 Database upgrade

In order to do upgrade from older version of the server to the current one you will sometimes have to update your database. This manual assumes that you installed a new version of UVOS in a new directory (what is strongly recommended). It is not possible to use a different DB engine when upgrading (e.g. if old UVOS used postgresSQL the new one have to use it too).

Required actions:

1. Stop the old UVOS server.
2. (!!) Backup your database used by the old UVOS. If you use embedded DB (HSQL) you can skip this step.
3. Install and configure (at least DB settings in datamap.properties) the new UVOS installation. However do not start it or invoke initdb.sh script. → in case of postgresSQL specify the same settings as was used by the old UVOS installation. → in case of HSQL db just copy (copy, not move!) the contents of its data/ folder to the data/ folder of the new installation.
4. Eventually you have to upgrade your DB. There is a script which does the job. It can upgrade from any older version to the current one, however only in steps.

EXAMPLE To perform an upgrade from version 1.0 to 1.2 you will have to first upgrade to 1.1 and then to 1.2. To do so invoke:

```
# bin/updateDbVersion.sh 1.0 1.1
# bin/updateDbVersion.sh 1.1 1.2
```

If the script completes without errors you are done! Otherwise you should contact support mailing list.

13.2.2 Re-configuration

We strongly advise to start from the configuration files provided in a new UVOS distribution and to update them manually to the previous settings. Simply copying the old configuration files and overwriting the new installation files can cause severe problems as often there are many configuration changes.

14 APPENDIX - permission requirements

This appendix lists all permissions and other rules that are required to invoke UVOS functions. The column "Required permissions" lists the names of permissions needed in the scope of the group to invoke a specified function. If there is no group involved or if there are any other restrictions an explanation is given in the "Other authorization rules" column. The label [Self Access] means that the function operates on an identity and if this identity is the same as the caller's identity then selfAccess authorization policy designator is valid.

Function	Short description	Required permissions	Other authorization rules
<i>Query functions</i>			
isMember (Identity who, Group group, boolean effective)	Checks if the given identity is a member of the given group.	read	[Self Access]
getAllGroups (Identity who, boolean implied)	Gets all groups , which the given identity is a member of.	read	[Self Access] Global permission is needed.
areEquivalent (Identity i1, Identity i2)	Checks if two identities represent the same entity.	identityCtl	[Self Access] Global permission is needed.
getAttributes (Element owner, String attribute, boolean effective, boolean includeScoped, boolean includeImplied)	Retrieves attributes for the given element (i.e. identity, group or identity in a group scope).	read	[Self Access] Without global read perm attributes valid in groups where caller has no read perm are filtered out.
getAllEquivalents (Identity who)	Retrieves all identities equivalent to the one given as a parameter.	identityCtl	[Self Access] Global permission is needed.
getGroupContent (Group group)	Retrieves the group contents.	read	Everybody can get the root's (/) contents.
getAllIdentities ()	Retrieves all identities stored in the database.	read	Global permission is needed.
<i>Query history functions</i>			
Those offer the same features as normal query function but in the past (time is specified as additional argument).			
Always fullRead permission in global scope is needed and in case of getAllEquivalents and areEquivalent identityCtl too.			
<i>Management functions</i>			
addGroup (Group parent, String name)	Adds a new group.	write	

Function	Short description	Required permissions	Other authorization rules
<code>removeGroup (Group toRemove, boolean recursive)</code>	Removes the given group.	write	Write permission is required for the removed group, all its subgroups and its parent group.
<code>copyGroup (Group toCopy, Group newParent, String newName, boolean deleteOriginal)</code>	Copies or moves the given group to the content of a different group.	write	Write permission is required for the copied group, all its subgroups, its old and new parents groups.
<code>addIdentity (Identity toAdd)</code>	Adds a new identity.	identityCtl OR write	Required permission must be valid globally.
<code>addIdentity (Identity toAdd, Identity equivalentIdentity)</code>	Adds a new identity, which represents the same entity as the one given as a parameter.	see →	Requires either global write perm or (global identityCtl perm + write perm for every group equivalent identity is a member of + the same or better global permissions as equivalent identity has).
<code>removeIdentity (Identity toRemove)</code>	Deletes an identity.	see →	Requires global write perm or (global identityCtl and write perm for every group toRemove is a member of + the same or better global permissions as equivalentIdentity).
<code>setAttribute (Element whom, Attribute toAdd, boolean update)</code>	Adds a new attribute.	write	For global attributes global permission is needed.
<code>removeAttribute (Element whom, String toRemove)</code>	Removes the attribute.	write	For global attributes global permission is needed.
<code>addToGroup (Identity toAdd, Group group)</code>	Adds the given identity to a group.	write	
<code>removeFromGroup (Identity toRemove, Group group)</code>	Removes the given identity from the given group.	write	

Function	Short description	Required permissions	Other authorization rules
setIdentityLabel (Identity toChange, String label)	Changes the label of the identity.	see →	Requires global write perm or (global identityCtl perm + write perm for all groups toChange is a member of + the same or better global permissions as equivalentIdentity).
getAttributeTypes ()	Returns a list of all types of attributes.	-	
getIdentityTypes ()	Returns a list of all types of identities.	-	
updateAttributeTypes (Element toUpdate, boolean clear)	Updates a list of attribute's types.	write	Requires no perm to add a new attribute type and global write otherwise.
disableAttribute (Element whose, String toDisable, String valueToDisable)	Temporary disables the given attribute's value.	fullRead	[Self Access]
enableAttribute (Element whose, String toEnable, String valueToEnable)	Enables a disabled earlier of the given attribute.	fullRead	[Self Access]
getDisabledAttribute (Element whose)	Returns a list of attributes with disabled values.	fullRead	[Self Access]
purgeHistoricalData (Element deleteFrom)	Permanently deletes all service data that is older then deleteFrom (exclusive).	write	Write must be valid in globally.
<i>Applications Management</i>			
updateApplicationForm (VOApplicationForm applicationDef, boolean update)	Adds or updates an application definition.	write	write must be valid for the group which is set in applicationDef.
getApplications (Integer formId, String status)	Lists all applications for the selected form and/or with selected status. Both filtering arguments may be null, which eliminates the constraint.	fullRead	Requires perm for the group, which is the application form's base. In case of getting applications of all forms a global fullRead is required.
submitApplication (VOApplication application)	Adds a new application.	-	

Function	Short description	Required permissions	Other authorization rules
processApplication(id, ApplicationActions action, String notes, boolean sendConfirmation, VOApplication application)	Process an application. This operation only marks the application accordingly but it doesn't add a new identity (it must be performed by client software manually).	write	write must be valid for the group which is set in applicationDef.
csrProcessedNotification(csr, boolean accepted, String certificate, boolean sendNotification)	Used to signal the server that the application with the given CSR should be updated, as the contained CSR was processed by a CA.	-	
<i>Authorization related functions</i>			
modifyPermissions (Group group, PermissionDesignator designator, Permissions permissions)	Modifies permissions of the group.	write	
checkPermissions (Group group, Identity whose)	Retrieves a set of permissions for the given identity in the group.	read	[Self Access]
checkMyPermissions (Group group)	Retrieves a set of permissions for the method caller identity in the given group.	read	[Self Access]
getGroupAuthZ (Group group, boolean effective)	Retrieves a specification of authZ settings of the given group.	write	
modifyAuthentication(id, Object newToken)	Changes authentication token of the given identity.	write	[Self Access] In self access mode no write permission is needed, otherwise global write is required.