UNICORE/X MANUAL

UNICORE Team

Document Version:	1.0.0
Component Version:	6.4.0
Date:	09 05 2011

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

Contents

The UNICORE/X server is the central component of a UNICORE site. It hosts the services such as job submission, job management, storage access, and provides the bridge to the functionality of the target resources, e.g. batch systems or file systems.

For more information about UNICORE visit http://www.unicore.eu.

1 Getting started

1.1 Prerequisites

To run UNICORE/X, you need the SUN or OpenJDK Java 6 (JRE or SDK). If not installed on your system, you can download it from http://java.oracle.com

UNICORE/X has been most extensively tested on Linux-like systems, but runs on Windows and MacOS as well.

Please note that

- to integrate into secure production environments, you will need access to a certificate authority and generate certificates for all your UNICORE servers.
- to interface with a resource management system like SGE or Torque, you need to install and configure the UNICORE TSI.
- to make your resources accessible outside of your firewalls, you should setup and configure a UNICORE Gateway.

All these configuration options will be explained in the manual below.

1.2 Installation

UNICORE/X can be installed from either a tar.gz or zip archive, or (on Linux) from rpm/deb packages.

To install from the tar.gz or zip archive, unpack the archive in a directory of your choice. You should then review the config files in the conf/ directory, and adapt paths, hostname and ports. The config files are commented, and you can also check the configuration guide Section 2

To install from a Linux package, please use the package manager of your system to install the archive.

Note
Using the Linux packages, you can install only a single UNICORE/X instance per machine (without manual changes). The tar.gz / zip archives are self contained, and you can easily install multiple servers per machine.

The following table gives an overview of the file locations for both tar.gz and Linux bundles.

Table 1: Directory Layout

Name in this manual	tar.gz, zip	rpm	Description
CONF	<basedir>/conf/	/etc/unicore/unicorex	Config files
LIB	<basedir>/lib/	/usr/share/unicore/unicorex	Java libraries
LOG	<basedir>/log/	/var/log/unicore/unicorex/	Log files
BIN	<basedir>/bin/	/usr/sbin/	Start/stop scripts
—	—	/etc/init.d/unicore-unicorex	Init script

1.2.1 Starting/Stopping

There are two scripts that expect to be run from the installation directory. To start, do

```
cd <basedir>
bin/start.sh
```

To stop:

```
cd <basedir>
bin/stop.sh
```

Using the init script on Linux, you would do (as root)

```
etc/init.d/unicore-unicorex start|stop
```

1.3 Overview of the main configuration options

UNICORE/X is a fairly complex software which has many interfaces to other UNICORE components and configuration options. This section tries to give an overview of what can and should be configured. The detailed configuration guide follows in the next chapters

1.3.1 Mandatory configuration

- Certificates and basic security: UNICORE uses X.509 certificates for all servers. For UNICORE/X these are configured in the wsrflite.xml config file
- Attribute sources: to map clients (i.e. X.509 certificates) to local attributes such as user name, groups and role, various attribute sources are available. For details, consult the attribute sources section Section 4.
- Backend / target system access: to access a resource manager like SGE or Torque, the UNICORE TSI needs to be installed and UNICORE/X needs to be configured accordingly. Please consult the XNJS section Section 6.

2 Configuration of UNICORE/X

UNICORE/X has several sub-components. These are configured using several config files residing in the CONF directory, see the installation guide Section 1 for the location of the CONF directory.

2.1 Config file overview

The following table indicates the main configuration files. Depending on configuration and installed extensions, some of these files may not be present, or more files may be present.

UNICORE/X watches some most configuration files for changes, and tries to reconfigure if they are modified, at least where possible. This is indicated in the "dynamically reloaded" column. are indicated.

Table 2: UNICORE/X configuration files

config file	**usage**	**dynamically reloaded**
uas.config	General settings, startup behaviour, storages, AIP setup	yes
wsrflite.xml	Services to be deployed, SSL settings, Web server settings	yes
simpleidb	Backend, installed applications, resources	yes
xnjs.xml	Back end properties	no
xnjs_legacy.xml	Back end properties preconfigured for the Perl TSI	no
logging.properties	logging levels, logfiles and their properties	yes
xacml2Policies/*.xml	Access control policy for securing the web services	yes, via xacml2.config (do *touch xacml2.config* to trigger)
xacml2.config	Configure the XACML2 access control component	yes
vo.config	Configure the use of UVOS (optional attribute source)	no
simpleuudb	A file mapping user DNs to local attributes (optional attribute source)	yes

Table 2: (continued)

config file	usage	dynamically reloaded
jmxremote.password	Access control file for remote monitoring using the Java management extensions (JMX)	no

2.2 Settings for the UNICORE/X process

The properties controlling the Java virtual machine running the UNICORE/X process are configured in

- UNIX: the start script in the BIN directory
- Windows: the "conf\\wrapper.conf" configuration file

These properties include settings for maximum memory, and also the properties for configuring JMX, see the admin guide [?] for more on JMX.

General

2.3 Config file formats

UNICORE/X uses two different formats for configuration.

2.3.1 Java properties

- Each property can be assigned a value using the syntax "name=value"
- Please do not quote values, as the quotes will be interpreted as part of the value
- Comment lines are started by the "#"
- Multiline values are possible by ending lines with "\", e.g.

```
name=value1 \
value2
```

In this example the value of the "name" property will be "value1 value2".

2.3.2 XML

Various XML dialects are being used, so please refer to the example files distributed with UNICORE for more information on the syntax. In general XML is a bit unfriendly to edit, and it is rather easy to introduce typos.

Note
It is advisable to run a tool such as *xmllint* after editing XML files to check for typos

2.4 Integration of UNICORE/X into a UNICORE infrastructure

Since UNICORE/X is the central component, it is interfaced to other parts of the UNICORE architecture, i.e. the Gateway, the Registry and the TSI.

2.4.1 Gateway

The gateway address is usually hard-coded into CONF/wsrflite.xml, and on the gateway side there is an entry VSITE_NAME=address pointing to the UNICORE/X container. In some scenarios it's convenient to auto-register with a gateway. This can be enabled using the following properties.

Table 3: Gateway settings

config file	property name	range of values	description	modifyable at runtime
wsrflite.xml	Base URL	host/port	the host/port of the gateway	(yes)
uas.config	uas.gatewayregistration	true or false	whether autoregistration should be enabled	no
	uas.gatewayregistration.updateinterval	an integer	registration refresh interval in seconds	no
	uas.security.consignor.checksignature	true or false	whether gateway assertions must be signed (see also security)	no

Table 3: (continued)

config file	property name	range of values	description	modifyable at runtime
	uas.onstartup.wait	true or false	whether UNICORE/X should check if the gateway is up during startup	no

Note

To use the autoregistration feature, the gateway configuration must be set up accordingly

2.4.2 Registry

It is possible to configure UNICORE/X to contact one or more external or "global" Registries in order to publish information on crucial services there. Most of the following properties deal with the automatic discovery and/or manual setup of the external registries being used.

Table 4: Registry settings

config file	property name	range of values	description	modifyable at runtime
uas.config	uas.externalregistry.use	"true", "false"	whether to publish service information in an external registry	yes
	uas.externalregistry.url	a valid URL	use this URL for external registry if automatic discovery is switched off or fails	yes
	uas.externalregistry.url*	more valid URLs	additional registry URLs	yes

Table 4: (continued)

config file	property name	range of values	description	modifyable at runtime
wsrflite.xml	unicore.wsrflite.sg.defaulttermtime	a long value	alive-check interval for registry entries in seconds ; an external registry will enforce its own value	yes
	uas.externalregistry.autodiscover	"true", "false"	if set to "true", try to autodiscover the external registry via UDP multicast	yes

2.5 Startup code

In order to provide a flexible initialization process for the UAS, we introduce a property named "uas.onstartup", which is defined in the file "uas.config". The value string of this property consists of a whitespace separated list of java classes which must be implementing the "Runnable" interface. Many extensions for UNICORE/X rely on an entry in this property to initialise themselves.

Table 5: Startup code

class name	description	usage
de.fzj.unicore.uas.util.DefaultOnStartup	Initialises the job management system and the "local" registry; should usually be run on startup	normal UNICORE/X servers
de.fzj.unicore.bes.util.BESOnStartup	Initialises the OGSA-BES job management system	UNICORE/X servers that expose BES services
de.fzj.unicore.cisprovider.impl.InitOnStartup	Sets up the CIS infoprovider	UNICORE/X servers that want to provide information in GLUE2 format or want to be visible in the CIS

Table 5: (continued)

class name	description	usage
com.intel.gpe.gridbeans.PublishGridBeanService	if available, publishes the GridBeanService to the registry	UNICORE/X servers that host a Gridbean service
de.fzj.unicore.uas.util.CreateSMSOnStartup	creates and deploys a single instance of the SMS that is shared between users, named default_storage	if a shared storage is required

2.6 Security

2.6.1 Overview

Security is a complex issue, and many options exist. On a high level, the following items need to be configured.

- SSL setup (keystore and truststore settings for securing the basic communication between components)
- Attribute sources configuration (which will map Grid users to local properties such as role, Unix login and groups)
- Access control setup (controlling in detail who can do what on which services)
- Message level security (message signatures)

2.6.2 SSL configuration

Here you configure the server identity and the certificates of other services that want to contact this server.

Table 6: SSL configuration

config file	property name	range of values	description
wsrflite.xml	unicore.wsrflite.ssl.keystore	Name of keystore file	The keystore must contain at least one private/public keypair
	unicore.wsrflite.ssl.keytype	PKCS12, JKS	Keystore type

Table 6: (continued)

config file	property name	range of values	description
	unicore.wsrflite.ssl.keypass	Keystore password	
	unicore.wsrflite.ssl.keyalias	Alias of the key to use	
	unicore.wsrflite.ssl.clientauth	"true" or "false"	Whether to require client-authentication
	unicore.wsrflite.ssl.truststore	Name of truststore file	The truststore contains certificates that are trusted by the server
	unicore.wsrflite.ssl.truststoretype	PKCS12, JKS	
	unicore.wsrflite.ssl.truststorepass	Truststore password	

2.6.3 Attribute sources configuration

Attribute sources provide information about which local role and properties a Grid user has. UNICORE knows several attribute sources which can be combined using various combining algorithms. These are configured in the uas.config file. Due to the complexity, the description of the configuration options can be found in a separate chapter Section 4.

2.6.4 Access control configuration

Access control works by checking a Grid user’s attributes (obtained from the attribute sources) against a set of policies. Again, several options exist, which are described in a separate chapter Section 8.

2.6.5 Message signatures

UNICORE/X will require important messages (like job submissions or file exports). The property controlling this is - uas.security.signatures If set to true, signatures are required.

2.7 Configuring the XNJS and TSI

Information on the configuration of the XNJS and TSI backend can be found here Section 6.

2.8 Configuring storages on TargetSystem instances

Each TargetSystem instance can have one or more storages attached to it. Usually, only the HOME storage is created, which allows users access their home directory on the target system. You can add storages easily, using configuration entries in uas.config.

Table 7: Additional storages

config file	property name	range of values	description
uas.config	uas.targetsystem.storage.N	String	disambiguates several configuration entry sets
	uas.targetsystem.storage.N.type	FIXEDPATH, VARIABLE or CUSTOM	FIXEDPATH: mapped to a fixed directory, VARIABLE: resolved using a lookup, CUSTOM: specified class is used
	uas.targetsystem.storage.N.path	String	Denotes a path or the name of an environment variable (depending on the type)
	uas.targetsystem.storage.N.class	Java class name	Java class to use, only necessary when type is CUSTOM
	uas.targetsystem.storage.N.protocols	Space-separated protocol names	Which protocols to enable, default is "BFT RBYTEIO"

Here, "N" stands for an identifier (e.g. 1,2, 3, . . .) to distinguish the storages. For example, to configure two storages, one named TEMP pointing to "/tmp" and the other named DEISA_HOME pointing to "$DEISA_HOME", you would add the following configuration entries in uas.config:

```
uas.targetsystem.storage.1=TEMP
uas.targetsystem.storage.1.type=FIXEDPATH
uas.targetsystem.storage.1.path=/tmp
uas.targetsystem.storage.1.protocols=UFTP BFT
uas.targetsystem.storage.2=DEISA_HOME
uas.targetsystem.storage.2.type=VARIABLE
uas.targetsystem.storage.2.path=$DEISA_HOMES

# example for a custom SMS implementation (e.g. for Hadoop or iRODS ←↩
    )
uas.targetsystem.storage.3=IRODS
uas.targetsystem.storage.3.type=CUSTOM
uas.targetsystem.storage.3.path=/
uas.targetsystem.storage.3.class=my.custom.sms.ImplementationClass
```

Note that you can optionally control the file transfer protocols that should be enabled for each storage.

2.8.1 Controlling target system's storage resources

By default storage resource names (used in storage address) are formed from the owning user's xlogin and the storage type name, e.g. "someuser-Home". This is quite useful as users can write a URL of the storage without prior searching for its address. However if the site's user mapping configuration, maps more then one grid certificate to the same xlogin then this solution is not acceptable: only the first user connecting would be able to access her/his storage. This is as resource owners are expressed as grid user names (certificate DNs) and not xlogins. To have an unique, but dynamically created and non user friendly names of storages (and solve the problem of non-unique DN mappings) set this option in uas.config:

```
uas.targetsystem.storage.force_unique_ids=true
```

If you want to disable the default "Home" storage, you can set the following property in uas.config:

```
uas.targetsystem.home.disable=true
```

2.9 Configuring the StorageFactory service

The StorageFactory service allows clients to dynamically create storage instances. These can have different types, for example you could have storages on a normal filesystem, and other storages on an Apache Hadoop cluster.

The basic property controls which storage types are supported

```
uas.storagefactory.types=TYPE1 TYPE2 ...
```

Each supported storage type is configured using a set of properties

```
uas.storagefactory.TYPE1.description=GPFS file system
uas.storagefactory.TYPE1.fixedpath=GPFS file system
uas.storagefactory.TYPE1.path=/mnt/gpfs/unicore/unicorex-1/storage- ↩
    factory
uas.storagefactory.TYPE1.protocols=UFTP BFT

# if this is set to true, the directory corresponding to a storage  ↩
    instance will
# be deleted when the instance is destroyed. Defaults to "true"
uas.storagefactory.DEFAULT.cleanup=true
```

The "path" parameter determines the base directory used for the storage instances (i.e. on the backend), and the unique ID of the storage will be appended automatically.

The "cleanup" parameter controls whether the storage directory will be deleted when the storage is destroyed.

The "protocols" parameter controls which file transfer protocols should be enabled. By default, "BFT" and "RBYTEIO" will be enabled.

If you have a custom storage type, an additional "class" parameter defines the Java class name to use. For example:

```
uas.storagefactory.TYPE1.type=CUSTOM
uas.storagefactory.TYPE1.class=de.fzj.unicore.uas.hadoop. ↩
    SMSHadoopImpl
```

2.10 HTTP proxy, timeout and web server settings

The UNICORE Services Environment container has a number of settings related to the web server and to the HTTPClient library used for outgoing HTTP(s) calls. These are shown in the following two tables.

Table 8: Web server options

property name	range of values	default value	description
unicore.wsrflite.jetty.maxThreads	Integer	255	Maximum number of threads for Jetty
unicore.wsrflite.jetty.minThreads	Integer	1	Minimum number of threads
unicore.wsrflite.jetty.maxIdleTime	Integer	1000	Milliseconds before an idle connection will be timed out

Table 8: (continued)

property name	range of values	default value	description
unicore.wsrflite.jetty.lowThreads	Integer	50	If the number of free threads is below this value, idle connections will be timed out quicker
unicore.wsrflite.jetty.lowResourceMaxIdleTime	Integer	100	under "low resource" condition, milliseconds before an idle connection will be timed out
unicore.wsrflite.jetty.maxIdleTime	Integer	1000	Milliseconds before an idle connection will be timed out
unicore.wsrflite.jetty.gzip.minGzipSize	Integer	65535	The size of the largest data chunk that will not be compressed (if the client supports gzip)
unicore.wsrflite.jetty.gzip.bufferSize	Integer	8192	Buffer size used for gzip compression

Table 9: Outgoing HTTP call options

property name	range of values	default value	description
http.connection.timeout	Integer	10000	Socket connection timeout in millis
http.timeout	Integer	10000	Socket read timeout in millis
http.proxyHost	String		HTTP proxy host
http.proxyPort	Integer		HTTP proxy port
http.proxy.user	String		Proxy server user
http.proxy.password	String		Proxy server password
http.nonProxyHosts	String		Space separated list of host name fragments which are not proxied

3 Security concepts in UNICORE/X

This section describes the basic security concepts and architecture used in UNICORE/X. The overall procedure performed by the security infrastructure can be summarised as follows:

- the incoming message is authenticated by the SSL layer
- extract the information used for authorisation from the message sent to the server. This information includes: originator of the message(in case the message passed through a UNICORE gateway), trust delegation tokens, incoming VO membership assertions, etc.
- deal with trust delegation
- generate or lookup attributes to be used used for authorisation in the configured attribute sources Section 4
- perform policy check by executing a PDP Section 8 request

All these steps can be switched on/off, and use pluggable components. Thus, the security level of a UNICORE/X server is widely configurable

3.1 Security concepts

3.1.1 Identity

A server has a certificate, which is used to identify the server when it makes a web service request. This certificate resides in the server keystore, and can be configured in the usual config file (see the configuration reference Section 2.

3.1.2 Security tokens

When a client makes a request to UNICORE/X, a number of tokens are read from the message headers. These are placed in the security context that each WSRF instance has. Currently, tokens are the certificates for the UNICORE consignor and user, if available. Also, trust delegation assertions are read, and it is checked if the message is signed.

3.1.3 Resource ownership

Each service is *owned* by some entity identified by a distinguished name (X500 Principal). By default, the server is the owner. When a resource is created on user request (for example when submitting a job), the user is the owner.

3.1.4 Trust delegation

When the user and consignor are not the same, UNICORE/X will check whether the consignor has the right to act on behalf of the user. This is done by checking whether a trust delegation assertion has been supplied and is valid.

3.1.5 Attributes

UNICORE/X retrieves user attributes using either a local component or a remote service. In the default configuration, the XUUDB attribute service is contacted. See the attribute sources guide Section 4 for more information.

3.1.6 Policy checks

Each request is checked based on the following information.

- available security tokens
- the resource owner
- the resource accessed (e.g. service name + WSRF instance id)
- the activity to be performed (the web method name such as GetResourceProperty)

The validation is performed by the PDP (Policy Decision Point). The default PDP uses a list of rules expressed in XACML 2.0 format that are configured for the server. The PDP section Section 8 describes how to configure different engines for policy evaluation including a remote one.

3.1.7 Authorisation

A request is allowed, if the PDP allows it, based on the user's attributes.

3.1.8 Proxy certificate support

UNICORE clients can be configured to create a proxy certificate and send it to the server. On the server, the proxy can be used to invoke GSI-based tools. Please read the proxy section Section 10 about the configuration details.

4 Configuring attribute sources

The authorization process in UNICORE/X requires that each Grid user (identified by an X.509 certificate or just the DN) is assigned some *attributes* such as her *role*.

So the single most important item for security configuration is selecting and maintaining a so called *attribute source*, which is used by UNICORE/X to assign attribtes to Grid users.

Several attribute sources are available, that can even be combined for maximum flexibility and administrative control.

To configure the attribute sources, the `uas.security.attributes.order` property in the *uas.config* file is used. This is a space-separated list with attribute sources names, where the named attribute sources will be queried one after the other, allowing you to query multiple attribute sources, override values etc.

A second property, `uas.security.attributes.combiningPolicy`, allows you to control how attributes from different sources are combined.

For example, the following configuration snippet

```
#
# Authorisation attribute source configuration
#
uas.security.attributes.order=XUUDB FILE

#
# Combining policy
#
# MERGE_LAST_OVERRIDES (default), FIRST_APPLICABLE,  ←↩
    FIRST_ACCESSIBLE or  MERGE
uas.security.attributes.combiningPolicy=MERGE_LAST_OVERRIDES
```

will declare two attribute sources, "XUUDB" and "FILE", which should be both queried and combined using the MERGE_LAST_OVERRIDES policy.

Since all attribute sources will be queried, it has to be defined how attributes will be combined. For example, assume you have both XUUDB and FILE, and both return a xlogin attribute for a certain user, say "xlogin_1" and "xlogin_2".

The different combining policies are

- MERGE_LAST_OVERRIDES : new attributes override those from previous sources. In our example, the result would be "xlogin_2".
- FIRST_APPLICABLE : the attributes from the first source that returned a non empty list of attributes are used. In our case this would be "xlogin_1". If there were no xlogin attribute for the user in XUUDB then "xlogin_2" would be returned.
- FIRST_ACCESSIBLE : the attributes from the first source that is accessible are used. In our case this would be "xlogin_1". This policy is useful for redundant attribute sources. E.g. you

can configure two instances of XUUDB with the same users data; the 2nd one will be tried only if the first one is down.

- MERGE : attributes are merged. In our example, the result would be "xlogin_1, xlogin_2", and the user would be able to choose between them.

Each of the sources needs a mandatory configuration option defining the Java class, and several optional properties that configure the attribute source. In our example, one would need to configure both the "XUUDB" and the "FILE" source:

```
uas.security.attributes.XUUDB.class=...
uas.security.attributes.XUUDB.xuudbHost=...
...

uas.security.attributes.FILE.class=...
uas.security.attributes.FILE.file=...
...
```

Additionally you can mix several combining policies together, see "Chained attribute source" below for details.

4.1 XUUDB

The XUUDB is the standard option in UNICORE. It has the following features

- Web service interface for querying and administration. It is suitable for serving data for multiple clients. Usually it is deployed to handle attributes for a whole grid site.
- Access protected by client-authenticated SSL
- Supports the `xlogin`, `role` and `project` attributes (where `project` maps to Unix groups)
- Multiple xlogins per certificate or DN, where the user can select one
- Entries are grouped using the so-called *Grid component ID* (GCID). This makes it easy to assign users different attributes when accessing different UNICORE/X servers.

To enable and configure the XUUDB, set the following properties in *uas.config*.

```
uas.security.attributes.order=... XUUDB ...
uas.security.attributes.XUUDB.class=eu.unicore.uas.security. ↩
    XUUDBAuthoriser
uas.security.attributes.XUUDB.xuudbHost=https://<xuudbhost>
uas.security.attributes.XUUDB.xuudbPort=<xuudbport>
uas.security.attributes.XUUDB.xuudbGCID=<your_gcid>
```

Full XUUDB documentation is available from http.://www.unicore.eu/documentation/manuals/xuudb

4.2 UVOS

The UNICORE Virtual Organisation Service (UVOS) is a powerful tool for managing access based on the concept of virtual organisations. A detailed description and configuration guidance can be obtained from http://unicore-dev.zam.kfa-juelich.de/documentation/xfire-voutils-1.4.0/manual.html

4.3 File attribute source

In simple circumstances, or as an addition to a XUUDB or UVOS, the file attribute source can be used. As the name implies a simple map file is used to map DNs to xlogin, role and other attributes. It is useful when you don't want to setup an additional service (XUUDB or UVOS) and also when you want to locally override attributes for selected users (e.g. to ban somebody).

To use, set

```
uas.security.attributes.order=... FILE ...
uas.security.attributes.FILE.class=eu.unicore.uas.security.file. ↩
    FileAttributeSource
uas.security.attributes.FILE.file=<your map file>
uas.security.attributes.FILE.matching=<strict|regexp>
```

The map file itself has the following format:

```
<?xml version="1.0" encoding="UTF-8"?>
<fileAttributeSource>
   <entry key="USER DN">
      <attribute name="role">
         <value>user</value>
      </attribute>
      <attribute name="xlogin">
         <value>unixuser</value>
         <value>nobody</value>
         ...
      </attribute>
      ...
   </entry>
   ...
</fileAttributeSource>
```

You can add an arbitrary number of attributes and attribute values.

The *matching* option controls how a client's DN is mapped to a file entry. In *strict* mode, the canonical representation of the key is compared with the canonical representation of the argument. In *regexp* mode the key is considered a Java regular expression and the argument is matched with it. When constructing regular expressions a special care must be taken to construct the regular expression from the canonical representation of the DN. The canonical representation is defined here. (but you don't have to perform the two last upper/lower case operations). In 90% of all cases (no multiple attributes in one RDN, no special characters, no uncommon attributes) it just means that you should remove extra spaces between RDNs.

The evaluation is simplistic: the first entry matching the client is used (which is important when you use regular expressions).

The attributes file is automatically refreshed after any change, before a subsequent read. If the syntax is wrong then an error message is logged and the old version is used.

Recognized attribute names are:

- xlogin
- role
- group
- supplementaryGroups
- addOsGroups
- queue

Attributes with those names (case insensitive) are handled as special UNICORE attributes (e.g. the `xlogin` is used to provide available local OS user names for the client). You can define other attributes but those will be ignored.

4.4 Chained attribute source

Chained attribute source is a meta source which allows you to mix different combining policies together. It is configured as other attribute sources with two parameters (except of its class): order and combiningPolicy. The result of the chain attribute source is the set of attributes returned by the configured chain.

Let's consider the following example situation where we want to configure two redundant UVOS servers (both serving the same data) to achieve high availability. Additionally we want to override settings for some users using a local file attribute source (e.g. to ban selected users).

```
# The main chain configuration:
uas.security.attributes.order=UVOS_CLUSTER FILE
uas.security.attributes.combiningPolicy=MERGE_LAST_OVERRIDES

# The FILE source cfg:
uas.security.attributes.FILE.class=eu.unicore.uas.security.file. ↩
    FileBasedAuthoriser
uas.security.attributes.FILE.file=<your map file>

# The UVOS_CLUSTER is a sub chain:
uas.security.attributes.UVOS_CLUSTER.class=de.fzj.unicore.uas. ↩
    security.util.AttributeSourcesChain
```

```
uas.security.attributes.UVOS_CLUSTER.order=UVOS1 UVOS2
uas.security.attributes.UVOS_CLUSTER.combiningPolicy= ↩
    FIRST_ACCESSIBLE

# And configuration of the two real sources used in the sub chain:
uas.security.attributes.UVOS1.class=...
...
uas.security.attributes.UVOS2.class=...
...
```

5 The UNICORE persistence layer

UNICORE/X stores its state in data bases. The information that is stored includes

- user's resources (instances of storage, job and other services)
- jobs on the XNJS

The job directories themselves reside on the target system, but UNICORE keeps additional information (like, which Grid user owns a particular job).

The data on user resources is organised by service name, i.e. each service (for example, Job-Management) stores its information in a separate database table (having the same name as the service, e.g. "JobManagement").

Job information is stored in a table named "JOBS", while finished jobs go into a table called "FINISHED_JOBS".

The UNICORE persistence layer offers two kinds of storage:

- on the filesystem of the UNICORE/X server (using the H2 database engine)
- on a database server (MySQL, or the so-called server mode of H2)

While the first one is very easy to setup, and easy to manage, the second option allows advanced setups like clustering/load balancing configurations involving multiple UNICORE/X servers sharing the same persistent data.

Data migration from one database system to another is in principle possible, but you should select the storage carefully before going into production. In general, if you do not require clustering/load balancing, you should choose the default filesystem option, since it is less administrative effort.

5.1 Configuring the persistence layer

Peristence properties are configured in two files:

- wsrflite.xml for all service data
- xnjs.xml (or xnjs_legacy.xml) for job data

It is recommended to specify a configuration file using the `persistence.config` property. Thus, persistence configuration can be easily shared between the components (XNJS and WSRFlite). If the "persistence.config" property is set, the given file will be read as a Java properties file, and the properties will be used. However, "local" properties will override the ones given in the file.

Note
All properties can be specified on a "per table" basis, by appending ".<TABLENAME>" to the property name. This means you can even select different storage systems for different data, e.g. store service data on the filesystem and jobs in MySQL. The table name is case-sensitive.

Table 10: Basic configuration options

property name	**default value**	**description**
persistence.config		name of a config file to read
persistence.class	de.fzj.unicore.persist.impl.H2Persist	name of the Java class
persistence.driver	(implemention class may provide their own)	Java class name of the JDBC driver
persistence.database		name of the database to connect to
persistence.user		database user name
persistence.password		database password
persistence.host	localhost	database server host
persistence.port	depends on the implementation	database server port

5.1.1 Caching

By default, caching of data in memory is enabled. It can be switched off and configured on a per-table (i.e. per entity class) basis. If you have a lot of memory for your server, you might consider increasing the cache size for certain components.

The following properties are used to control the caching behaviour

Table 11: Caching options

property name	range of values	default value	description
persistence.cache.enable	String	"true"	"true" or "false" to enable/disable caching for the given table
persistence.cache.maxSize	Integer	50	maximum number of elements to keep in the cache

For example, to set the maximum size of the JOBS cache to 1000, you'd configure

```
persistence.cache.maxSize.JOBS
```

5.1.2 The H2 engine

H2 is a pure Java database engine. It can be used in embedded mode (i.e. the engine runs in-process), or in server mode, if multiple UNICORE servers should use the same database server. For more information, visit http://www.h2database.com

Table 12: Additional configuration for H2

property name	range of values	default value	description
persistence.directory	String		name of the directory for storing data
persistence.h2.server_mode	"true" or "false"	false	whether to connect to a h2 server using tcp
persistence.h2.cache_size	String denoting an Integer	4096	in-memory cache size for H2 in kilobytes

Connection URL

In embedded mode (i.e. the default non-server mode), the connection URL is constructed from the configuration properties as follows

```
  jdbc:h2:file:<persistence.directory>/<table_name>
```

In server mode, the connection URL is constructed as follows

```
  jdbc:h2:tcp://<persistence.host>:<persistence.port>/<persistence. ↩
      directory>/<table_name>
```

5.1.3 The MySQL Engine

The MySQL database engine does not need an introduction. To configure its use for UNICORE persistence data, you need to set

```
persistence.class=de.fzj.unicore.persist.impl.MySQLPersist
```

To use MySQL, you need access to an installed MySQL server. It is beyond the scope of this guide to describe in detail how to setup and operate MySQL. The following is a simple sequence of steps to be performed for setting up the required database structures.

- open the mysql console
- create a dedicated user, say *unicore* who will connect from some server in the domain "yourdomain.com" or from the local host:

```
CREATE USER ’unicore’@’%.yourdomain.com’ identified by ’ ↩
    some_password’ ;
CREATE USER ’unicore’@’localhost’ identified by ’some_password’ ;
```

- create a dedicated database for use by the UNICORE/X server:

```
CREATE DATABASE ’unicore_data_demo_site’;
USE ’unicore_data_demo_site’;
```

- allow the unicore user access to that database:

```
GRANT ALL PRIVILEGES ON ’unicore_data_demo_site’ to ’unicore’@’ ↩
    localhost’;
GRANT ALL PRIVILEGES ON ’unicore_data_demo_site’ to ’unicore’@’%. ↩
    yourdomain.com’;
```

The UNICORE persistence properties would in this case look like this:

```
persistence.class=de.fzj.unicore.persist.impl.MySQLPersist
persistence.database=unicore_data_demo_site
persistence.user=unicore
persistence.password=some_password
persistence.host=<your_mysql_host>
persistence.port=<your_mysql_port>
```

Table 13: Additional configuration for MySQL

property name	**default value**	**description**
persistence.mysql.tabletype	MyISAM	MySQL table type to use

5.2 Clustering

If you intend to run a configuration with multiple UNICORE servers accessing a shared database, you need to enable clustering mode by setting a property

```
persistence.cluster.enable=true
```

The clustering config file can be set using a (per-table) property

```
persistence.cluster.config=<path to config file>
```

If this is not set, a default configuration is used.

For clustering, the Hazelcast library is used (http://www.hazelcast.com/documentation.jsp). A simple TCP based configuration is

```
<hazelcast>
    <group>
        <name>dev</name>
        <password>dev-pass</password>
    </group>
    <network>
        <port auto-increment="true">5701</port>
        <join>
            <multicast enabled="false">
                <multicast-group>224.2.2.3</multicast-group>
                <multicast-port>54327</multicast-port>
            </multicast>
            <tcp-ip enabled="true">
                <interface>127.0.0.1</interface>
            </tcp-ip>
        </join>
        <interfaces enabled="false">
            <interface>10.3.17.*</interface>
        </interfaces>
    </network>
    <executor-service>
        <core-pool-size>16</core-pool-size>
        <max-pool-size>64</max-pool-size>
        <keep-alive-seconds>60</keep-alive-seconds>
    </executor-service>
    <queue name="default">
        <max-size-per-jvm>10000</max-size-per-jvm>
        <time-to-live-seconds>0</time-to-live-seconds>
    </queue>
    <map name="default">
        <backup-count>1</backup-count>
        <eviction-policy>NONE</eviction-policy>
        <max-size>0</max-size>
        <eviction-percentage>25</eviction-percentage>
    </map>
</hazelcast>
```

The most important part is the "tcp-ip" setting, which must list at least one other node in the cluster. The "group" setting allows to run multiple clusters on the same set of hosts, just make sure that the group name is the same for all nodes in a cluster.

Most of the other settings (map, executor-service, etc) are currently not important, because only the distributed lock feature of Hazelcast is used. Please read the Hazelcast documentation for further information.

6 Configuring the XNJS

The XNJS is the UNICORE/X component that deals with the actual job execution and file access. It is configured using an XML file named *xnjs.xml* or *xnjs_legacy.xml*. The actual file that is used is set in the uas.config property `uas.targetsystemfactory.xnjs.configfile`.

```
#
# in uas.config
#
uas.targetsystemfactory.xnjs.configfile=conf/xnjs.xml
```

Here’s an overview of the most important properties that can be set in this file.

Table 14: Main XNJS properties

config file	property name	range of values	default value	description
xnjs.xml	XNJS.filespace	an absolute path on the target system's filesystem	"data/FILESPACE"	the directory on the target system where job directories will be created
	XNJS.statedir	a path on the UNICORE/X machine	"data/NJSSTATE"	the directory on the UNICORE/X machine where the XNJS keeps its state
	XNJS.idbfile	a file or directory name	"conf/simpleidb"	the IDB containing application definitions etc.
	XNJS.numberofworkers	an integer	"4"	the number of worker threads used to process jobs

Most of the other settings in this file are used to configure the internals of the XNJS and should usually be left at their default values.

6.1 The UNICORE TSI

This section describes installation and usage of the UNICORE TSI. This is a mandatory step if you want to interface to batch systems such as Torque, SGE, or LoadLeveller to efficiently use a compute cluster.

> **Note**
> Without this component, all jobs will run on the UNICORE/X server, under the user id that started UNICORE/X.

In a nutshell, you have to perform the following steps

- Install the UNICORE TSI on your cluster front end node

- Edit the tsi.properties file
- On the UNICORE/X server, edit uas.config, simpleidb and xnjs_legacy.xml
- Start the newly installed TSI (as root in a multiuser setting)
- Restart UNICORE/X

6.1.1 Installation of the correct TSI

The TSI is a set of perl modules that is running on the target system. In case of a cluster system, you'll need to install it on the frontend machine(s), i.e. the machine from where your jobs are submitted to the batch system. There are different variants available for the different batch systems such as Torque or SGE.

Usually installation and start of the TSI will be performed as the root user. The TSI will then be able to change to the current Grid user's id for performing work (Note: nothing will ever be executed as "root"). You can also use a normal user, but then all commands will be executed under this user's id.

- First, download and install the UNICORE TSI package. The UNICORE core server bundle ("quickstart" package) includes the TSI in the *tsi* subdirectory. You should copy this folder to the correct machine first. In the following this will be denoted by <tsidir>
- Install the correct TSI variant by

```
cd <tsidir>
 ./Install.sh
```

When prompted for the path, choose an appropriate on, denoted <*your_tsi*> in the following

- Check the tsi file in

```
<tsidir>/<your_tsi>/perl/tsi
```

especially command locations, path settings etc.

- set permissions using

```
 cd <tsidir>
  ./Install_permissions.sh
```

- MAKE A NOTE of the exact location of the *tsi_ls* and *tsi_df* files *<tsidir>/<your_tsi>/tsi_ls* and *<tsidir>/<your_tsi>/tsi_df*

6.1.2 Required TSI Configuration

Configuration is done by editing <tsi_dir>/conf/tsi.properties The following settings are needed:

```
#path to your tsi installation
tsi.path=<tsi_dir>/<your_tsi>

#UNICORE/X machine
tsi.njs_machine=<UNICORE/X host>

#UNICORE/X listener port (check unicorex/conf/xnjs_legacy.xml  ↩
    variable "CLASSICTSI.replyport"
tsi.njs_port=7654

#TSI listener port (check unicorex/conf/xnjs_legacy.xml variable " ↩
    CLASSICTSI.port"
tsi.my_port=4433
```

6.1.3 UNICORE/X configuration

Edit *unicorex/conf/uas.config* and set the variable

```
uas.targetsystemfactory.xnjs.configfile=conf/xnjs_legacy.xml
```

Edit *unicorex/conf/xnjs_legacy.xml*. Check the filespace location, this is where the local job directories will be created. On a cluster, these have to be on a shared part of the filesystem.

Check the *CLASSICTSI* related properties. Set the correct value for the machine and the ports (these can usually be left at their default values)

Set the value of `CLASSICTSI.TSI_LS` to the path of *tsi_ls* as noted above.

Set the value of `CLASSICTSI.TSI_DF` to the path of *tsi_df* as noted above.

Here is an example section for the classic TSI properties.

```
    <eng:Property name="XNJS.tsiclass" value="de.fzj.unicore.xnjs. ↩
        legacy.LegacyTSI"/>
    <!-- TSI machine and ports used -->
    <eng:Property name="CLASSICTSI.machine" value="localhost"/>
    <eng:Property name="CLASSICTSI.port" value="4433"/>
    <eng:Property name="CLASSICTSI.replyport" value="7654"/>
    <!-- location of the tsi_ls file -->
    <eng:Property name="CLASSICTSI.TSI_LS" value="tsi/tsi_ls"/>
    <!-- location of the tsi_df file -->
    <eng:Property name="CLASSICTSI.TSI_DF" value="tsi/tsi_df"/>
    <!-- commands on the target system -->
    <eng:Property name="CLASSICTSI.CP" value="/bin/cp"/>
    <eng:Property name="CLASSICTSI.RM" value="/bin/rm"/>
    <eng:Property name="CLASSICTSI.MV" value="/bin/mv"/>
    <eng:Property name="CLASSICTSI.MKDIR" value="/bin/mkdir -p"/>
    <eng:Property name="CLASSICTSI.CHMOD" value="/bin/chmod"/>
    <eng:Property name="CLASSICTSI.MKFIFO" value="/usr/bin/mkfifo ↩
        "/>
    <eng:Property name="CLASSICTSI.PERL" value="/usr/bin/perl"/>
    <!-- interval between updates of job stati (milliseconds) -->
    <eng:Property name="CLASSICTSI.statusupdate.interval" value ↩
        ="5000"/>
    <!-- how often the XNJS will re-try to get the status of a job
         in case the job is not listed in the status listing -->
    <eng:Property name="CLASSICTSI.statusupdate.grace" value="0"/>
    <!-- a user that is allowed to see all jobs on the batch system ↩
         -->
    <eng:Property name="CLASSICTSI.priveduser" value="someuser"/>
    <!-- I/O buffer size, will greatly impact filetransfer  ↩
        performance -->
    <eng:Property name="CLASSICTSI.BUFFERSIZE" value="1000000"/>
```

6.1.4 Additional parameters

Some additional parameters exist for tuning the XNJS-TSI communication.

Table 15: XNJS-TSI communication settings

property name	range of values	default value	description
CLASSICTSI.BUFFERSIZE	integer	1000000	Buffersize for filetransfers in bytes
CLASSICTSI.socket.timeout	integer	300000	Socket timeout in milliseconds
CLASSICTSI.socket.connect.timeout	integer	10000	Connection timeout in milliseconds

6.1.5 Tuning the batch system settings

UNICORE uses the normal batch system commands (e.g. qstat) to get the status of running jobs. There is a special case if a job is not listed in the qstat output. UNICORE will then assume the job is finished. However, in some cases this is not true, and UNICORE will have a wrong job status. To work around, there is a special property

```
    <!-- how often the XNJS will re-try to get the status of a job
         in case the job is not listed in the status listing -->
    <eng:Property name="CLASSICTSI.statusupdate.grace" value="2"/>
```

If the value is larger than zero, UNICORE will re-try to get the job status.

Start the TSI using (as root in a multiuser environment)

```
cd <tsi_dir>/conf
 ../bin/start_tsi
```

(or use the `unicore-tsi` init script if available in your installation)

Finally, restart the UNICORE/X server

Note
When changing TSIs, it's a good idea to remove the UNICORE/X state and any files before restarting. In the quickstart configuration, this is done using See the section on persistence Section 5 for details

6.1.6 Enabling SSL for the XNJS to TSI communication

The UNICORE/X server can be set up to use SSL for communicating with the Perl TSI. On the UNICORE/X side, this is very simple to switch on. In the XNJS config file, add a line to the "Core" section:

```
    <!-- enable SSL using the normal UNICORE/X key and trusted  ↩
        certificates -->
    <eng:LoadComponent>de.fzj.unicore.uas.xnjs. ↩
        XNJSSecurityConficuration</eng:LoadComponent>
```

On the TSI side it is a bit more complex, and you need to have the TSI from the 6.3.0 distribution or later installed. First of all, your Perl installation must include the module "IO::Socket:SSL" and its dependencies. If you do not have it, you can get it from the CPAN archive.

In the *tsi.properties* configuration file, you set the keystore and truststore to be used:

```
# SSL parameters
# Keystore must contain the private TSI key and certificate
# Trustore must contain the certificate of the CA
tsi.keystore=/certs/keystore.pem
tsi.keypass=yourpassword
tsi.truststore=/certs/keystore.pem
```

Both keystore and truststore are in pem format.

7 The IDB

The UNICORE IDB (incarnation database) contains information on how abstract job definitions are to be mapped onto real executables. This process (called "incarnation") is performed by the XNJS component. The second IDB function is advertising target system capabilities and allowing to check client resource requests against these.

The IDB is a (set of) XML files, which by default is called *simpleidb*.

For reference, the current XML schema for the IDB can be read from the SVN repository.

7.1 Applications

The most important functionality of the IDB is providing executables for abstract applications. An abstract application is given by name and version, whereas an executable application is given in terms of executable, arguments and environment variables.

7.1.1 Simple applications

Here is an example entry for the "Date" application on a UNIX system

```
 <idb:IDBApplication>
  <idb:ApplicationName>Date</idb:ApplicationName>
  <idb:ApplicationVersion>1.0</idb:ApplicationVersion>
  <jsdl:POSIXApplication xmlns:jsdl="http://schemas.ggf.org/jsdl ↩
      /2005/11/jsdl-posix">
    <jsdl:Executable>/bin/date</jsdl:Executable>
  </jsdl:POSIXApplication>
</idb:IDBApplication>
```

As can be seen, "Date" is simply mapped to "/bin/date".

7.1.2 Arguments

Command line arguments are specified using <Argument> tags:

```
 <idb:IDBApplication>
  <idb:ApplicationName>LS</idb:ApplicationName>
  <idb:ApplicationVersion>1.0</idb:ApplicationVersion>
  <jsdl:POSIXApplication xmlns:jsdl="http://schemas.ggf.org/jsdl ↩
      /2005/11/jsdl-posix">
    <jsdl:Executable>/bin/ls</jsdl:Executable>
    <jsdl:Argument>-l</jsdl:Argument>
    <jsdl:Argument>-t</jsdl:Argument>
  </jsdl:POSIXApplication>
 </idb:IDBApplication>
```

This would result in a command line "/bin/ls -l -t".

7.1.3 Conditional Arguments

The job submission from a client usually contains environment variables to be set when running the application. It often happens that a certain argument should only be included if a corresponding environment variable is set. This can be achieved by using "conditional arguments" in the incarnation definition. Conditional arguments are indicated by a quastion mark "?" appended to the argument value:

```
<idb:IDBApplication>
  <idb:ApplicationName>java</idb:ApplicationName>
  <idb:ApplicationVersion>1.5.0</idb:ApplicationVersion>
  <jsdl:POSIXApplication xmlns:jsdl="http://schemas.ggf.org/jsdl ↩
      /2005/11/jsdl-posix">
    <jsdl:Executable>/usr/bin/java</jsdl:Executable>
    <jsdl:Argument>-cp$CLASSPATH?</jsdl:Argument>
    <!-- other args omitted for clarity -->
  </jsdl:POSIXApplication>
 </idb:IDBApplication>
```

Here, `<jsdl:Argument>-cp$CLASSPATH?</jsdl:Argument>` is an optional argument.

If a job submission now includes a Environment variable named `CLASSPATH`

```
<jsdl:Environment name="CLASSPATH">myjar.jar</jsdl:Environment>
```

the incarnated commandline will be "/usr/bin/java -cp$CLASSPATH . . . ", otherwise just "/usr/bin/java . . . ".

This allows very flexible incarnations.

7.1.4 More

For more details about IDB application definitions, please consult the detailed application definition guide Section 7.4.

7.2 TargetSystemProperties

The TargetSystemProperties element contains information about a site's available resources, as well as additional information that should be published to clients.

7.2.1 Textual information

Simple strings can be entered into the IDB which are then accessible client-side. This is very useful for conveying system-specifics to client code and also to users. These text-info strings are entered into the IDB as a subtag of the `TargetSystemProperties` tag

Here is an example

```
<idb:TargetSystemProperties>

  <!-- text infos -->
  <idb:Info Name="Administator email">admin@site.org</idb:Info>

</idb:TargetSystemProperties>
```

These pieces of information are accessible client side as part of the target system properties.

7.2.2 Resources

Resources of a target system are specified using JSDLs Resource tag. It allows to specify things like number of nodes, CPUtime (per CPU), CPUs per node, total number of CPUs, etc.

These capabilities are specified giving an exact value and a range, for example:

```
  <jsdl:Exact>3600</jsdl:Exact>
  <jsdl:Range>
   <jsdl:LowerBound>1</jsdl:LowerBound>
   <jsdl:UpperBound>86400</jsdl:UpperBound>
  </jsdl:Range>
```

The Range gives upper and lower bounds, where as the Exact value is interpreted as the DEFAULT, when the client does not request anything specific. If the Exact value is specified, the resource is part of the site's default resource set.

There exist a number of standard settings. You may choose to not specify some of them, if they do not make sense on your system. For example, some sites do not allow the user to explicitely select nodes and processors per node, but only "total number of CPUs".

- `jsdl:IndividualCPUTime` : The wall clock time.
- `jsdl:IndividualCPUCount` : The number of CPUs per node
- `jsdl:IndividualPhysicalMemory` : The amount of memory per node (in bytes)
- `jsdl:TotalResourceCount` : The number of nodes.
- `jsdl:TotalCPUCount` : The total number of CPUs.

7.2.3 "Total CPUs" vs. "Nodes and CPUs per node"

Users can specify the number of processors either as just "total number of CPUs", or they can give a value for both "nodes" and "CPUs per node". If both are given, the values containing more information (i.e. nodes + CPUs per node) are used.

Similarly, if the administrator specifies both possibilities with a default value in the IDB, the nodes + CPUs per node will have precedence.

7.2.4 Other types of resources

Most HPC sites have special settings that cannot be mapped to the generic JSDL elements shown in the previous section. Therefore UNICORE 6 includes a mechanism to allow sites to specify their own system settings and allow users to set these using the Grid middleware.

Custom resources are described in this section Section 7.8.

7.2.5 Example Resources section

This example includes the elements defining capabilities, and some informational elements like CPUArchitecture and operating system info.

```
<idb:TargetSystemProperties>
    <jsdl:Resources xmlns:jsdl="http://schemas.ggf.org/jsdl ↩
        /2005/11/jsdl">
      <jsdl:CPUArchitecture>
        <jsdl:CPUArchitectureName>x86</jsdl:CPUArchitectureName>
      </jsdl:CPUArchitecture>

        <!-- O/S -->
      <jsdl:OperatingSystem>
       <jsdl:OperatingSystemType>
         <jsdl:OperatingSystemName>LINUX</jsdl:OperatingSystemName>
       </jsdl:OperatingSystemType>
       <jsdl:OperatingSystemVersion>2.6.13</jsdl: ↩
           OperatingSystemVersion>
       <jsdl:Description>A free UNIX clone</jsdl:Description>
      </jsdl:OperatingSystem>

      <!-- cpu time (per cpu) in seconds -->
      <jsdl:IndividualCPUTime>
        <jsdl:Exact>3600</jsdl:Exact>
        <jsdl:Range>
         <jsdl:LowerBound>1</jsdl:LowerBound>
         <jsdl:UpperBound>86400</jsdl:UpperBound>
        </jsdl:Range>
      </jsdl:IndividualCPUTime>

      <!-- Nodes -->
      <jsdl:TotalResourceCount>
        <jsdl:Exact>1.0</jsdl:Exact>
        <jsdl:Range>
         <jsdl:LowerBound>1.0</jsdl:LowerBound>
         <jsdl:UpperBound>16.0</jsdl:UpperBound>
        </jsdl:Range>
      </jsdl:TotalResourceCount>

      <!-- CPUs per node -->
      <jsdl:IndividualCPUCount>
        <jsdl:Exact>8.0</jsdl:Exact>
        <jsdl:Range>
         <jsdl:LowerBound>1.0</jsdl:LowerBound>
         <jsdl:UpperBound>8.0</jsdl:UpperBound>
        </jsdl:Range>
      </jsdl:IndividualCPUCount>

      <!-- total CPUs -->
      <jsdl:TotalCPUCount>
        <jsdl:Exact>8.0</jsdl:Exact>
        <jsdl:Range>
         <jsdl:LowerBound>1.0</jsdl:LowerBound>
         <jsdl:UpperBound>128.0</jsdl:UpperBound>
        </jsdl:Range>
      </jsdl:TotalCPUCount>

      <!-- Memory per node (bytes) -->
      <jsdl:IndividualPhysicalMemory>
        <jsdl:Exact>268435456</jsdl:Exact>
        <jsdl:Range>
         <jsdl:LowerBound>1024576</jsdl:LowerBound>
         <jsdl:UpperBound>1073741824</jsdl:UpperBound>
        </jsdl:Range>
      </jsdl:IndividualPhysicalMemory>

    </jsdl:Resources>
  </idb:TargetSystemProperties>
```

7.3 Script templates

If you need to modify the scripts that are generated by UNICORE/X and sent to the TSI, you can achieve this using two entries in the IDB.

```
<idb:IDB xmlns:idb="http://www.fz-juelich.de/unicore/xnjs/idb">

<!-- Templates -->
<idb:SubmitScriptTemplate>
#!/bin/sh
#COMMAND
#RESOURCES
#SCRIPT
</idb:SubmitScriptTemplate>

<idb:ExecuteScriptTemplate>
#!/bin/sh
#COMMAND
#RESOURCES
```

```
#SCRIPT
</idb:ExecuteScriptTemplate>

<!-- rest of IDB omitted -->

</idb:IDB>
```

The SubmitScriptTemplate is used for batch job submission, the ExecuteScriptTemplate is used for everything else (e.g. creating directories, resolving user's home, etc)

UNICORE/X generates the TSI script as follows:

- the "#COMMAND" entry will be replaced by the action for the TSI, e.g. "#TSI_SUBMIT".
- the "#RESOURCES" will be replaced by the resource requirements, e.g. "#TSI_NODES=. . . "
- the "#SCRIPT" is the user script

Modifying these templates can be used to perform special actions, such as loading modules, or changing the shell (but use something compatible to *sh*). For example, to add some special directory to the path for user scripts submitted in batch mode, you could use

```
<idb:IDB xmlns:idb="http://www.fz-juelich.de/unicore/xnjs/idb">

<!-- Templates -->
<idb:SubmitScriptTemplate>
#!/bin/sh
#COMMAND
#RESOURCES
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/openmpi-2.1/lib; export  ↩
   LD_LIBRARY_PATH
PATH=$PATH:/opt/openmpi-2.1/bin; export PATH
#SCRIPT
</idb:SubmitScriptTemplate>

<!-- rest of IDB omitted -->

</idb:IDB>
```

7.3.1 Properties

In the IDB file, XNJS properties can be specified, for example the command locations identified by property names starting with "CLASSICTSI."

```
<idb:IDB xmlns:idb="http://www.fz-juelich.de/unicore/xnjs/idb">
<!--- rest of IDB omitted -->
 <idb:Property name="..."
               value="..."/>
 </idb:IDB
```

7.4 More on the IDB Application definitions

Simple application definitions and application arguments have already been covered in the previous section [?]. Here, more details are presented.

7.4.1 Pre and post-commands

Sometimes it is useful to be able to execute one or several commands before or after the execution of an application. For example, to add directories to the path, or perform some preprocessing. The IDB allows to specify these using the PreCommand and PostCommand tags.

For example

```
 <idb:IDBApplication>
   <idb:ApplicationName>java</idb:ApplicationName>
   <idb:ApplicationVersion>1.5.0</idb:ApplicationVersion>
   <jsdl:POSIXApplication xmlns:jsdl="http://schemas.ggf.org/jsdl ↩
       /2005/11/jsdl-posix">
     <jsdl:Executable>/usr/bin/java</jsdl:Executable>
     <jsdl:Argument>-cp$CLASSPATH?</jsdl:Argument>
     <!-- other args omitted for clarity -->
   </jsdl:POSIXApplication>
   <idb:PreCommand>PATH=$PATH:/opt/myapp/bin ; export PATH</idb: ↩
       PreCommand>
   <idb:PreCommand>/opt/example/aquire_license</idb:PreCommand>
   <idb:PostCommand>/opt/example/release_license</idb:PostCommand>
  </idb:IDBApplication>
```

These commands will be executed as part of the user's job script.

7.4.2 Interactive execution when using a batch system

If an application should not be submitted to the batch system, but be run on the login node (i.e. interactively), a flag in the IDB can be set:

```
 <idb:IDBApplication>
   <idb:ApplicationName>SomeApp</idb:ApplicationName>
   <idb:ApplicationVersion>1.0</idb:ApplicationVersion>

   <!-- instructs TSI to run the application interactively -->
   <idb:PreferInteractive>true</idb:PreferInteractive>

   <jsdl:POSIXApplication xmlns:jsdl="http://schemas.ggf.org/jsdl ↩
       /2005/11/jsdl-posix">
     <!-- other args omitted for clarity -->
   </jsdl:POSIXApplication>
  </idb:IDBApplication>
```

Note

This should only be used for very short-running tasks, since UNICORE cannot track the status of such a task. It is simply forked by the TSI, and UNICORE will just assume it is finished after a short while.

7.5 Application metadata (simple)

For client components it is very useful to have a description of an application in terms of its arguments. This allows for example the "Generic" GridBean in the UNICORE Rich client to automatically build a nice GUI for the application.

You can optionally attach metadata to an applications arguments.

```
<jsdl:Argument Description="Verbose Execution"
               Type="boolean"
               ValidValues="true false"
               DependsOn="..."
               Excludes="..."
               IsEnabled="false"
               IsMandatory="false">+v$VERBOSE?</jsdl:Argument>
```

Some metadata is inferred automatically, such as the argument name (`VERBOSE` in the example above).

The meaning of the attributes should be fairly obvious.

- the `Description` attribute contains a human-readable description of the argument
- the `Type` attribute can have the values "string", "boolean", "int", "double", "filename" or "choice". In the case of "choice", the `ValidValues` attribute is used to specify the list of valid values. The type `filename` is used to specify that this is an input file for the application, allowing clients to enable special actions for this.
- The `MimeType` attribute allows to specify the mime-types of an input or output file as a comma-separated list. This can be used by smart clients, for example to check the viability of workflows.
- The `ValidValues` attribute is used to limit the range of valid values, depending on the `Type` of the argument. The processing of this attribute is client-dependent. The UNICORE Rich Client supports intervals for the numeric types, and Java regular expressions for the string types.
- `DependsOn` and `Excludes` are space-separated lists of argument names to control dependencies. For example, a "VERBOSE and a "QUIET" attribute should exclude each other.
- `IsMandatory` (values: true or false) specifies if a parameter MUST be provided.
- `IsEnabled` (values: true or false) is intended to tell clients that the parameter should initially be enabled in the GUI.

7.5.1 Application metadata (complex)

You can also add metadata as XML to the IDB entry, which allows you to add your custom metadata:

The XML schema can be found online at http://unicore.svn.sourceforge.net/viewvc/unicore/-jsdl-xmlbeans/trunk/src/main/schema/jsdl-unicore.xsd

Currently the XML metadata only encompass argument metadata, similar to the "simple" metadata described above. However, custom metadata can be added in case an application requires it.

Here is a simple example.

```
<idb:IDBApplication>
  <idb:ApplicationName>SomeApp</idb:ApplicationName>
  <idb:ApplicationVersion>1.0</idb:ApplicationVersion>

  <!-- metadata -->
  <u6:Metadata xmlns:u6="http://www.unicore.eu/unicore/jsdl- ↩
      extensions">
    <!-- example argument-->
    <u6:Argument>
      <u6:Name>PRECISION</u6:Name>
      <u6:ArgumentMetadata>
        <u6:Type>choice</u6:Type>
        <u6:Description>Precision of the computation</u6: ↩
            Description>
        <u6:ValidValue>Lax</u6:ValidValue>
        <u6:ValidValue>Reasonable</u6:ValidValue>
        <u6:ValidValue>Precise</u6:ValidValue>
        <u6:ValidValue>Pedantic</u6:ValidValue>
        <u6:IsMandatory>true</u6:IsMandatory>
      </u6:ArgumentMetadata>
    </u6:Argument>
    <!-- any custom XML can be added as well -->
    <!-- ... -->
  </u6:Metadata>
 </idb:IDBApplication>
```

The XML supports the Type, Description, MimeType, IsMandatory, DependsOn, Excludes and ValidValue elements, with the same semantics as described above.

7.6 Execution Environments

Execution environments are an advanced feature that allows you to configure the way an executable is executed in a more detailed and user-friendly fashion. A common scenario is the configuration of an environment for parallel execution of a program, such as MPI.

A typical simple MPI invocation looks like this

```
/usr/local/bin/openmpi -np 4 ./my_mpi_program [my_program_arguments ↩
    ]
```

but of course there are many more possible arguments to the MPI command, which also depend on the local installation. By using a predefined execution environment, a UNICORE user need not know all the details, but can set up her job in a simple fashion.

This document covers the options that are available to configure execution environments in the IDB.

- XML Schema for the execution environments: the current XML schema for the execution environment specification can be read from the SVN repository.

7.7 IDB definition of execution environments

The server-side setup of an execution environment is by adding an XML entry into the IDB. A simple environment might be used to run a user command using *time*. This example shows every possible option. You might want to consult the man page of *time*.

```
<idb:IDB xmlns:idb="http://www.fz-juelich.de/unicore/xnjs/idb">

<!-- sample execution environment definition in the IDB -->
  <ee:ExecutionEnvironment xmlns:ee="http://www.unicore.eu/unicore/ ↩
      jsdl-extensions">
    <ee:Name>TIME</ee:Name>
    <ee:Version>1.0</ee:Version>
    <ee:Description>Runs the user's command using the 'time' tool,  ↩
        measuring the used CPU time.</ee:Description>
    <ee:ExecutableName>/usr/bin/time</ee:ExecutableName>
    <ee:CommandlineTemplate>#EXECUTABLE #ARGS #USERCOMMAND # ↩
        USERARGS</ee:CommandlineTemplate>
    <ee:Argument>
      <ee:Name>Output</ee:Name>
      <ee:IncarnatedValue>-o</ee:IncarnatedValue>
      <ee:ArgumentMetadata>
        <ee:Type>string</ee:Type>
        <ee:Description>Write the resource use statistics to a FILE ↩
             instead of to the standard error stream</ee: ↩
            Description>
      </ee:ArgumentMetadata>
    </ee:Argument>
    <ee:Option>
      <ee:Name>Verbose</ee:Name>
      <ee:IncarnatedValue>-v</ee:IncarnatedValue>
      <ee:OptionMetadata>
        <ee:Description>Enable verbose mode</ee:Description>
      </ee:OptionMetadata>
    </ee:Option>
    <ee:PreCommand>
      <ee:Name>PRINT_START_TIME</ee:Name>
      <ee:IncarnatedValue>echo "Started at $(date)"</ee: ↩
          IncarnatedValue>
      <ee:OptionMetadata>
        <ee:Description>Explicitely print the start time</ee: ↩
            Description>
      </ee:OptionMetadata>
    </ee:PreCommand>
    <ee:PostCommand>
      <ee:Name>PRINT_FINISH_TIME</ee:Name>
      <ee:IncarnatedValue>echo "Finished at $(date)"</ee: ↩
          IncarnatedValue>
      <ee:OptionMetadata>
        <ee:Description>Explicitely print the finishing time</ee: ↩
            Description>
      </ee:OptionMetadata>
    </ee:PostCommand>
   </ee:ExecutionEnvironment>

</idb:IDB>
```

If a client now submits a job including a request for the "TIME" execution environment, UNICORE will generate a shell script that wraps the user command in the "time" invocation. Let's say the job request includes the "Output" argument, the "Verbose" option and both precommand and postcommand:

```
  <!-- sample execution environment request sent from client to  ↩
      server -->
  <ee:ExecutionEnvironment xmlns:ee="http://www.unicore.eu/unicore/ ↩
      jsdl-extensions">
    <ee:Name>TIME</ee:Name>
    <ee:Version>1.0</ee:Version>
    <ee:Argument>
      <ee:Name>Output</ee:Name>
      <ee:Value>time_profile</ee:IncarnatedValue>
    </ee:Argument>
    <ee:Option>
      <ee:Name>Verbose</ee:Name>
    </ee:Option>
    <ee:PreCommand>
      <ee:Name>PRINT_START_TIME</ee:Name>
    </ee:PreCommand>
    <ee:PostCommand>
      <ee:Name>PRINT_FINISH_TIME</ee:Name>
     </ee:PostCommand>
   </ee:ExecutionEnvironment>
```

The script generated by UNICORE will look like this (leaving out some standard things):

```
#!/bin/bash

# ...

echo "Started at $(date)"
/usr/bin/time -o time_profile -v /path/to/my_user_application
echo "Finished at $(date)"

# ...
```

In the following the various XML tags that are available are explained in detail.

- `ExecutableName` : This is the name of the executable that "defines" the environment.
- `CommandlineTemplate` : To control the exact commandline that is created, this template is used.

The default template is

```
#EXECUTABLE #ARGS #USERCOMMAND #USERARGS
```

where

- `#EXECUTABLE` is the executable defined using ExecutableName
- `#ARGS` are the arguments and options for the executable
- `#USERCOMMAND` is the user's executable
- `#USERARGS` are the arguments to the user's executable
- `Argument` : the Argument elements are used to create arguments to the executable. They have several subtags.
- `Name` is the name of the argument.
- `IncarnatedValue` is the argument as used in the commandline.
- `ArgumentMetadata` are described below.
- `ArgumentMetadata` : This element allows to describe an Argument in more detail. It has the following subtags
- `Type` the argument type. Valid values are "string", "boolean", "int", "float" and "choice"
- `Description` is a human-friendly description
- `Default` a possible default value
- `ValidValue` tags are used to denote possible values

- `DependsOn` denotes other arguments that this argument requires
- `Excludes` denotes other arguments that clash with this argument
- `PreCommand` : This tag denotes a command that is executed immediately before the actual executable. Its subtags are the same as for `Option`.
- `PostCommand` : This tag denotes a command that is executed after the actual execution. Its subtags are the same as for `PreCommand`.

7.8 Custom resource definitions

Most sites (especially in HPC) have special settings that cannot be mapped to the generic JSDL elements shown in the previous section. Therefore UNICORE 6 includes a mechanism to allow sites to specify their own system settings and allow users to set these using the Grid middleware.

This requires two things

- Custom resource definitions in the IDB
- Customisation of the TSI Submit.pm module

If this mechanism is not flexible enough for your needs, consider looking at dynamic incarnation which is described here Section 7.9.

7.8.1 The IDB

You can insert <Resource> elements into the Resources section, an example follows.

```
<jsdl:Resources>

   <idb:Resource xmlns:idb="http://www.fz-juelich.de/unicore/xnjs/ ↩
       idb">
     <idb:Name>TasksPerNode</idb:Name>
     <idb:Type>int</idb:Type>
     <idb:Description>The number of tasks per node. If larger than ↩
          32, the node will run in SMT mode.</idb:Description>
     <idb:Min>1</idb:Min>
     <idb:Max>64</idb:Max>
     <idb:Default>32</idb:Default>
   </idb:Resource>

 </jsdl:Resources>
```

Apart from the numeric types <int> or <double>, there are the <string>, <choice> and <boolean> types. The <choice> allows you to specify a set of allowed values. This is useful for example to specify a selection of batch queues, or a selection of network topologies.

For example, defining queues could look like this:

```
<jsdl:Resources>

   <idb:Resource xmlns:idb="http://www.fz-juelich.de/unicore/xnjs/ ↩
       idb">
     <idb:Name>Queue</idb:Name>
     <idb:Type>choice</idb:Type>
     <idb:Description>The batch queue to use</idb:Description>
     <idb:Default>normal</idb:Default>
     <idb:AllowedValue>normal</idb:AllowedValue>
     <idb:AllowedValue>fast</idb:AllowedValue>
     <idb:AllowedValue>small</idb:AllowedValue>
     <idb:AllowedValue>development</idb:AllowedValue>
   </idb:ResourceSetting>

 </jsdl:Resources>
```

This example defines four available queues, with the "normal" one being used by default.

Note
The resource name "Queue" is recognized automatically by UNICORE and mapped to the correct TSI_QUEUE parameter when sending the job to the TSI.

7.8.2 Submitted JSDL

Clients can now send a special element in the JSDL job, for example requesting a certain value for the "TasksPerNode" setting:

```
   <jsdl:JobDescription>
 ...
       <jsdl:Resources>

         <jsdl-u:ResourceRequest xmlns:jsdl-u="http://www.unicore. ↩
             eu/unicore/jsdl-extensions">
           <jsdl-u:Name>TasksPerNode</jsdl-u:Name>
           <jsdl-u:Value>64</jsdl-u:Value>
         </jsdl-u:ResourceRequest>

       </jsdl:Resources>
     </jsdl:JobDescription>
```

or for the queue example:

```
   <jsdl:JobDescription>
 ...
       <jsdl:Resources>
```

```
     <jsdl-u:ResourceRequest xmlns:jsdl-u="http://www.unicore. ↩
       eu/unicore/jsdl-extensions">
      <jsdl-u:Name>Queue</jsdl-u:Name>
      <jsdl-u:Value>development</jsdl-u:Value>
     </jsdl-u:ResourceRequest>

    </jsdl:Resources>
   </jsdl:JobDescription>
```

7.8.3 TSI request

The UNICORE/X server will send the following snippet to the TSI:

```
#!/bin/sh
#TSI_SUBMIT
# ...
#TSI_SSR_TASKSPERNODE 64.0
# ....
```

As you can see, a special TSI command tag "#TSI_SSR_TASKSPERNODE" has been added. Now the remaining step is to have the TSI Submit.pm module has to parse this properly, and generate the correct batch system command.

7.9 Tweaking the incarnation process

In UNICORE the term incarnation refers to the process of changing the abstract and probably universal *grid request* into a sequence of operations *local to the target system*. The most fundamental part of this process is creation of the execution script which is invoked on the target system (usually via a batch queuing subsystem (BSS)) along with an execution context which includes local user id, group, BSS specific resource limits.

UNICORE provides a flexible incarnation model - most of the magic is done automatically by TSI scripts basing on configuration which is read from the IDB. IDB covers script creation (using templates, abstract application names etc). Mapping of the grid user to the local user is done by using UNICORE Attribute Sources like XUUDB or UVOS.

In rare cases the standard UNICORE incarnation mechanism is not flexible enough. Typically this happens when the script which is sent to TSI should be tweaked in accordance to some runtime constraints. Few examples may include:

- Administrator wants to set memory requirements for all invocations of the application X to 500MB if user requested lower amount of memory (as the administrator knows that the application consumes always at least this amount of memory).
- Administrator wants to perform custom logging of suspected requests (which for instance exceed certain resource requirements threshold)

- Administrator need to invoke a script that create a local user's account if it doesn't exist.
- Administrator wants to reroute some requests to a specific BSS queue basing on the arbitrary contents of the request.
- Administrator wants to set certain flags in the script which is sent to TSI when a request came from the member of a specific VO. Later those flags are consumed by TSI and are used as submission parameters.

Those and all similar actions can be performed with the Incarnation tweaking subsystem. Note that though it is an extremely powerful mechanism, it is also a very complicated one and configuring it is error prone. Therefore always try to use the standard UNICORE features (like configuration of IDB and attribute sources) in the first place. Treat this incarnation tweaking subsystem as the last resort!

To properly configure this mechanism at least a very basic Java programming language familiarity is required. Also remember that in case of any problems contacting the UNICORE support mailing list can be the solution.

7.9.1 Operation

It is possible to influence incarnation in two ways:

- *BEFORE-SCRIPT* it is possible to change all UNICORE variables which are used to produce the final TSI script just *before it is created* and
- *AFTER-SCRIPT* later on to *change the whole TSI script*.

The first BEFORE-SCRIPT option is suggested: it is much easier as you have to modify some properties only. In the latter much more error prone version you can produce an entirely new script or just change few lines of the script which was created automatically. It is also possible to use both solutions simultaneously.

Both approaches are configured in a very similar way by defining rules. Each rule has its condition which triggers it and list of actions which are invoked if the condition was evaluated to true. The condition is in both cases expressed in the same way. The difference is in case of actions. Actions for BEFORE-SCRIPT rules can modify the incarnation variables but do not return a value. Actions for AFTER-SCRIPT read as its input the original TSI script and must write out the updated version. Theoretically AFTER-SCRIPT actions can also modify the incarnation variables but this doesn't make sense as those variables won't be used.

7.9.2 Basic configuration

By default the subsystem is turned off. To enable it you must perform two simple things:

- Add the `XNJS.incarnationTweakerConfig` property to the XNJS config file. The value of the property must provide a location of the file with dynamic incarnation rules.

- Add some rules to the file configured above.

The following example shows how to set the configuration file to the value `conf/incarnationTweaker.xml`:

```
  ...
  <eng:Properties>
    ...
    <eng:Property name="XNJS.incarnationTweakerConfig" value="conf/ ↩
        incarnationTweaker.xml"/>
    ...
  </eng:Properties>
  ...
```

The contents of the rules configuration file must be created following this syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:incarnationTweaker xmlns:tns="http://eu.unicore/xnjs/ ↩
    incarnationTweaker"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

        <tns:beforeScript>
                <!-- Here come BEFORE-SCRIPT rules-->
        </tns:beforeScript>

        <tns:afterScript>
                <!-- And here AFTER-SCRIPT rules-->
        </tns:afterScript>
</tns:incarnationTweaker>
```

7.9.3 Creating rules

Each rule must conform to the following syntax:

```
        <tns:rule finishOnHit="false">
                <tns:condition> <!-- Here comes the rule’s  ↩
                    condition --> </tns:condition>

                <tns:action type="ACTION-TYPE">ACTION-DEFINITION</ ↩
                    tns:action>
                <!-- More actions may follow -->
        </tns:rule>
```

The rule’s attribute `finishOnHit` is optional, by default its value is false. When it is present and set to true then this rule becomes the last rule invoked if it’s condition was met.

You can use as many actions as you want (assuming that at least one is present), actions are invoked in the order of appearance.

SpEL and Groovy

Rule conditions are always boolean expressions of the Spring Expression Language (SpEL). As SpEL can be also used in some types of actions it is the most fundamental tool to understand.

Full documentation is available here: http://static.springsource.org/spring/docs/3.0.0.M3/spring-framework-reference/html/ch07.html

The most useful is the section 7.5: http://static.springsource.org/spring/docs/3.0.0.M3/spring-framework-reference/html/ch07s05.html

Actions can be also coded using the Groovy language. You can find Groovy documentation at Groovy's web page: http://groovy.codehaus.org

Creating conditions

Rule conditions are always Spring Expression Language (SpEL) boolean expressions. To create SpEL expressions, the access to the request-related variables must be provided. All variables which are available for conditions are explained in the dynamic incarnation context Section 7.10 section.

Creating BEFORE-SCRIPT actions

There are the following action types which you can use:

- `spel` (the default which is used when type parameter is not specified) treats action value as SpEL expression which is simply evaluated. This is useful for simple actions that should modify value of one variable.
- `script` treats action value as a SpEL expression which is evaluated and which should return a string. Evaluation is done using SpEL templating feature with `\${` and `}` used as variable delimiters (see section 7.5.13 in Spring documentation for details). The returned string is used as a command line which is invoked. This action is extremely useful if you want to run an external program with some arguments which are determined at runtime. Note that if you want to cite some values that may contain spaces (to treat them as a single program argument) you can put them between double quotes `"`. Also escaping characters with "\" works.
- `groovy` treats action value as a Groovy script. The script is simply invoked and can manipulate the variables.
- `groovy-file` works similarly to the `groovy` action but the Groovy script is read from the file given as the action value.

All actions have access to the same variables as conditions; see the dynamic incarnation context Section 7.10 section for details.

Creating AFTER-SCRIPT actions

There are the following action types which you can use:

- `script` (the default which is used when type parameter is not specified) treats action value as SpEL expression which is evaluated and which should return a string. Evaluation is done using SpEL templating feature with `\${` and `}` used as variable delimiters (see section 7.5.13 in Spring documentation for details). The returned string used as a command line which is invoked. The invoked application gets as its standard input the automatically created TSI script and is supposed to return (using standard output) the updated script which shall be used instead. This action is extremely useful if you want to run an external program with some arguments which are determined at runtime. Note that if you want to cite some values that may contain spaces (to treat them as a single program argument) you can put them between double quotes `"`. Also escaping characters with \ works.
- `groovy` treats action value as a Groovy script. The script has access to one special variable `input` of type `Reader`. The original TSI script is available from this reader. The groovy script is expected to print the updated TSI script which shall be used instead of the original one.
- `groovy-file` works the same as the `groovy` action but the Groovy script is read from the file given as the action value.

All actions have access to the same variables as conditions; see the section on dynamic incarnation context Section 7.9 for details.

7.9.4 Final notes

- The rules configuration file is automatically reread at runtime.
- If errors are detected in the rules configuration file upon server startup then the whole subsystem is disabled. If errors are detected at runtime after an update then old version of rules is continued to be used. Always check the log file!
- When rules are read the system tries to perform a dry run using an absolutely minimal execution context. This can detect some problems in your rules but mostly only in conditions. Actions connected to conditions which are not met won't be invoked. Always try to submit a real request to trigger your new rules!
- Be careful when writing conditions: it is possible to change incarnation variables inside your condition - such changes also influence incarnation.

7.9.5 Complete examples and hints

Invoking a logging script for users who have the `specialOne` role. Note that the script is invoked with two arguments (role name and client's DN). As the latter argument may contain spaces we surround it with quotation marks.

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:incarnationTweaker xmlns:tns="http://eu.unicore/xnjs/ ↩
    incarnationTweaker"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

        <tns:beforeScript>
                <tns:rule>
                        <tns:condition>client.role.name == " ↩
                            specialOne"</tns:condition>
                        <tns:action type="script">/opt/scripts/ ↩
                            logSpecials.sh ${client.role.name} "${ ↩
                            client.distinguishedName}"</tns:action>
                </tns:rule>
        </tns:beforeScript>

        <tns:afterScript>
        </tns:afterScript>
</tns:incarnationTweaker>
```

A more complex example. Let’s implement the following rules:

- The Application with a IDB name `HEAVY-APP` will always get 500MB of memory requirement if user requested less or nothing.
- All invocations of an executable */usr/bin/serial-app* are made serial, i.e. the number of requested nodes and CPUs are set to 1.
- For all requests a special script is called which can create a local account if needed along with appropriate groups.
- There is also one AFTER-RULE. It invokes a groovy script which adds an additional line to the TSI script just after the first line. The line is added for all invocations of the */usr/bin/serial-app* program.

The realization of the above logic can be written as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:incarnationTweaker xmlns:tns="http://eu.unicore/xnjs/ ↩
    incarnationTweaker"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

        <tns:beforeScript>
                <tns:rule>
                        <tns:condition>app.applicationName == " ↩
                            HEAVY-APP" and (resources. ↩
                            individualPhysicalMemory == null
                                          or resources. ↩
                                             individualPhysicalMemory ↩
                                              &lt; 500000000)</tns ↩
                                             :condition>
                        <tns:action>resources. ↩
                            individualPhysicalMemory=500000000</tns ↩
                            :action>
                </tns:rule>
                <tns:rule>
                        <tns:condition>app.executable == "/usr/bin/ ↩
                            serial-app" and resources. ↩
                            individualCPUCount != null</tns: ↩
                            condition>
                        <tns:action>resources.individualCPUCount ↩
                            =1</tns:action>
                        <tns:action>resources.totalResourceCount ↩
                            =1</tns:action>
                </tns:rule>
                <tns:rule>
                        <tns:condition>true</tns:condition>
                        <tns:action type="script">/opt/ ↩
                            addUserIfNotExists.sh ${client.xlogin. ↩
                            userName} ${client.xlogin.encodedGroups ↩
                            }</tns:action>
                </tns:rule>
        </tns:beforeScript>

        <tns:afterScript>
                <tns:rule>
                        <tns:condition>app.executable == "/usr/bin/ ↩
                            serial-app"</tns:condition>
                        <tns:action type="groovy">
int i=0;
input.eachLine() { line ->
if(i==1) {
     println("#TSI_MYFLAG=SERIAL");
     println(line);
} else
     println(line);

i++;
}
                        </tns:action>
                </tns:rule>
        </tns:afterScript>
</tns:incarnationTweaker>
```

Remember that some characters are special in XML (e.g. < and `&`). You have to encode them with XML entities (e.g. as `&lt;` and `&gt;` respectively) or put the whole text in a CDATA section. A CDATA section starts with "<![CDATA[" and ends with "]]>". Example:

```
<tns:condition><!CDATA[ resources.individualPhysicalMemory <  ↩
    500000000 ]]></tns:condition>
```

Note that usually it is better to put Groovy scripts in a separate file. Assuming that you placed the contents of the groovy AFTER-action above in a file called */opt/scripts/filter1.g* then the following AFTER-SCRIPT section is equivalent to the above one:

```
        <tns:afterScript>
                <tns:rule>
                        <tns:condition>app.executable == "/usr/bin/ ↩
                            serial-app"</tns:condition>
                        <tns:action type="groovy-file">/opt/scripts ↩
                            /filter1.g</tns:action>
                </tns:rule>
        </tns:afterScript>
```

To check your rules when you develop them, it might be wise to enable DEBUG logging on incarnation tweaker facility. To do so add the following setting to the `logging.properties` file:

```
log4j.logger.unicore.xnjs.IncarnationTweaker=DEBUG
```

You may also want to see how the final TSI script looks like. Most often TSI places it in a file in job's directory. However if the TSI you use doesn't do so (e.g. in case of the NOBATCH TSI) you can trigger logging of the TSI script on the XNJS side. There are two ways to do it. You can enable DEBUG logging on the `unicore.xnjs.tsi.TSIConnection` facility:

```
log4j.logger.unicore.xnjs.tsi.TSIConnection=DEBUG
```

This solution is easy but it will produce also much more of additional information in you log file. If you want to log TSI scripts only, you can use AFTER-SCRIPT rule as follows:

```
        <tns:afterScript>
                <tns:rule>
                        <tns:condition>true</tns:condition>
                        <tns:action type="groovy">
org.apache.log4j.Logger log=org.apache.log4j.Logger.getLogger(" ↩
    unicore.xnjs.RequestLogging");
log.info("Dumping TSI request:");
input.eachLine() { line ->
     println(line);
     log.info("  " + line);
}
                        </tns:action>
                </tns:rule>
        </tns:afterScript>
```

The above rule logs all requests to the normal Unicore/X log file with the INFO level.

7.10 Incarnation tweaking context

Dynamic incarnation tweaker conditions and also all actions are provided with access to all relevant data structures which are available at XNJS during incarnation.

The following variables are present:

- `Client client` provides access to authorization material: xlogin, roles, attributes etc. NOTE: In general it makes sense to modify only the xlogin field in the Client object, the rest are available only for information purposes. E.g. there is a `queue` field, but changing it in the incarnation tweaker rules will have no effect on incarnation. Use the `queue` property available from `resources` variable instead. You can read client's queue to check what queue settings were defined in attribute sources for the user. The source
- `ApplicationInfo app` provides access to information about application to be executed (both abstract IDB name and actual target system executable). You can change the values here to influence the incarnation. Remember that changing the user's DN here won't influence authorization layer as authorization was already done for each request at this stage. The source
- `ResourcesWrapper resources` provides access to resource requirements of the application. The source
- `ExecutionContext ec` provides access to the application environment: interactive setting, environment variables, working directory and stdin/out/err files. The source
- `IncarnatedExecutionEnvironment execEnv` provides access to the template which is used to produce the final script. In most cases only manipulating pre- and post- commands makes sense. The source
- `IncarnationDataBase idb` provides an (read only) access to the contents of the IDB. The source

Each of the available variables has many properties that you can access. It is best to check source code of the class to get a complete list of them. You can read property X if it has a corresponding Java `public Type getX()` method. You can set a property Y if it has a corresponding Java `public void setY(Type value)` method.

7.10.1 Simple example

Let's consider the variable `client`. In the `Client` class you can find methods:

```
public String getDistinguishedName()

public void setDistinguishedName(String distinguishedName)
```

This means that the following SpEL condition is correct:

```
client.distinguishedName != null and client.distinguishedName == " ↩
    CN=Roger Zelazny,C=US"
```

Note that it is always a safe bet to check first if the value of a property is not null.

Moreover you can also set the value of the distinguished name in an action (this example is correct for both SpEL and Groovy):

```
client.distinguishedName="CN=Roger Zelazny,C=US"
```

7.10.2 Advanced example

Often the interesting property is not available directly under one of the above enumerated variables. In case of the `client` variable one example may be the `xlogin` property holding the list of available local accounts and groups and the ones which were selected among them.

Example of condition checking the local user id:

```
client.xlogin.userName != null and client.xlogin.userName == "roger ↩
    "
```

Setting can also be done in an analogous way. However always pay attention to the fact that not always setting a value will succeed. E.g. for Xlogin it is possible to set a selected xlogin only to one of those defined as available (see contents if the respective `setSelectedLogin()` method). Therefore to change local login to a fixed value it is best to just override the whole XLogin object like this (SpEL):

```
client.xlogin=new eu.unicore.security.Xlogin(new String[] {"roger ↩
    "}, new String{"users"})
```

7.10.3 Resources variable

As it is bit difficult to manipulate the resources requirements object which is natively used by UNICORE, it is wrapped to provide an easier to use interface. The only exposed properties are those requirements which are actually used by UNICORE when the TSI script is created.

You can access the low level (and complicated) original resources object through the `resources.allResources` property.

8 Authorization back-end (PDP) guide

The authorization process in UNICORE/X requires that nearly all operations must be authorized prior to execution (exceptions may be safely ignored).

UNICORE allows to choose which authorization back-end is used. The module which is responsible for this operation is called Policy Decision Point (PDP). You can choose one among already available PDP modules or even develop your own engine.

Local PDPs use a set of policy files to reach an authorisation decision, remote PDPs query a remote service.

Local UNICORE PDPs use the XACML language to express the authorization policy. The XACML policy language is introduced in the Guide to XACML security policies Section 9. You can also review this guide if you want to have a deeper understanding of the authorization process.

8.1 Basic configuration

There are three options in the *uas.config* file which are relevant to all PDPs:

- `uas.security.accesscontrol` (values: `true` or `false`) This boolean property can be used to completely turn off the authorization. This guide makes sense only if this option is set to `true`. Except for test scenarios this should never be switched off, otherwise every user can in principle access all resources on the server.
- `uas.security.accesscontrol.pdp` (value: full class name) This property is used to choose which PDP module is being used.
- `uas.security.accesscontrol.pdp.config` (value: file path) This property provides a location of a configuration file of the selected PDP.

8.2 Available PDP modules

8.2.1 XACML 2.0 PDP

The implementation class of this module is: `eu.unicore.uas.pdp.local.LocalHerasafPDP` so to enable this module use the following configuration in uas.config:

```
uas.security.accesscontrol.pdp.config=<CONFIG_DIR>/xacml2.conf
uas.security.accesscontrol.pdp=eu.unicore.uas.pdp.local. ↩
    LocalHerasafPDP
```

The configuration file content is very simplistic as it is enough to define only few options:

```
# The directory where XACML 2.0 policy files are stored
localpdp.directory=conf/xacml2Policies

# Wildcard expression to select actual policy files from the  ↩
    directory defined above
localpdp.filesWildcard=*.xml

# Combining algorithm for the policies. You can use the full XACML  ↩
    id or its last part.
localpdp.combiningAlg=first-applicable
```

The policies from the `localpdp.directory` are always evaluated in alphabetical order, so it is good to name files with a number. By default the first-applicable combining algorithm is used and UNICORE policy is stored in two files: *01coreServices.xml* and *99finalDeny.xml*. The first file contains the default access policy, the latter a single fall through deny rule. Therefore you can put your own policies using an additional file in file named e.g. *50localRules.xml*.

The policies are reloaded whenever you change (or touch) the configuration file of this PDP, e.g. like this:

```
touch conf/xacml2.conf
```

8.2.2 XACML 1.x PDP

The implementation class of this module is: `eu.unicore.uas.pdp.localsun.LocalSunPDP` so to enable this module use the following configuration in uas.config:

```
uas.security.accesscontrol.pdp.config=conf/xacml.config
uas.security.accesscontrol.pdp=eu.unicore.uas.pdp.localsun. ↩
    LocalSunPDP
```

This module is the one that was the only available option in UNICORE prior to release 6.4.0

The rules are contained in one or more policy files as listed in the *xacml.config* configuration file. However note that in case of this legacy implementation it mostly doesn't make sense to use more then one file as it not possible to control the combining algorithm (which would be only-one-applicable). Therefore the configuration file is rather absolutely constant:

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://sunxacml.sourceforge.net/schema/config-0.3"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        defaultPDP="pdp" defaultAttributeFactory="attr"
        defaultCombiningAlgFactory="comb" defaultFunctionFactory=" ↩
            func">
  <pdp name="pdp">
    <attributeFinderModule class="com.sun.xacml.finder.impl. ↩
        CurrentEnvModule"/>
    <attributeFinderModule class="com.sun.xacml.finder.impl. ↩
        SelectorModule"/>
    <policyFinderModule class="com.sun.xacml.finder.impl. ↩
        FilePolicyModule">
      <list>
        <string>conf/security_policy.xml</string>
      </list>
    </policyFinderModule>
  </pdp>
  <attributeFactory name="attr" useStandardDatatypes="true"/>
  <combiningAlgFactory name="comb" useStandardAlgorithms="true"/>
  <functionFactory name="func" useStandardFunctions="true">

  </functionFactory>
</config>
```

In case you modified the policy file(s), you can force a reload into the running server by "touch"ing the xacml.conf configuration file. For example, under Unix you can execute

```
touch conf/xacml.conf
```

Opening the file in an editor and saving it will also do the trick.

8.2.3 Remote SAML/XACML 2.0 PDP (Argus PDP)

This PDP allows for outsourcing authorization decision to the remote PDP service. Typically the Argus PDP is used for this purpose but in principle any PDP which implements the SAML XAML Authorization Query Protocol can be used.

The implementation class of this module is: `eu.unicore.uas.pdp.argus.ArgusPDP` so to enable this module use the following configuration in uas.config:

```
uas.security.accesscontrol.pdp.config=<CONFIG_DIR>/argus.config
uas.security.accesscontrol.pdp=eu.unicore.uas.pdp.argus.ArgusPDP
```

The PDP configuration is very simple as it is only required to provide the Argus endpoint and query timeout (in milliseconds).

```
arguspdp.serverAddress=https://argus.exampledomain.org:8152/authz
arguspdp.queryTimeout=15000
```

You can use both http and https addresses. In the latter case server's certificate is used to make the connection.

If the remote PDP can not be contacted due to any reason the authorization decision is always deny.

9 Guide to XACML security policies

XACML authorization policies need not to be modified on a day-to-day basis when running the UNICORE server. The most common tasks as banning or allowing users can be performed very easily using UNICORE Attribute Sources like XUUDB or UVOS. This guide is intended for advanced administrators who want to change the non-standard authorization process and for developers who want to provide authorization policies for services they create.

The XACML standard is a powerful way to express fine grained access control. The idea is to have XML policies describing how and by whom actions on resources can be performed. A very readable introduction into XACML can be found with Sun's XACML implementation.

There are several versions of XACML policy language. Currently UNICORE supports both 1.x and 2.0 versions. Those are quite similar and use same concepts, however note that syntax is a bit different. In this guide we provide examples using XACML 2.0. The same examples in the legacy XACML 1.1 format are available below Section 9.5.

UNICORE allows to choose one of several authorization back-end implementations called Policy Decision Points (PDP). Among others you can decide whether to use local XACML 1.x policies or local XACML 2.0 policies. The authorization section Section 8 shows how to choose and configure each of the available PDPs.

In UNICORE terms XACML is used as follows. Before each operation (i.e. execution of a web service call), an XACML request is generated, which currently includes the following attributes:

- the name of the service being accessed (e.g. JobManagementService)
- the name of the method being invoked (used as the "action" in XACML terms)
- the distinguished name of the user making the request
- the role of the user as retrieved from an attribute source (XUUDB/UVOS/. . .)

The request is processed by the server and checked against a (set of) policies. Policies contain rules that can either deny or permit a request, using a powerful set of functions.

9.1 Policy sets and combining of results

Typically, the authorization policy is stored in one file. However as this file can get long and unmanageable sometimes it is better to split it into several ones. This additionally allows to easily plug additional policies to the existing authorization process. In UNICORE, this feature is implemented in the XAML 2.0 PDP.

When policies are split in multiple files each of those files must contain (at least one) a separate policy. A PDP must somehow combine result of evaluation of multiple policies. This is done by so-called policy combining algorithm. The following algorithms are available, the part after last colon describes behaviour of each:

```
urn:oasis:names:tc:xacml:1.1:policy-combining-algorithm:ordered- ↩
    permit-overrides
urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit- ↩
    overrides
urn:oasis:names:tc:xacml:1.1:policy-combining-algorithm:ordered- ↩
    deny-overrides
urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:deny- ↩
    overrides
urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:first- ↩
    applicable
urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:only-one- ↩
    applicable
```

Each policy file can contain one or more rules, so it is important to understand how possible conflicts are resolved. The so-called combining algorithm for the rules in a single policy file is specified in the top-level Policy element.

The XACML (from version 1.1 onwards) specification defines six algorithms: permit-overrides, deny-overrides, first-applicable, only-one-applicable, ordered-permit-overrides and ordered-deny-overrides. For example, to specify that the first matching rule in the policy file is used to make the decision, the Policy element must contain the following "RuleCombiningAlgId" attribute:

```
<Policy xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        PolicyId="ExamplePolicy"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule- ↩
            combining-algorithm:first-applicable">
```

The full identifiers of the combining algorithms are as follows:

```
urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:deny- ↩
    overrides
urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit- ↩
    overrides
urn:oasis:names:tc:xacml:1.1:rule-combining-algorithm:ordered-deny- ↩
    overrides
urn:oasis:names:tc:xacml:1.1:rule-combining-algorithm:ordered- ↩
    permit-overrides
urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:first- ↩
    applicable
```

9.2 Role-based access to services

A common use case is to allow/permit access to a certain service based on a user's role This can be achieved with the following XACML rule, which describes that a user with role "admin" is given access to all service.

```
<Rule RuleId="Permit:Admin" Effect="Permit">
        <Description> Role "admin" may do anything. </Description>
        <Target />
        <Condition>
          <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function: ↩
              string-equal">
            <Apply FunctionId="urn:oasis:names:tc:xacml:1.0: ↩
                function:string-one-and-only">
                <SubjectAttributeDesignator
                    DataType="http://www.w3.org/2001/XMLSchema# ↩
                        string" AttributeId="role" />
            </Apply>
            <AttributeValue DataType="http://www.w3.org/2001/ ↩
                XMLSchema#string">admin</AttributeValue>
          </Apply>
        </Condition>
</Rule>
```

If the access should be limited to a certain service, the `Target` element must contain a service identifier, as follows. In this example, access to the *DataService* is granted to those who have the *data-access* role.

```
<Rule RuleId="rule2" Effect="Permit">
        <Description>Allow users with role "data-access" access to  ↩
            the DataService</Description>
        <Target>
          <Resources>
            <Resource>
              <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0: ↩
                  function:anyURI-equal">
```

```
                <AttributeValue DataType="http://www.w3.org/2001/ ↩
                    XMLSchema#anyURI">DataService</AttributeValue>
                <ResourceAttributeDesignator AttributeId="urn:oasis ↩
                    :names:tc:xacml:1.0:resource:resource-id"
                                             DataType="http://www. ↩
                                                 w3.org/2001/ ↩
                                                 XMLSchema#anyURI"  ↩
                                                 MustBePresent=" ↩
                                                 true" />
              </ResourceMatch>
            </Resource>
          </Resources>
        </Target>

        <Condition>
          <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function: ↩
              string-equal">
            <Apply FunctionId="urn:oasis:names:tc:xacml:1.0: ↩
                function:string-one-and-only">
              <SubjectAttributeDesignator DataType="http://www.w3. ↩
                  org/2001/XMLSchema#string" AttributeId="role" />
            </Apply>
           <AttributeValue DataType="http://www.w3.org/2001/ ↩
               XMLSchema#string">data-access</AttributeValue>
          </Apply>
        </Condition>
```

By using the <Action> tag in policies, web service access can be controlled on the method level. In principle, XACML supports even control based on the content of some XML document, such as the incoming SOAP request. However this is not yet used in UNICORE/X.

9.3 Limiting access to services to the service instance owner

Most service instances (corresponding e.g. to jobs, or files) should only ever be accessed by their owner. This rule is expressed as follows

```
<Rule RuleId="Permit:AnyResource_for_its_owner" Effect="Permit">
        <Description> Access to any resource is granted for its  ↩
            owner </Description>
        <Target />
        <Condition>
          <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function: ↩
              x500Name-equal">
            <Apply FunctionId="urn:oasis:names:tc:xacml:1.0: ↩
                function:x500Name-one-and-only">
              <SubjectAttributeDesignator AttributeId="urn:oasis: ↩
                  names:tc:xacml:1.0:subject:subject-id"
```

```
                                          DataType="urn:oasis:names ↩
                                              :tc:xacml:1.0:data- ↩
                                              type:x500Name"
                                          MustBePresent="true" />
            </Apply>
            <Apply FunctionId="urn:oasis:names:tc:xacml:1.0: ↩
                function:x500Name-one-and-only">
              <ResourceAttributeDesignator
                AttributeId="owner" DataType="urn:oasis:names:tc: ↩
                    xacml:1.0:data-type:x500Name"
                MustBePresent="true" />
            </Apply>
          </Apply>
        </Condition>
</Rule>
```

9.4 More details on XACML use in UNICORE/X

To get more detailed information about XACML policies (e.g. to get the list of all available functions etc) consult the XACML specification. To get more information on XACML use in UNICORE/X it is good to set the logging level of security messages to DEBUG:

```
log4j.logger.unicore.security=DEBUG
```

You will be able to read what input is given to the XACML engine and what is the detailed answer. Alternatively, ask on the support mailing list.

9.5 Policy examples in XACML 1.1 syntax

This section contains the same examples as are contained in the previous section, but using XACML 1.x syntax. For more detailed discussion of each example please refer to the previous section.

Policy header with first-applicable combining algorithm.

```
<Policy xmlns="urn:oasis:names:tc:xacml:1.0:policy"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        PolicyId="ExamplePolicy"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule- ↩
            combining-algorithm:first-applicable">
```

A user with role "admin" is given access to all service.

```
<Rule RuleId="rule1" Effect="Permit">
 <Description>Allow users with role "admin" access to any service</ ↩
     Description>
    <Target>
```

```
      <Subjects>
        <AnySubject/>
      </Subjects>
      <Resources>
        <AnyResource/>
      </Resources>
      <Actions>
        <AnyAction/>
      </Actions>
    </Target>
 <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function: ↩
     string-equal">
   <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string- ↩
       one-and-only">
     <SubjectAttributeDesignator DataType="http://www.w3.org/2001/ ↩
         XMLSchema#string" AttributeId="role" />
   </Apply>
   <!-- here is the role value -->
   <AttributeValue DataType="http://www.w3.org/2001/XMLSchema# ↩
       string">admin</AttributeValue>
 </Condition>
 /Rule>
```

Defining which resource access is defined with the `Target` element:

```
<Rule RuleId="rule2" Effect="Permit">
 <Description>Allow users with role "data-access" access to the  ↩
     DataService</Description>
    <Target>
      <Subjects>
        <AnySubject/>
      </Subjects>
      <Resources>
       <!-- specify the data service -->
       <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0: ↩
            function:anyURI-equal">
          <AttributeValue DataType="http://www.w3.org/2001/ ↩
              XMLSchema#anyURI">DataService</AttributeValue>
          <ResourceAttributeDesignator DataType="http://www.w3.org ↩
              /2001/XMLSchema#anyURI"
                                       AttributeId="urn:oasis:names ↩
                                           :tc:xacml:1.0:resource: ↩
                                           resource-id"/>
        </ResourceMatch>
      </Resource>
      </Resources>
      <Actions>
        <AnyAction/>
      </Actions>
    </Target>
 <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function: ↩
     string-equal">
   <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string- ↩
       one-and-only">
     <SubjectAttributeDesignator DataType="http://www.w3.org/2001/ ↩
         XMLSchema#string" AttributeId="role" />
   </Apply>
   <!-- here is the role value -->
   <AttributeValue DataType="http://www.w3.org/2001/XMLSchema# ↩
       string">data-access</AttributeValue>
 </Condition>
 /Rule>
```

Allowing access for the resource owner:

```
<Rule RuleId="PermitJobManagementServiceForOwner" Effect="Permit">
  <Description>testing</Description>
  <Target>
    <Subjects> <AnySubject/> </Subjects>
    <Resources>
      <Resource>
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0: ↩
            function:anyURI-equal">
          <AttributeValue DataType="http://www.w3.org/2001/ ↩
              XMLSchema#anyURI">JobManagementService</ ↩
              AttributeValue>
          <ResourceAttributeDesignator AttributeId="urn:oasis:names ↩
              :tc:xacml:1.0:resource:resource-id" DataType="http:// ↩
              www.w3.org/2001/XMLSchema#anyURI" MustBePresent="true ↩
              "/>
        </ResourceMatch>
      </Resource>
    </Resources>
    <Actions> <AnyAction/> </Actions>
  </Target>
  <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function: ↩
      x500Name-equal">
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function: ↩
        x500Name-one-and-only">
      <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc: ↩
          xacml:1.0:subject:subject-id" DataType="urn:oasis:names: ↩
          tc:xacml:1.0:data-type:x500Name" MustBePresent="true"/>
    </Apply>
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function: ↩
        x500Name-one-and-only">
      <ResourceAttributeDesignator AttributeId="owner" DataType=" ↩
          urn:oasis:names:tc:xacml:1.0:data-type:x500Name"  ↩
          MustBePresent="true"/>
    </Apply>
  </Condition>
</Rule>
```

10 Proxy certificate support

Note

First, a warning: proxies are not really supported in UNICORE, except for a very limited set of usage scenarios. Many "normal" things will not work with proxy certificates. Thus, only use this feature if really strictly necessary. No feature in UNICORE *requires* proxies

Proxies are supported in two ways in UNICORE

- transport-layer security and authentication via the UNICORE gateway
- enable usage of GSI based software such as GridFTP

This document provides information and configuration snippets for the second usage scenario. Information about the first case can be found on the SourceForge Wiki page EnableProxySupport.

10.1 TLS proxy support

Using proxies for TLS means that the proxy certificate is used by the client to establish the SSL connection. You must use a gateway with the appropriate configuration for this to work. On the UNICORE/X side it is necessary to set a property in uas.config :

```
uas.authoriser.proxysupport=true
```

10.2 GSI tools support

Your UNICORE client needs to create and send the proxy. Both UCC and URC support this, please consult your client documentation for the details.

10.2.1 Storing the proxy in the job directory

First, you need to enable a handler on the web services engine. In the unicorex/conf/wsrflite.xml, add a handler definition on the target system service:

```
<service name="TargetSystemService" wsrf="true" persistent="true ↩
    ">
 ...
 <!-- additional proxy extraction handler definition -->
 <handler type="in" class="de.fzj.unicore.uas.security. ↩
     ProxyCertInHandler"/>
</service>
```

The handler can also be added for all services like this:

```
<!-- add proxy extract handler on all services.
     This needs to be done *before* the service definitions -->
<globalHandler type="in" class="de.fzj.unicore.uas.security. ↩
    ProxyCertInHandler"/>

<service name="...">
</service>

...
```

Secondly, you need to modify the XNJS configuration to enable a component that stores the proxy in the format expected by GSI (no encryption, PEM format).

So open the XNJS config file (e.g. *conf/xnjs.xml*) and edit the ProcessingChain section.

```
  <eng:ProcessingChain actionType="JSDL" jobDescriptionType="{ ↩
      http://schemas.ggf.org/jsdl/2005/11/jsdl}JobDefinition">
  <!-- stores proxy to uspace -->
  <eng:Processor>de.fzj.unicore.uas.xnjs. ↩
      ProxyCertToUspaceProcessor</eng:Processor>
  <!-- usual entries -->
  <eng:Processor>de.fzj.unicore.xnjs.jsdl.JSDLProcessor</eng: ↩
      Processor>
  <eng:Processor>de.fzj.unicore.xnjs.ems.processors.UsageLogger</ ↩
      eng:Processor>
</eng:ProcessingChain>
```

10.2.2 Configuring gridftp

Using GridFTP basically works out of the box, if the client sends a proxy and you have Globus installed on your TSI login node. However it can be customised using two settings in the XNJS config file ("xnjs.xml" or "xnjs_legacy.xml").

```
  <!-- name / path of the executable -->
  <eng:Property name="globus-url-copy" value="/usr/local/bin/ ↩
      globus-url-copy"/>
  <!-- additional parameters for globus-url-copy -->
  <eng:Property name="globus-url-copy.parameters" value=""/>
```