UNICORE/X MANUAL

UNICORE Team

Document Version:	1.0.0
Component Version:	7.0.1
Date:	26 02 2014

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 OpenJDK or Oracle Java (JRE or SDK). We recommend using the latest version of Java 7.

If not installed on your system, you can download it from http://www.oracle.com/technetwork/-java/javase/downloads/index.html

If using the Oracle Java, you also need to download and install the "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files" available at the same website.

UNICORE/X has been most extensively tested on Linux-like systems, but runs on Windows and MacOS/X 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 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/lib	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
```

Startup can take some time. After a successful start, the log files (e.g. `LOG/startup.log`) contain a message "Server started." and a report on the status of any connections to other servers (e.g. the TSI or global registry).

To stop the server, do:

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

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

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

1.2.2 Log files

UNICORE/X writes its log file(s) to the LOG directory. By default, log files are rolled daily, There is no automated removal of old logs, if required you will have to do this yourself.

Details about the logging configuration are given in Section 3.2.

2 Configuration of UNICORE/X

2.1 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 sections.

2.1.1 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 Section 5.
- 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 Section 7.

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

2.2 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**
startup.properties	Java settings (e.g. memory), lib/log/conf directories	no
uas.config	General settings, startup behaviour, storages, AIP setup	yes
wsrflite.xml	Services to be deployed, SSL settings, Web server settings	yes

Table 2: (continued)

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

2.3 Settings for the UNICORE/X process (e.g. memory)

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

- UNIX: the CONF/startup.properties configuration file
- Windows: the "CONF\\wrapper.conf" configuration file

These properties include settings for maximum memory, and also the properties for configuring JMX, see Section 3 for more on JMX.

General

2.4 Config file formats

UNICORE/X uses two different formats for configuration.

2.4.1 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".

2.4.2 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

2.5 UNICORE/X container configuration overview

The following table gives an overview of the basic settings for a UNICORE/X server. These can be set in uas.config or wsrflite.xml. Many of the settings (e.g. security) will be explained in more detail in separate sections.

Property name	**Type**	**Default value / mandatory**	**Description**
`container.baseurl`	string	-	Server URL as visible from the outside, usually the gateway's address.
`container.client.[.*]`	string *can have subkeys*	-	Properties with this prefix are used to configure clients created by the container. See separate documentation for details.

Property name	Type	Default value / mandatory	Description
container.configfile	filesystem path	-	Allows to specify the location of XML configuration file of the USE container. Useful only in properties file.
container.deployment.dynamic	[true, false]	false	Controls whether dynamic deployment (at runtime) of services is enabled.
container.deployment.dynamic.jarDirectory	filesystem path	-	Directory with Jar files for dynamic deployment.
container.externalregistry.autodiscover	[true, false]	false	Whether to autodiscover registries using multicast *(runtime updateable)*
container.externalregistry.url*	list of properties with a common prefix	-	List of external registry URLs to register local services. *(runtime updateable)*
container.externalregistry.use	[true, false]	false	Whether the service should register itself in external registry(-ies), defined separately. *(runtime updateable)*
container.host	string	localhost	Server interface to listen on.
container.httpServer.[.*]	string *can have subkeys*	-	Properties with this prefix are used to configure container's Jetty HTTP server. See separate documentation for details.
container.onstartup	string	-	Space separated list of runnables to be executed on server startup. It is preferred to use onstartup.
container.onstartup.<NUMBER>	list of properties with a common prefix	-	List of runnables to be executed on server startup.
container.onstartupSelftest	[true, false]	true	Controls whether to run tests of connections to external services on startup.

Property name	Type	Default value / mandatory	Description
container.persistence.[.*]	string *can have subkeys*	-	Properties with this prefix are used to configure container's persistence layer. See separate documentation for details.
container.port	integer [0 — 65535]	7777	Server listen port.
container.registry.globalAdvertise	[true, false]	false	If this server runs a global registry, this setting controls whether it is advertised using multicast.
container.resources.executor.idletime	integer number	60000	The timeout in millis for removing idle threads.
container.resources.executor.maxsize	integer number	32	The maximum thread pool size for the scheduled execution service
container.resources.executor.minsize	integer number	10	The minimum thread pool size for the scheduled execution service
container.resources.scheduled.idletime	integer number	60000	Timeout in millis for removing idle threads.
container.resources.scheduled.size	integer >= 1	3	Defines the thread pool size for the execution of scheduled services.
container.security.[.*]	string *can have subkeys*	-	Properties with this prefix are used to configure container's security. See separate documentation for details.
container.servletpath	string	/services	Servlet context path. In most cases shouldn't be changed.
container.sitename	string	DEMO-SITE	Short, human friendly, name of the target system, should be unique in the grid.

Property name	Type	Default value / mandatory	Description
`container.unicore.wsrflite.isPersistent[.*]`	[true, false] *can have subkeys*	`false`	Global setting controlling persistence of WS-resources state. Additionally it can be used as a per-service setting, after appending a dot and service name to the property key.
`container.wsrf.expirycheck.initial[.*]`	integer number *can have subkeys*	`120`	The initial delay for WS-resource expiry checking (seconds). Additionally it can be used as a per-service setting, after appending a dot and service name to the property key.
`container.wsrf.expirycheck.period[.*]`	integer number *can have subkeys*	`60`	The interval for WS-resource expiry checking (seconds). Additionally it can be used as a per-service setting, after appending a dot and service name to the property key.
`container.wsrf.instanceLockingTimeout[.*]`	integer number *can have subkeys*	`30`	The timeout when attempting to lock WS-resources. Additionally it can be used as a per-service setting, after appending a dot and service name to the property key.
`container.wsrf.lifetime.default[.*]`	integer >= 1 *can have subkeys*	`86400`	Default lifetime of WS-resources (in seconds). Add dot and service name as a suffix of this property to set a default per particular service type.
`container.wsrf.lifetime.maximum[.*]`	integer >= 1 *can have subkeys*	-	Maximum lifetime of WS-resources (in seconds). Add dot and service name as a suffix of this property to set a limit per particular service type.

Property name	Type	Default value / mandatory	Description
container.wsrf.maxInstancesPerUser[.*]	integer >= 1 *can have subkeys*	2147483647	Maximum number per user of WS-resource instances. Add dot and service name as a suffix of this property to set a limit per particular service type.
container.wsrf.persistence.persist	string	de.fzj.unicore.wsrflite.persistence.InMemory	Implementation used to maintain the persistence of WS-resources state.
container.wsrf.sg.defaulttermtime	integer >= 1	1800	The default termination time of service group entries in seconds.

2.6 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 and (optionally) a Registry.

2.6.1 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 container.security.gateway.* properties.

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

2.6.2 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.

For example

```
container.externalregistry.use=true
container.externalregistry.url=https://host1:8080/REGISTRY/services ↩
    /Registry?res=default_registry
container.externalregistry.url2=https://host2:8080/BACKUP/services/ ↩
    Registry?res=default_registry
```

2.7 Startup code

In order to provide a flexible initialization process for the UAS, we introduce a property named "container.onstartup". The value(s) 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 3: 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.GLUEOnStartup	Initialises the GLUE2 infoprovider	UNICORE/X servers that want to provide information in GLUE2 format or want to be visible in the CIS
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	Initialises and deploys a single instance of the SMS that is shared between users, named default_storage	if a shared storage is required

2.8 Security

2.8.1 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 assign an authorisation role, UNIX login, group and other properties to Grid users. 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 Section 5.
- Access control setup (controlling in detail who can do what on which services). Again, several options exist, which are described in Section 10.
- Message level security (message signatures). UNICORE can be configured (and is so by default) to require digital signatures on important messages (like job submissions or file exports). When enabled UNICORE guarantees non-repudiation, i.e. the client can not deny that it invoked the operation. Check the table below for the option allowing to disable this feature to have a bit better performance.

2.8.2 General security options

This table presents all security related options, except credential and truststore settings which are described in the subsequent section.

Property name	**Type**	**Default value / mandatory**	**Description**
`container.security.accesscontrol[.*]`	[true, false] *can have subkeys*	`true`	Controls whether access checking (authorisation) is enabled. Can be used per service after adding dot and service name to the property key. *(runtime updateable)*

Property name	Type	Default value / mandatory	Description
container.security.accesscontrol.pdp	Class extending de.fzj.unicore.wsrflite.security.pdp.UnicoreXPDP	-	Controls which Policy Decision Point (PDP, the authorisation engine) should be used. Default value is determined as follows: if eu.unicore.uas.pdp.local.LocalHerasafPDP is available then it is used. If not then this option becomes mandatory.
container.security.accesscontrol.pdpConfig	filesystem path	-	Path of the PDP configuration file
container.security.attributes[.*]	string *can have subkeys*	-	Prefix used for configurations of particular attribute sources.
container.security.attributes.combiningPolicy	string	MERGE_LAST_OVERRIDES	What algorithm should be used for combining the attributes from multiple attribute sources (if more then one is defined).
container.security.attributes.order	string	-	Attribute sources in invocation order.
container.security.credential.[.*]	string *can have subkeys*	-	Properties with this prefix are used to configure the credential used by the container. See separate documentation for details.
container.security.defaultVOs.<NUMBER>	list of properties with a common prefix	*empty string*	List of default VOs, which should be assigned for a request without a VO set. The first VO on the list where the user is member will be used.
container.security.delegationTruststore.[.*]	string *can have subkeys*	-	When separateDelegationTruststore is true allows to configure the trust delegation truststore (using normal truststore properties with this prefix).

Property name	Type	Default value / mandatory	Description
container.security.dynamicAttributes[.*]	string *can have subkeys*	-	Prefix used for configurations of particular dynamic attribute sources.
container.security.dynamicAttributes.combiningPolicy	string	MERGE_LAST_OVERRIDES	What algorithm should be used for combining the attributes from multiple dynamic attribute sources (if more then one is defined).
container.security.dynamicAttributes.order	string	-	Dynamic attribute sources in invocation order.
container.security.gateway.certificate	filesystem path	-	Path to gateway's certificate file in PEM or DER format. Note that DER format is used only for files with *.der* extension. It is used only for gateway's authentication assertions verification (if enabled). Note that this is not needed to set it if waiting for gateway on startup is turned on.
container.security.gateway.checkSignature	[true, false]	true	Controls whether gateway's authentication assertions are verified.
container.security.gateway.enable	[true, false]	true	Whether to accpet gateway-based authentication. Note that if it is enabled either the site must be secured (usually via firewall) to disable non-gateway access or the verification of gateway's assertions must be enabled.
container.security.gateway.registration	[true, false]	false	Whther the site should try to autoregister itself with the Gateway. This must be also configured on the Gateway side.

Property name	Type	Default value / mandatory	Description
container.security.gateway.registrationUpdateInterval	integer >= 10	30	How often the automatic gateway registration should be refreshed.
container.security.gateway.waitOnStartup	[true, false]	true	Controls whether to wait for the gateway at startup.
container.security.gateway.waitTime	integer >= 1	180	Controls for how long to wait for the gateway on startup (in seconds).
container.security.separateDelegationTruststore	[true, false]	false	Significant for XSEDE integration: when turned on, allows for using a separate truststore for delegation checking then the one used for SSL connections checking.
container.security.sessionLifetime	integer >= 1	28800	Controls the lifetime of security sessions (in seconds).
container.security.sessionsEnabled	[true, false]	true	Controls whether the server supports security sessions which reduce client/server traffic and load.
container.security.sessionsPerUser	integer >= 1	5	Controls the number of security sessions each user can have. If exceeded, some cleanup will be performed.
container.security.signatures	[true, false]	false	Controls whether signatures (providing non-repudiation guarantees) on key requests should be required. If the system is setup without user certificates, signatures must be disabled.
container.security.sslEnabled	[true, false]	true	Controls whether secure SSL mode is enabled.

Property name	Type	Default value / mandatory	Description
`container.security.trustedAssertionIssuers.[.*]`	string *can have subkeys*	-	Allows for configuring a truststore (using normal truststore properties with this prefix) with certificates of trusted services (not CAs!) which are permitted to issue trust delegations and authenticate with SAML. Typically this truststore should contain certificates of all Unity instanes installed.
`container.security.truststore.-[.*]`	string *can have subkeys*	-	Properties with this prefix are used to configure container's trust settings and certificates validation. See separate documentation for details.

2.8.3 Configuring PKI trust settings

Public Key Infrastructure (PKI) trust settings are used to validate certificates. This is performed, in the first place when a connection with a remote peer is initiated over the network, using the SSL (or TLS) protocol. Additionally certificate validation can happen in few other situations, e.g. when checking digital signatures of various sensitive pieces of data.

Certificates validation is primarily configured using a set of initially trusted certificates of so called Certificate Authorities (CAs). Those trusted certificates are also known as *trust anchors* and their collection is called a *trust store*.

Except of *trust anchors* validation mechanism can use additional input for checking if a certificate being checked was not revoked and if its subject is in a permitted namespace.

UNICORE allows for different types of trust stores. All of them are configured using a set of properties.

- *Keystore trust store* - the only format supported in older UNICORE versions. Trusted certificates are stored in a single binary file in JKS or PKCS12 format. The file can be only manipulated using a special tool like JDK *keytool* or openssl (in case of PKCS12 format). This format is great if trust store should be in a single file or when compatibility with other Java solutions or older UNICORE releases is desired.
- *OpenSSL trust store* - allows to use a directory with CA certificates stored in PEM format, under precisely defined names: the CA certificates, CRLs, signing policy files and namespaces files are named <hash>.0, <hash>.r0, <hash>.signing_policy and <hash>.namespaces.

Hash is the old hash of the trusted CA certificate subject name (in Openssl version > 1.0.0 use -suject_hash_old switch to generate it). If multiple certificates have the same hash then the default zero number must be increased. This format is the same as used by other then UNICORE popular middlewares as Globus and gLite. It is suggested when a common trust store with such middlewares is needed.

- *Directory trust store* - the most flexible and convenient option, suggested for all remaining cases. It allows to use a list of wildcard expressions, concrete paths of files or even URLs to remote files as a set of trusted CAs and in the same way for the CRLs. With this trust store administrator can simply configure all files (or all with a specified extension) in a directory to be used as a trusted certificates.

In all cases trust stores can be (and by default are) configured to be automatically refreshed.

Property name	**Type**	**Default value / mandatory**	**Description**
`container.security.truststore.allowProxy`	[ALLOW, DENY]	`ALLOW`	Controls whether proxy certificates are supported.
`container.security.truststore.type`	[keystore, openssl, directory]	*mandatory to be set*	The truststore type.
`container.security.truststore.updateInterval`	integer number	`600`	How often the truststore should be reloaded, in seconds. Set to negative value to disable refreshing at runtime. *(runtime updateable)*
--- Directory type settings ---			
`container.security.truststore.directoryConnectionTimeout`	integer number	`15`	Connection timeout for fetching the remote CA certificates in seconds.
`container.security.truststore.directoryDiskCachePath`	filesystem path	-	Directory where CA certificates should be cached, after downloading them from a remote source. Can be left undefined if no disk cache should be used. Note that directory should be secured, i.e. normal users should not be allowed to write to it.

Property name	Type	Default value / mandatory	Description
container.security.truststore.directoryEncoding	[PEM, DER]	PEM	For directory truststore controls whether certificates are encoded in PEM or DER.
container.security.truststore.directoryLocations.*	list of properties with a common prefix	-	List of CA certificates locations. Can contain URLs, local files and wildcard expressions. *(runtime updateable)*
--- Keystore type settings ---			
container.security.truststore.keystoreFormat	string	-	The keystore type (jks, pkcs12) in case of truststore of keystore type.
container.security.truststore.keystorePassword	string	-	The password of the keystore type truststore.
container.security.truststore.keystorePath	string	-	The keystore path in case of truststore of keystore type.
--- Openssl type settings ---			
container.security.truststore.opensslNewStoreFormat	[true, false]	false	In case of openssl truststore, specifies whether the trust store is in openssl 1.0.0+ format (true) or older openssl 0.x format (false)
container.security.truststore.opensslPath	filesystem path	/etc/grid-security/certificates	Directory to be used for opeenssl truststore.
--- Revocation settings ---			
container.security.truststore.crlConnectionTimeout	integer number	15	Connection timeout for fetching the remote CRLs in seconds (not used for Openssl truststores).

Property name	Type	Default value / mandatory	Description
container.security.truststore.crlDiskCachePath	filesystem path	-	Directory where CRLs should be cached, after downloading them from remote source. Can be left undefined if no disk cache should be used. Note that directory should be secured, i.e. normal users should not be allowed to write to it. Not used for Openssl truststores.
container.security.truststore.crlLocations.*	list of properties with a common prefix	-	List of CRLs locations. Can contain URLs, local files and wildcard expressions. Not used for Openssl truststores. *(runtime updateable)*
container.security.truststore.crlMode	[REQUIRE, IF_VALID, IGNORE]	IF_VALID	General CRL handling mode. The IF_VALID setting turns on CRL checking only in case the CRL is present.
container.security.truststore.crlUpdateInterval	integer number	600	How often CRLs should be updated, in seconds. Set to negative value to disable refreshing at runtime. *(runtime updateable)*
container.security.truststore.ocspCacheTtl	integer number	3600	For how long the OCSP responses should be locally cached in seconds (this is a maximum value, responses won't be cached after expiration)
container.security.truststore.ocspDiskCache	filesystem path	-	If this property is defined then OCSP responses will be cached on disk in the defined folder.
container.security.truststore.ocspLocalResponders.<NUMBER>	list of properties with a common prefix	-	Optional list of local OCSP responders

Property name	Type	Default value / mandatory	Description
container.security.truststore.ocspMode	[REQUIRE, IF_AVAILABLE, IGNORE]	IF_AVAILABLE	General OCSP ckecking mode. REQUIRE should not be used unless it is guaranteed that for all certificates an OCSP responder is defined.
container.security.truststore.ocspTimeout	integer number	10000	Timeout for OCSP connections in miliseconds.
container.security.truststore.revocationOrder	[CRL_OCSP, OCSP_CRL]	OCSP_CRL	Controls overal revocation sources order
container.security.truststore.revocationUseAll	[true, false]	false	Controls whether all defined revocation sources should be always checked, even if the first one already confirmed that a checked certificate is not revoked.

Examples

Note
Various UNICORE modules use different property prefixes. Here we don't put any, but in practice you have to use the prefix (see the reference table above for the actual prefix). Also properties might need to be provided using different syntax, as XML.

Directory trust store, with a minimal set of options:

```
truststore.type=directory
truststore.directoryLocations.1=/trust/dir/*.pem
truststore.crlLocations=/trust/dir/*.crl
```

Directory trust store, with a complete set of options:

```
truststore.type=directory
truststore.allowProxy=DENY
truststore.updateInterval=1234
truststore.directoryLocations.1=/trust/dir/*.pem
truststore.directoryLocations.2=http://caserver/ca.pem
truststore.directoryEncoding=PEM
```

```
truststore.directoryConnectionTimeout=100
truststore.directoryDiskCachePath=/tmp
truststore.crlLocations.1=/trust/dir/*.crl
truststore.crlLocations.2=http://caserver/crl.pem
truststore.crlUpdateInterval=400
truststore.crlMode=REQUIRE
truststore.crlConnectionTimeout=200
truststore.crlDiskCachePath=/tmp
```

Openssl trust store:

```
truststore.type=openssl
truststore.opensslPath=/truststores/openssl
truststore.opensslNsMode=EUGRIDPMA_GLOBUS_REQUIRE
truststore.allowProxy=ALLOW
truststore.updateInterval=1234
truststore.crlMode=IF_VALID
```

Java keystore used as a trust store:

```
truststore.type=keystore
truststore.keystorePath=src/test/resources/certs/truststore.jks
truststore.keystoreFormat=JKS
truststore.keystorePassword=xxxxxx
```

2.8.4 Configuring the credential

UNICORE uses private key and a corresponding certificate (called together as a *credential*) to identify users and servers. Credentials might be provided in several formats:

- Credential can be obtained from a *keystore file*, encoded in JKS or PKCS12 format.
- Credential can be loaded as a pair of PEM files (one with private key and another with certificate),
- or from a pair of DER files,
- or even from a single file, with PEM-encoded certificates and private key (in any order).

The following table list all parameters which allows for configuring the credential. Note that nearly all options are optional. If not defined, the format is tried to be guessed. However some credential formats require additional settings. For instance if using *der* format the *keyPath* is mandatory as you need two DER files: one with certificate and one with the key (and the latter can not be guessed).

Property name	Type	Default value / mandatory	Description
`container.security.credential.path`	filesystem path	*mandatory to be set*	Credential location. In case of *jks*, *pkcs12* and *pem* store it is the only location required. In case when credential is provided in two files, it is the certificate file path.
`container.security.credential.format`	[jks, pkcs12, der, pem]	-	Format of the credential. It is guessed when not given. Note that *pem* might be either a PEM keystore with certificates and keys (in PEM format) or a pair of PEM files (one with certificate and second with private key).
`container.security.credential.password`	string	-	Password required to load the credential.
`container.security.credential.keyPath`	string	-	Location of the private key if stored separately from the main credential (applicable for *pem* and *der* types only),
`container.security.credential.keyPassword`	string	-	Private key password, which might be needed only for *jks* or *pkcs12*, if key is encrypted with different password then the main credential password.
`container.security.credential.keyAlias`	string	-	Keystore alias of the key entry to be used. Can be ignored if the keystore contains only one key entry. Only applicable for *jks* and *pkcs12*.

Examples

Note
Various UNICORE modules use different property prefixes. Here we don't put any, but in practice you have to use the prefix (see the reference table above for the actual prefix). Also properties might need to be provided using different syntax, as XML.

Credential as a pair of DER files:

```
credential.format=der
credential.password=the\!njs
credential.path=/etc/credentials/cert-1.der
credential.keyPath=/etc/credentials/pk-1.der
```

Credential as a JKS file (credential type can be autodetected in almost every case):

```
credential.path=/etc/credentials/server1.jks
credential.password=xxxxxx
```

2.9 Configuring the XNJS and TSI

Information on the configuration of the XNJS and TSI backend can be found in Section 7.

2.10 Configuring storages on TargetSystem instances

Each TargetSystem instance can have one or more storages attached to it. Note that this is different case from the shared storage (the one created with CreateSMSOnStartup hook) which is not attached to any particular TargetSystem. The practical difference is that to use storages attached to a TargetSystem, a user must first create one.

By default, NO storages are created.

For example, to allows users access their home directory on the target system, you need to add a storage. This is done using configuration entries in uas.config.

Property name	Type	Default value / mandatory	Description
`coreServices.targetsystem.storage.N.checkExistence`	[true, false]	`true`	Whether the existence of the base directory should be checked when creating the storage.

Property name	Type	Default value / mandatory	Description
coreServices.targetsystem.storage.N.class	Class extending de.fzj.unicore.uas.StorageManagement	-	Storage implementation class used (and mandatory) in case of the CUSTOM type.
coreServices.targetsystem.storage.N.cleanup	[true, false]	false	Whether files of the storage should be removed when the storage is destroyed. This is mostly useful for storage factories. *(runtime updateable)*
coreServices.targetsystem.storage.N.defaultUmask	integer number	77	Default (initial) umask for files in the storage. Must be an octal number. Note that this property is not updateable at runtime for normal storages as it wouldn't have sense (it is the initial umask by definition). However in case of storage factory it is, i.e. after the property change, the SMSes created by the factory will use the new umask as the initial one. At runtime the SMS umask can be changed by the clients (if are authorized to do so).
coreServices.targetsystem.storage.N.description	string	Filesystem	Description of the storage. It will be presented to the users. *(runtime updateable)*
coreServices.targetsystem.storage.N.disableMetadata	[true, false]	false	Whether the metadata service should be disabled for this storage.
coreServices.targetsystem.storage.N.disableTrigger	[true, false]	false	Whether the triggering feature should be disabled for this storage.

Property name	Type	Default value / mandatory	Description
`coreServices.targetsystem.storage.N.filterFiles`	[true, false]	`false`	If set to true then this SMS will filter returned files in response of the ListDirectory command: only files owned or accessible by the caller will be returned. *(runtime updateable)*
`coreServices.targetsystem.storage.N.name`	string	-	Storage name. If not set then the identifier is used.
`coreServices.targetsystem.storage.N.path`	string	-	Denotes a storage base path or the name of an environment variable in case of the VARIABLE type.
`coreServices.targetsystem.storage.N.protocols`	string	-	Which protocols to enable, default is defined by the global container setting. *(runtime updateable)*
`coreServices.targetsystem.storage.N.settings.-[.*]`	string *can have subkeys*	-	Useful for CUSTOM storage types: allows to set additional settings (if needed) by such storages. Please refer to documentation of a particular custom storage type for details. Note that while in general updates of the properties at runtime are propagated to the chosen implementation, it is up to it to use the updated values or ignore changes. *(runtime updateable)*
`coreServices.targetsystem.storage.N.type`	[HOME, FIXEDPATH, CUSTOM, VARIABLE]	`FIXEDPATH`	Storage type. FIXEDPATH: mapped to a fixed directory, VARIABLE: resolved using an environmental variable lookup, CUSTOM: specified class is used.

Property name	Type	Default value / mandatory	Description
`coreServices.targetsystem.storage.N.workdir`	string	-	(DEPRECATED, use *path* instead)

Here, "N" stands for an identifier (e.g. 1,2, 3, ...) to distinguish the storages. For example, to configure three storages (Home, 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:

```
coreServices.targetsystem.storage.0.name=Home
coreServices.targetsystem.storage.0.type=HOME

coreServices.targetsystem.storage.1.name=TEMP
coreServices.targetsystem.storage.1.type=FIXEDPATH
coreServices.targetsystem.storage.1.path=/tmp
coreServices.targetsystem.storage.1.protocols=UFTP BFT

coreServices.targetsystem.storage.2.name=DEISA_HOME
coreServices.targetsystem.storage.2.type=VARIABLE
coreServices.targetsystem.storage.2.path=$DEISA_HOMES

# example for a custom SMS implementation (e.g. for Hadoop or iRODS ↩
    )
coreServices.targetsystem.storage.3.name=IRODS
coreServices.targetsystem.storage.3.type=CUSTOM
coreServices.targetsystem.storage.3.path=/
coreServices.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.

2.10.1 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:

```
coreServices.targetsystem.uniqueStorageIds=true
```

2.11 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

```
coreServices.sms.enabledFactories=TYPE1 TYPE2 ...
```

Each supported storage type is configured using a set of properties

Property name	**Type**	**Default value / mandatory**	**Description**
`coreServices.sms.factory.N.checkExistence`	[true, false]	`true`	Whether the existence of the base directory should be checked when creating the storage.
`coreServices.sms.factory.N.class`	Class extending de.fzj.unicore.uas.StorageManagement	-	Storage implementation class used (and mandatory) in case of the CUSTOM type.
`coreServices.sms.factory.N.cleanup`	[true, false]	`false`	Whether files of the storage should be removed when the storage is destroyed. This is mostly useful for storage factories. *(runtime updateable)*
`coreServices.sms.factory.N.defaultUmask`	integer number	`77`	Default (initial) umask for files in the storage. Must be an octal number. Note that this property is not updateable at runtime for normal storages as it wouldn't have sense (it is the initial umask by definition). However in case of storage factory it is, i.e. after the property change, the SMSes created by the factory will use the new umask as the initial one. At runtime the SMS umask can be changed by the clients (if are authorized to do so).

Property name	Type	Default value / mandatory	Description
`coreServices.sms.factory.N.description`	string	`Filesystem`	Description of the storage. It will be presented to the users. *(runtime updateable)*
`coreServices.sms.factory.N.disableMetadata`	[true, false]	`false`	Whether the metadata service should be disabled for this storage.
`coreServices.sms.factory.N.disableTrigger`	[true, false]	`false`	Whether the triggering feature should be disabled for this storage.
`coreServices.sms.factory.N.filterFiles`	[true, false]	`false`	If set to true then this SMS will filter returned files in response of the ListDirectory command: only files owned or accessible by the caller will be returned. *(runtime updateable)*
`coreServices.sms.factory.N.name`	string	-	Storage name. If not set then the identifier is used.
`coreServices.sms.factory.N.path`	string	-	Denotes a storage base path or the name of an environment variable in case of the VARIABLE type.
`coreServices.sms.factory.N.protocols`	string	-	Which protocols to enable, default is defined by the global container setting. *(runtime updateable)*
`coreServices.sms.factory.N.settings.[.*]`	string *can have subkeys*	-	Useful for CUSTOM storage types: allows to set additional settings (if needed) by such storages. Please refer to documentation of a particular custom storage type for details. Note that while in general updates of the properties at runtime are propagated to the chosen implementation, it is up to it to use the updated values or ignore changes. *(runtime updateable)*

Property name	Type	Default value / mandatory	Description
coreServices.sms.factory.N.type	[HOME, FIXEDPATH, CUSTOM, VARIABLE]	FIXEDPATH	Storage type. FIXEDPATH: mapped to a fixed directory, VARIABLE: resolved using an environmental variable lookup, CUSTOM: specified class is used.
coreServices.sms.factory.N.workdir	string	-	(DEPRECATED, use *path* instead)

For example

```
coreServices.sms.factory.TYPE1.description=GPFS file system
coreServices.sms.factory.TYPE1.fixedpath=GPFS file system
coreServices.sms.factory.TYPE1.path=/mnt/gpfs/unicore/unicorex-1/ ↩
    storage-factory
coreServices.sms.factory.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"
coreServices.sms.factory.TYPE1.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 normal storage properties (see the previous section) are also accepted: "protocols", "type", "class", "filterFiles" etc.

If you have a custom storage type, an additional "class" parameter defines the Java class name to use (as in normal SMS case). For example:

```
coreServices.sms.factory.TYPE1.type=CUSTOM
coreServices.sms.factory.TYPE1.class=de.fzj.unicore.uas.hadoop. ↩
    SMSHadoopImpl
```

2.12 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.

The server options are shown in the following table.

Property name	Type	Default value / mandatory	Description
container.httpServer.disabledCipherSuites	string	*empty string*	Space separated list of SSL cipher suites to be disabled.
container.httpServer.fastRandom	[true, false]	false	Use insecure, but fast pseudo random generator to generate session ids instead of secure generator for SSL sockets.
container.httpServer.gzip.enable	[true, false]	false	Controls whether to enable compression of HTTP responses.
container.httpServer.gzip.minGzipSize	integer number	100000	Specifies the minimal size of message that should be compressed.
container.httpServer.highLoadConnections	integer >= 1	200	If the number of connections exceeds this amount, then the connector is put into a special *low on resources* state. Existing connections will be closed faster. Note that this value is honored only for NIO connectors. Legacy connectors go into low resources mode when no more threads are available.
container.httpServer.lowResourceMaxIdleTime	integer >= 1	100	In low resource conditions, time (in ms.) before an idle connection will time out.
container.httpServer.maxIdleTime	integer >= 1	200000	Time (in ms.) before an idle connection will time out. It should be large enough not to expire connections with slow clients, values below 30s are getting quite risky.
container.httpServer.maxThreads	integer >= 1	255	Maximum number of threads to have in the thread pool for processing HTTP connections.
container.httpServer.minThreads	integer >= 1	1	Minimum number of threads to have in the thread pool for processing HTTP connections.

Property name	Type	Default value / mandatory	Description
container.httpServer.requireClientAuthn	[true, false]	true	Controls whether the SSL socket requires client-side authentication.
container.httpServer.soLingerTime	integer number	-1	Socket linger time.
container.httpServer.useNIO	[true, false]	true	Controls whether the NIO connector be used. NIO is best suited under high-load, when lots of connections exist that are idle for long periods.
container.httpServer.wantClientAuthn	[true, false]	true	Controls whether the SSL socket accepts (but does not require) client-side authentication.

The client options are the following

Property name	Type	Default value / mandatory	Description
container.client.digitalSigningEnabled	[true, false]	true	Controls whether signing of key web service requests should be performed.
container.client.httpAuthnEnabled	[true, false]	false	Whether HTTP basic authentication should be used.
container.client.httpPassword	string	*empty string*	Password for use with HTTP basic authentication (if enabled).
container.client.httpUser	string	*empty string*	Username for use with HTTP basic authentication (if enabled).
container.client.inHandlers	string	*empty string*	Space separated list of additional handler class names for handling incoming WS messages

Property name	Type	Default value / mandatory	Description
container.client.maxWsCallRetries	integer number	3	Controls how many times the client should try to call a failing web service. Note that only the transient failure reasons cause the retry. Note that value of 0 enables unlimited number of retries, while value of 1 means that only one call is tried.
container.client.messageLogging	[true, false]	false	Controls whether messages should be logged (at INFO level).
container.client.outHandlers	string	*empty string*	Space separated list of additional handler class names for handling outgoing WS messages
container.client.securitySessions	[true, false]	true	Controls whether security sessions should be enabled.
container.client.serverHostnameChecking	[NONE, WARN, FAIL]	WARN	Controls whether server's hostname should be checked for matching its certificate subject. This verification prevents man-in-the-middle attacks. If enabled WARN will only print warning in log, FAIL will close the connection.
container.client.sslAuthnEnabled	[true, false]	true	Controls whether SSL authentication of the client should be performed.
container.client.wsCallRetryDelay	integer number	10000	Amount of milliseconds to wait before retry of a failed web service call.
--- HTTP client settings ---			
container.client.http.allow-chunking	[true, false]	true	If set to false, then the client will not use HTTP 1.1 data chunking.
container.client.http.connection-close	[true, false]	false	If set to true then the client will send connection close header, so the server will close the socket.

Property name	Type	Default value / mandatory	Description
container.client.http.connection.timeout	integer number	20000	Timeout for the connection establishing (ms)
container.client.http.maxPerRoute	integer number	6	How many connections per host can be made. Note: this is a limit for a single client object instance.
container.client.http.maxRedirects	integer number	3	Maximum number of allowed HTTP redirects.
container.client.http.maxTotal	integer number	20	How many connections in total can be made. Note: this is a limit for a single client object instance.
container.client.http.socket.timeout	integer number	0	Socket timeout (ms)
--- HTTP proxy settings ---			
container.client.http.nonProxyHosts	string	-	Space (single) separated list of hosts, for which the HTTP proxy should not be used.
container.client.http.proxy.password	string	-	Relevant only when using HTTP proxy: defines password for authentication to the proxy.
container.client.http.proxy.user	string	-	Relevant only when using HTTP proxy: defines username for authentication to the proxy.
container.client.http.proxyHost	string	-	If set then the HTTP proxy will be used, with this hostname.
container.client.http.proxyPort	integer number	-	HTTP proxy port. If not defined then system property is consulted, and as a final fallback 80 is used.
container.client.http.proxyType	string	HTTP	HTTP proxy type: HTTP or SOCKS.

3 Administration

3.1 Controlling UNICORE/X memory usage

You can set a limit on the number of service instances (e.g. jobs) per user. This allows you to make sure your server stays nicely up and running even if flooded by jobs. To enable, edit `CONF/wsrflite.xml` and add properties, e.g.

```
<property name="unicore.maxInstancesPerUser.JobManagement" value ↩
    ="200"/>
<property name="unicore.maxInstancesPerUser.FileTransferBFT"  ↩
    value="20"/>
```

The last part of the property name is the service name as defined in wsrflite.xml.

When the limits are reached, the server will report an error to the client (e.g. when trying to submit a new job).

3.2 Logging

UNICORE uses the Log4j logging framework (http://logging.apache.org/log4j/1.2/), which supports many useful options, such as logging to the server's syslog (on Linux). Log4j is configured using a config file. By default, this file is `CONF/logging.properties`. To change the default, edit the start script (`CONF/startup.properties`) or, on Windows, the `CONF/wrapper.conf`. The config file is specified with a Java property `log4j.configuration`.

Note
You can change the logging configuration at runtime by editing the logging.properties file. The new configuration will take effect a few seconds after the file has been modified.

By default, log files are written to the the `LOGS` directory.

The following example config file configures logging so that log files are rotated daily.

```
# Set root logger level to INFO and its only appender to A1.
log4j.rootLogger=INFO, A1

# A1 is set to be a rolling file appender with default params
log4j.appender.A1=org.apache.log4j.DailyRollingFileAppender
log4j.appender.A1.File=logs/uas.log

#configure daily rollover: once per day the uas.log will be copied
#to a file named e.g. uas.log.2008-12-24
log4j.appender.A1.DatePattern='.'yyyy-MM-dd
```

```
# A1 uses the PatternLayout
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c{1} %x -  ↩
    %m%n
```

Note
In Log4j, the log rotation frequency is controlled by the DatePattern. Check http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/DailyRollingFileAppender.html for the details.

Within the logging pattern, you can use special variables to output information. In addition to the variables defined by Log4j (such as *%d*), UNICORE defines several variables related to the client and the current job.

Variable	Description
%X{clientName}	the distinguished name of the current client
%X{jobID}	the unique ID of the currently processed job

A sample logging pattern might be

```
log4j.appender.A1.layout.ConversionPattern=%d [%X{clientName}] [%X{ ↩
    jobID}] [%t] %-5p %c{1} %x - %m%n
```

For more info on controlling the logging we refer to the log4j documentation:

- PatternLayout
- RollingFileAppender
- DailyRollingFileAppender

Log4j supports a very wide range of logging options, such as date based or size based file rollover, logging different things to different files and much more. For full information on Log4j we refer to the publicly available documentation, for example the Log4j manual.

3.2.1 Logger categories, names and levels

Logger names are hierarchical. In UNICORE, prefixes are used (e.g. "unicore.security") to which the Java class name is appended. For example, the XUUDB connector in UNICORE/X logs to the "unicore.security.XUUDBAuthoriser" logger.

Therefore the logging output produced can be controlled in a fine-grained manner. Log levels in Log4j are (in increasing level of severity) TRACE, DEBUG, INFO, WARN, ERROR, amd FATAL.

For example, to debug a security/authorisation problem in the UNICORE/X security layer, you can set

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

If you are just interested in XUUDB related output, you can set

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

so the XUUDBAuthoriser will log on DEBUG level, while the other security components log on INFO level.

Here is a table of the various logger categories

Log category	**Description**
unicore	All of UNICORE
unicore.security	Security layer
unicore.services	Service operational information
unicore.services.jobexecution	Information related to job execution
unicore.services.jobexecution.USAGE	Usage logging (see next section)
unicore.xnjs	XNJS subsystem (execution engine)
unicore.xnjs.tsi	TSI subsystem (batch system connector)
unicore.client	Client calls (to other servers)
unicore.wsrflite	Underlying services environment (WSRF framework)

Note
Please take care to not set the global level to TRACE or DEBUG for long times, as this will produce a lot of output.

3.2.2 Usage logging

Often it is desirable to keep track of the usage of your UNICORE site. The UNICORE/X server has a special logger category called `unicore.services.jobexecution.USAGE` which logs information about finished jobs at INFO level. If you wish to enable this, set

```
log4j.logger.unicore.services.jobexecution.USAGE=INFO
```

Note
If you are setting up a production environment and need a sophisticated accounting solution (featuring database with a real resources consumption, WWW interface and possibility to produce reports or aggregated data) then consider deploying UNICORE RUS Accounting. Further instructions can be found in Section 7.2.

It might be convenient to send usage output to a different file than normal log output. This is easily achieved with log4j:

```
# send usage logger output to a separate file

# use separate appender 'U1' for usage info
log4j.logger.unicore.services.jobexecution.USAGE=INFO,U1

# U1 is set to be a rolling file appender with default params
log4j.appender.U1=org.apache.log4j.DailyRollingFileAppender
log4j.appender.U1.File=logs/usage.log
# U1 uses the PatternLayout
log4j.appender.U1.layout=org.apache.log4j.PatternLayout
log4j.appender.U1.layout.ConversionPattern=%d [%t] %-5p %c{1} %x -  ↩
    %m%n
```

For each finished job, the usage logger will log a line with the following information (if available)

```
[result] [executable] [actionUUID] [clientDN] [BSSJobId] [ ↩
    clientXlogin] [jobName] [machineName] [VOs]
```

An example output line is:

```
2011-08-16 10:00:39,513 [XNJS-1-JobRunner-1] INFO  USAGE  - [ ↩
    SUCCESSFUL] [ /bin/date] [ e9deab79-af1f-4704-a6bd-427b3ab20969 ↩
    ] [CN=Bernd Schuller, OU=VSGC, OU=Forschungszentrum Juelich  ↩
    GmbH, O=GridGermany, C=DE] [82942] [schuller] [Date job  ↩
    submitted using UCC] [zam025c02.zam.kfa-juelich.de] []
```

3.3 Administration and monitoring

The health of a UNICORE/X container, and things like running services, lifetimes, etc. can be monitored in several ways.

3.3.1 Commandline client (UCC)

It is possible to use the UNICORE commandline client (UCC) to list jobs, extend lifetimes, etc.

The trick is to configure UCC so that it uses the *server* certificate of the UNICORE/X server, so that UCC will have administrator rights. Also you should connect directly to UNICORE/X, not to the registry as usual. Here is an example UCC configuration file. Say your UNICORE/X server is running on *myhost* on port 7777, your preferences file would look like this

```
registry=https://myhost:7777/services/Registry?res=default_registry

# use UNICORE/X keystore
credential.path=/path/to/unicorex/keystore
credential.password=...

# truststore config omitted
```

Note that the registry URL points directly to the UNICORE/X server, not to a gateway.

Examples

Some UCC commands that are useful are the *list-jobs*, *list-sites* and *wsrf* commands. Using *list-jobs* you can search for jobs with given properties, whereas the *wsrf* command allows to look at any resource, or even destroy resources.

To list all jobs on the server belonging to a specific user, do

```
ucc list-jobs -f Log contains <username>
```

where *username* is some unique part of the user's DN, or the xlogin. Similarly, you can filter based on other properties of the job.

The *wsrf* command can be used to destroy resources, extend their lifetime and look at their properties. Please see "ucc wsrf -h" for details.

Try

```
ucc wsrf getproperties https://myhost:7777/services/ ↩
    TargetSystemFactory?res=default_target_system_factory
```

3.3.2 The Admin web service

The Admin service is a powerful tool to get "inside information" about your server using the UCC (or possibly another UNICORE client) and run one of the available "admin actions", which provide useful functions.

If you have enabled the admin service, you can do

```
ucc admin-info -l
```

to get information about available admin services. Note that you need to have role "admin" to invoke the admin service. The output includes information about the available administrative commands. To run one of these, you can use the *admin-runcommand* command. For example, to temporarily disable job submission

```
ucc admin-runcommand ToggleJobSubmission
```

3.3.3 Java Management Extensions (JMX)

Using the Java Management Extensions, you can monitor any Java virtual machine using (for example) a program called "jconsole" that is part of the Sun/Oracle Java SDK. It allows to check memory and thread usage, as well as access to application specific management components ("MBeans").

Enabling access

Connecting to a running Java VM locally is always possible, however remote access needs to be specially configured. To enable remote access to JMX, please check the startup.properties file of UNICORE/X. There several system properties are defined that enable and configure remote access via JMX.

```
#
#enable JMX (use jconsole to connect)
#
OPTS=$OPTS" -Dcom.sun.management.jmxremote"
#enable JMX remote access protected by username/password
OPTS=$OPTS" -Dcom.sun.management.jmxremote.port=9128 -Dcom.sun. ↩
    management.jmxremote.authenticate=true"
OPTS=$OPTS" -Dcom.sun.management.jmxremote.ssl=false"
OPTS=$OPTS" -Dcom.sun.management.jmxremote.password.file=conf/ ↩
    jmxremote.password"
```

The password file "conf/jmxremote.password" must contain lines of the form "username=password", and must have its permissions set to "rw for owner only" (at least on Unix).

Using jconsole, you can now connect from a remote machine.

3.4 Migration of a UNICORE/X server to another physical host

If you want to migrate a UNICORE/X server to another host, there are several things to consider. The hostname and port are listed in CONF/wsrflite.xml and usually in the Gateway's connection.properties file. These you will have to change. Otherwise, you can copy the relevant files in CONF to the new machine. Also, the persisted state data needs to be moved to the new machine, if it is stored on the file system. If it is stored in a database, there is nothing to be done. If you are using a Perl TSI, you might need to edit the TSI's properties file and update the `tsi.njs_machine` property.

4 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
- perform policy check by executing a PDP 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

4.1 Security concepts

4.1.1 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 Section 2).

4.1.2 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.

4.1.3 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.

4.1.4 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.

4.1.5 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 Section 5 for more information.

4.1.6 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 Section 10 describes how to configure different engines for policy evaluation including a remote one.

4.1.7 Authorisation

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

4.1.8 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 Section 12 about the configuration details.

5 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*. Attributes are also used to subsequently incarnate the authorized user and possibly can be used for other purposes as well (for instance for accounting).

Therefore the most important item for security configuration is selecting and maintaining a so called *attribute source* (called sometimes attribute information point, AIP), which is used by USE to assign attributes to Grid users.

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

There are two kinds of attribute sources:

- *Classic* or *static attribute sources*, which are used BEFORE authorization. Those attribute sources maintain a simple mappings of user certificates (or DNs) to some attributes. The primary role of those sources is to provide attributes used for authorization, but also incarnation attributes may be assigned.
- *Dynamic attribute sources*, which are used AFTER authorization, only if it was successful. Therefore these attribute sources can assign only the incarnation attributes. The difference is that attributes are collected for already authorized users, so the attributes can be assigned in dynamic way not only using the user's identity but also all the static attributes. This feature can be used for assigning pool accounts for authorized users or adding additional supplementary gids basing on user's Virtual Organization.

5.1 UNICORE incarnation and authorization attributes

Note that actual names of the attributes presented here are not very important. Real attribute names are defined by attribute source (advanced attribute sources, like UVOS/SAML attribute source, even provide a possibility to choose what attribute names are mapped to internal UNICORE attributes). Therefore it is only important to know the concepts represented by the internal UNICORE attributes. On the other hand the values which are defined below are important.

The attributes in UNICORE can be multi-valued.

There are two special authorization attributes:

- *role* - represents an abstract user's role. The role is used in a default (and rarely changed) UNICORE authorization policy and in authorization process in general. There are several possible values that are recognized by the default authorization policy:
- `user` - value specifies that the subject is allowed to use the site as a normal user (submit jobs, get results, ...)

- `admin` - value specifies that the subject is an administrator and may do everything. For example may submit jobs, get results of jobs of other users and even delete them.
- `banned` - user with this role is explicitly banned and all her request are denied.
- `trustedAgent` - this value is used to express that the subject may act as any user the subject wants. E.g. if John has this role then can get Marry's job results and also submit job as Frank. It is highly suggested not to use this role, its support is minimal mostly for backwards compatibility.
- anything else - means that user is not allowed to do anything serious. Some very basic, read-only operations are allowed, but this is a technical detail. Also access to owned resources is granted, what can happen if the user had the `user` role before. Typically it is a good practice to use value `banned` in such case.
- *virtualOrganisations* - represents an abstract, Grid group of the user. By default it is not used directly anywhere in the core stack, but several subsystems (as dynamic attribute sources or jobs accounting) may be configured to use it.

There are several attributes used for incarnation:

- *xlogin* - specifies what local user id (in UNIX called *uid*) should be assigned for the grid user.
- *group* - specifies the primary group (primary gid) that the grid user should get.
- *supplementaryGroups* - specifies all supplementary groups the grid user should get.
- *addDefaultGroups* - boolean attribute saying whether groups assigned to the Xlogin (i.e. the local uid of the grid user) in the operating system should be additionally added for the grid user.
- *queue* - override IDB queues setting for the particular user, defining available BSS queues.

Finally UNICORE can consume other attributes. All other attributes can be used only for authorization or in more advanced setups (for instance using the Unicore/X incarnation tweaker). Currently all such additional attributes which are received from attribute source are treated as XACML attributes and are put into XACML evaluation context. This feature is rather rarely used, but it allows for creating a very fine grained authorization policies, using custom attributes.

Particular attribute source define how to assign these attribute to users. Not always all types of attributes are supported by the attribute source, e.g. XUUDB can not define (among others) per-user queues or VOs.

After introducing all the special UNICORE attributes, it must be noted that those attributes are used in two ways. Their primary role is to strictly define what is allowed for the user. For instance the *'Xlogin' values specify the valid uids from which the user may choose one*. One exception here is *Add operating system groups* - user is always able to set this according to his/her preference.

The second way of using those attributes is to specify the default behavior, when the user is not expressing a preference. E.g. a default *Group* (which must be single valued) specify which group should be used, if user doesn't provide any.

Attribute sources define the permitted values and default values for the attributes in various ways. Some use conventions (e.g. that first permitted value is a default one), some use a pair of real attributes to define the valid and default values of one UNICORE attribute.

5.2 Configuring Attribute Sources

Note
The following description is for configuring the classic, static attribute sources. However everything written here applies also to configuration of the dynamic sources: the only difference is that instead of `use.security.attributes.` property prefix, the `use.security.dynamicAttributes.` should be used.

Note
The full list of options related to attribute sources is available here: Section 2.8.2.

To configure the static attribute sources, the `use.security.attributes.order` property in the configuration 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, `use.security.attributes.combiningPolicy`, allows you to control how attributes from different sources are combined.

For example, the following configuration snippet

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

#
# Combining policy
#
# MERGE_LAST_OVERRIDES (default), FIRST_APPLICABLE,  ↩
   FIRST_ACCESSIBLE or  MERGE
use.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 multiple attribute sources can 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:

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

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

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

5.3 Available attribute sources

5.3.1 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 can be protected by a client-authenticated SSL

- XUUDB can store static mappings of grid users: the local `xlogin`, `role` and `project` attributes (where `project` maps to Unix groups)
- XUUDB since version 2 can also assign attributes in a dynamic way, e.g. from pool accounts.
- 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.

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

To enable and configure the XUUDB as a static attribute source, set the following properties in the configuration file:

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

To enable and configure the XUUDB as a dynamic attribute source, set the following properties in the configuration file:

```
use.security.dynamicAttributes.order=... XUUDB ...
use.security.dynamicAttributes.XUUDB.class=eu.unicore.uas.security. ↩
    xuudb.XUUDBDynamicAttributeSource
use.security.dynamicAttributes.XUUDB.xuudbHost=https://<xuudbhost>
use.security.dynamicAttributes.XUUDB.xuudbPort=<xuudbport>
```

5.3.2 SAML Virtual Organizations aware attribute source (e.g. UVOS)

UNICORE supports SAML attributes, which can be either fetched by the server or pushed by the clients, using a Virtual Organisations aware attribute source. In the most cases the UNICORE Virtual Organisation Service (UVOS) is deployed as a server providing attributes and handling VOs, as it supports all UNICORE features and therefore offers a greatest flexibility, while being simple to adopt. SAML attributes can be used only as a static attribute source.

The SAML attribute source is described in a separate section: [?].

5.3.3 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 (only static mappings are possible). 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

```
use.security.attributes.order=... FILE ...
use.security.attributes.FILE.class=eu.unicore.uas.security.file. ↩
    FileAttributeSource
use.security.attributes.FILE.file=<your map file>
use.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` (with values `true` or `false`)
- `queue`

Attributes with those names (case insensitive) are handled as special UNICORE incarnation attributes. The correspondence should be straightforward, e.g. the `xlogin` is used to provide available local OS user names for the client.

For all attributes except of the `supplementaryGroups` the default value is the first one provided. For `supplementaryGroups` the default value contains all defined values.

You can also define other attributes - those will be used as XACML authorization attributes, with XACML string type.

5.3.4 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, by assigning them the *banned* role).

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

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

# The UVOS_CLUSTER is a sub chain:
use.security.attributes.UVOS_CLUSTER.class=de.fzj.unicore.uas. ↩
    security.util.AttributeSourcesChain
use.security.attributes.UVOS_CLUSTER.order=UVOS1 UVOS2
use.security.attributes.UVOS_CLUSTER.combiningPolicy= ↩
    FIRST_ACCESSIBLE

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

6 The UNICORE persistence layer

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

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

depending on the services that are running in the container.

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, JobManagement) stores its information in a separate database table (having the same name as the service, e.g. "JobManagement").

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. Using MySQL has the additional benefit that the server starts up much faster.

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.

6.1 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 job (XNJS) data and other service data. If the "persistence.config" property is set, the given file will be read as a Java properties file, and the properties will be used.

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.

Property name	Type	Default value / mandatory	Description
`persistence.cache.enable[.*]`	[true, false] *can have subkeys*	`true`	Enable caching.
`persistence.cache.maxSize[.*]`	integer number *can have subkeys*	`10`	Maximum number of elements in the cache (default: 10).
`persistence.class[.*]`	string *can have subkeys*	`de.fzj.unicore.persist.impl.H2Persist`	The persistence implementation class, which controls with DB backend is used.
`persistence.cluster.config[.*]`	string *can have subkeys*	-	Clustering configuration file.
`persistence.cluster.enable[.*]`	[true, false] *can have subkeys*	`false`	Enable clustering mode.
`persistence.config`	filesystem path	-	Allows to specify a separate properties file containing the persistence configuration.
`persistence.database[.*]`	string *can have subkeys*	-	The name of the database to connect to (e.g. when using MySQL).
`persistence.directory[.*]`	string *can have subkeys*	-	The directory for storing data (embedded DBs).
`persistence.driver[.*]`	string *can have subkeys*	-	The database driver. If not set, the default one for the chosen DB backend is used.
`persistence.h2.cache_size[.*]`	integer number *can have subkeys*	`4096`	(H2) Cache size.
`persistence.h2.options[.*]`	string *can have subkeys*	-	(H2) Further options separated by *;*.
`persistence.h2.server_mode[.*]`	[true, false] *can have subkeys*	`false`	(H2) Connect to a H2 server.
`persistence.host[.*]`	string *can have subkeys*	`localhost`	The database host.

Property name	Type	Default value / mandatory	Description
persistence.index.basedir[.*]	string *can have subkeys*	/tmp	The base directory for storing the Lucene index files.
persistence.index.enable[.*]	[true, false] *can have subkeys*	false	Whether to enable the Lucene indexer.
persistence.max_connections[.*]	integer number *can have subkeys*	1	Connection pool maximum size.
persistence.mysql.tabletype[.*]	string *can have subkeys*	MyISAM	(MySQL) Table type (engine) to use.
persistence.password[.*]	string *can have subkeys*	*empty string*	The database password.
persistence.pool_timeout[.*]	integer number *can have subkeys*	3600	Connection pool timeout when trying to get a connection.
persistence.port[.*]	integer number *can have subkeys*	3306	The database port.
persistence.store_data_as_binary[.*]	[true, false] *can have subkeys*	false	Whether data is stored as a binary BLOB object (if not using JSON format).
persistence.store_data_as_json[.*]	[true, false] *can have subkeys*	true	Whether data is stored as a JSON String.
persistence.user[.*]	string *can have subkeys*	sa	The database username.
persistence.version[.*]	integer number *can have subkeys*	1	Version of the stored data.

6.1.1 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.

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

```
persistence.cache.maxSize.JOBS=1000
```

6.1.2 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

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>
```

6.1.3 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>
persistence.mysql.tabletype=MyISAM
```

6.2 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.

7 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 `coreServices.targetsystemfactory.xnjs.configfile`.

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

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

Table 4: 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.

7.1 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

7.1.1 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*

7.1.2 Required TSI Configuration

Configuration is done by editing <tsi_dir>/conf/tsi.properties At least 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
```

7.1.3 UNICORE/X configuration

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

```
coreServices.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"/>
```

7.1.4 Additional parameters

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

Table 5: 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

7.1.5 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. See Section 6 for details

---

7.1.6 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, set the following property to *false* (by default it is set to true):

```
    <!-- enable SSL -->
    <eng:Property name="CLASSICTSI.ssl.disable" value="false"/>
```

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.

7.2 Support for the UNICORE RUS Accounting

XNJS can produce accounting data and send it (using JMS messaging) to the UNICORE RUS Accounting which is a sophisticated and production ready system. The rus-job-processor module from this system is included in the Unicore/X release. Note that this system is supposed to work only when the classic (Perl) TSI is deployed.

Additionally to set up the whole UNICORE RUS Accounting, at least two additional components are needed to be installed (rus-service with a records database and rus-bssadapter that collects resource usage data from LRMS).

Further information on the RUS Accounting system is available in its documentation. Configuration of the rus-job-processor is available in this documentation too, in the respective section.

Other components of the RUS Accounting system can be downloaded from the UNICORE Life project, files section.

8 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.

8.1 Defining the IDB file

The IDB file is defined by the property "XNJS.idbfile", which must point to a file on the UNICORE/X machine which is readable by the UNICORE/X process. For security reasons, it should NOT be writable.

8.2 Using an IDB directory

While the IDB can be put into a single file, it is often convenient to use multiple files. In this case, the property "XNJS.idbfile" points to a directory. This directory should contain

- a single, mandatory, "main" IDB file
- optionally, multiple XML files containing application definitions (see below)
- optionally, multiple XML files containing execution environment definitions (see Section 8.8)

The main IDB file consists of an "IDB" XML element:

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

while application files use the Application element

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

and the execution environment files look like this:

```
<ee:ExecutionEnvironment xmlns:ee="http://www.unicore.eu/unicore/ ↩
    jsdl-extensions">
 ...
</ee:ExecutionEnvironment>
```

8.3 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.

8.3.1 Simple applications

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

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

8.3.2 Arguments

Command line arguments are specified using <Argument> tags:

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

8.3.3 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.

8.3.4 More

For more details about IDB application definitions, please consult Section 8.6.

8.4 TargetSystemProperties

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

8.4.1 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.

8.4.2 Resources

Resources of a target system are specified using the Resource tag defined in the JSDL specification (see http://www.gridforum.org/documents/GFD.56.pdf). 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.

8.4.3 "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.

8.4.4 CPU Architecture

JSDL allows to advertise the CPU architecture.

```
      <jsdl:CPUArchitecture>
        <jsdl:CPUArchitectureName>x86</jsdl:CPUArchitectureName>
      </jsdl:CPUArchitecture>
```

Due to restrictions imposed by the JSDL standard, the valid values for the CPUArchitecture-Name element are limited to a fixed list, some useful values are "x86", "x86_64", "sparc", "powerpc", and "other". For the full list please consult the JSDL standard.

8.4.5 Operating system

JSDL allows to advertise the operating system that the site runs.

```
<!-- O/S -->
<jsdl:OperatingSystem>
   <jsdl:OperatingSystemType>
      <jsdl:OperatingSystemName>LINUX</jsdl:OperatingSystemName>
   </jsdl:OperatingSystemType>
   <jsdl:OperatingSystemVersion>2.6.13</jsdl: ↩
       OperatingSystemVersion>
   <jsdl:Description>Ubuntu Linux</jsdl:Description>
</jsdl:OperatingSystem>
```

Due to restrictions imposed by the JSDL standard, the valid values for the OperatingSystem-Name element are limited to a fixed list, some useful values are "LINUX", "SOLARIS", "AIX", "MACOS", "WIN_NT", "WINDOWS_XP", "FREE_BSD" and "UNKNOWN". For the full list please consult the JSDL standard.

8.4.6 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 Section 8.10.

8.4.7 File systems

File systems such as SCRATCH can be defined in the IDB as well, for example

```
  <idb:TargetSystemProperties>

    <!-- SCRATCH file system -->
    <idb:Filesystem Name="SCRATCH" IncarnatedPath="/work/$USER" />

  </idb:TargetSystemProperties>
```

The job's environment will then contain a variable

```
SCRATCH="/work/$USER" ; export SCRATCH
```

JSDL data staging elements can contain the FileSystemName tag to indicate that the file should NOT be staged into the job working directory, but into the named file system.

8.4.8 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>
```

8.5 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><![CDATA[
#!/bin/sh
#COMMAND
#RESOURCES
#SCRIPT
]]>
</idb:SubmitScriptTemplate>

<idb:ExecuteScriptTemplate><![CDATA[
#!/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><![CDATA[
#!/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>
```

Note
Make sure that the commands added to the ExecuteScriptTGemplate DO NOT generate any output on standard out or standard error! Always redirect any output to /dev/null!

8.5.1 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
```

8.6 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.

8.6.1 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.

8.6.2 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.

8.7 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.

8.7.1 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.

8.7.2 Per-application node requirements

When an application requires a special hardware or software which is available only on a subset of cluster nodes, node filtering must be applied. It is typically solved by marking the nodes with a special label, named node property. In application description, the required node properties might be added as follows:

```
<idb:IDBApplication>
  <idb:ApplicationName>SOME_MPI_APP</idb:ApplicationName>
  <idb:ApplicationVersion>1.0</idb:ApplicationVersion>
  <jsdl:POSIXApplication xmlns:jsdl="http://schemas.ggf.org/jsdl ↩
      /2005/11/jsdl-posix">
    <jsdl:Executable>/opt/some_mpi_app/binary</jsdl:Executable>
  </jsdl:POSIXApplication>
  <!-- other elements as pre/post commands -->
  <idb:BSSNodesFilter>infiniband</idb:BSSNodesFilter>
  <idb:BSSNodesFilter>gpu</idb:BSSNodesFilter>
 </idb:IDBApplication>
```

With such configuration the SOME_MPI_APP application will be executed only on nodes having both the `infiniband` and `gpu` properties.

Note
Nodes properties will not work with the Java TSI, and need not to be supported for all kinds of batch systems when using the legacy TSI. Please refer to the TSI documentation for details.

8.8 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.

8.9 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 (in the JSDL `Resources` element), 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 runOnLoginNode="false">
      <ee:Name>PRINT_START_TIME</ee:Name>
    </ee:PreCommand>
    <ee:PostCommand runOnLoginNode="false">
      <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`. It has an additional attribute (with values "true" or "false") `runOnLoginNode` which controls whether the precommand is executed on the login node, or whether it is executed on the compute node (i.e. whether it is part of the job script sent to the TSI). By default, the precommand is executed on the login node.
- `PostCommand` : This tag denotes a command that is executed after the actual execution. Its subtags are the same as for `PreCommand`.

8.10 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 in Section 8.11.

8.10.1 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:Resource>

 </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.

Note
The resource name "Project" (i.e. the "TSI_PROJECT" TSI parameter) is mapped to the account parameter of the batch system, for example "-A" in the case of Torque. Note that JSDL has also a parameter allowing the user to set a job's project. It will be also translated to the TSI_PROJECT parameter. The JSDL project is currently an independent feature to the resource-defined "Project" and if IDB defines ranges for the resource "Project", then the JSDL project is not checked against them. If both JSDL project and resource "Project" are received then the one defined as the resource takes the precedence.

8.10.2 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>
```

8.10.3 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.

Note that every name of a custom resource defined in IDB is converted to upper case and spaces are replaced with the underscore character "_".

8.11 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.

8.11.1 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.

8.11.2 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>
```

8.11.3 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 Section 8.12.

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 Section 8.12 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 Section 8.11 for details.

8.11.4 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.
- It is possible (from the version 6.4.2 up) to stop the job processing from the rule's action. To do so with the `groovy` or `groovy-file` action, throw the `de.fzj.unicore.xnjs-.ems.ExecutionException` object from the script. In case of the `script` action, the script must exit with the exit status equal to `10`. The first 1024 bytes of its standard error are used as the message which is included in the ExecutionException. This feature works both for the BEFORE- and AFTER- SCRIPT actions. It is not possible to achieve this with the `spel` action type.

8.11.5 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 `<` and `>` 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>
```

It is possible to fail the job when a site-specific condition is met. E.g. with the groovy script:

```
        <tns:afterScript>
                <tns:rule>
                        <tns:condition>SOME - CONDITION</tns: ↩
                            condition>
                        <tns:action type="groovy">
throw new de.fzj.unicore.xnjs.ems.ExecutionException(de.fzj.unicore ↩
    .xnjs.util.ErrorCode.ERR_EXECUTABLE_FORBIDDEN, "Description for ↩
     the user");
                        </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.

8.12 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.

8.12.1 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"
```

8.12.2 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"})
```

8.12.3 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.

9 The UNICORE metadata service

UNICORE supports metadata management on a per-storage basis. This means, each storage instance (for example, the user's home, or a job working directory) has its own metadata management service instance.

Metadata management is separated into two parts: a front end (which is a web service) and a back end.

The front end service allows the user to manipulate and query metadata, as well as manually trigger the metadata extraction process. The back end is the actual implementation of the metadata management, which is pluggable and can be exchanged by custom implementations. The default implementation has the following properties

- Apache Lucene for indexing,
- Apache Tika for extracting metadata,
- metadata is stored as files directly on the storage resource, in files with a special ".metadata" suffix
- the index files are stored on the UNICORE/X server, in a configurable directory

9.1 Enabling the metadata service

First, UNICORE's service configuration file *<CONF>/wsrflite.xml* needs to be edited and the following service definition added in the *<services>* section:

```
<!-- enable the metadata management service -->
<service name="MetadataManagement" wsrf="true" persistent="true">
 <interface class="de.fzj.unicore.uas.MetadataManagement"/>
 <implementation class="de.fzj.unicore.uas.metadata. ↩
    MetadataManagementHomeImpl"/>
</service>
```

You will also need to define which implementation should be used. This is done via properties, which can be defined either in *<CONF>/wsrflite.xml* or *<CONF>/uas.config*.

In uas.config, set:

```
#
# Metadata manager settings
#

coreServices.metadata.managerClass=eu.unicore.uas.metadata. ↩
   LuceneMetadataManager

#
# use Tika for extracting metadata
```

```
# (if you do not want this, remove this property)
#
coreServices.metadata.parserClass=org.apache.tika.parser. ↩
    AutoDetectParser

#
# Lucene index directory:
#
# Configure a directory on the UNICORE/X machine where index
# files should be placed
#
coreServices.metadata.luceneDirectory=/tmp/data/luceneIndexFiles/
```

9.2 Controlling metadata extraction

If a file named `.unicore_metadata_control` is found in the base directory (i.e. where the crawler starts its crawling process), it is evaluated to decide which files should be included or excluded in the metadata extraction process.

By default, all files are included in the extraction process, except those matching a fixed set of patterns (".svn", and the UNICORE metadata and control files themselves).

The file format is a standard "key=value" properties file. Currently, the following keys are understood

- `exclude` a comma-separated list of string patterns of filenames to exclude
- `include` a comma-separated list of string patterns of filenames to include
- `useDefaultExcludes` if set to "false", the predefined exclude list will NOT be used

The include/exclude patterns may include wildcards *?* and *.

Examples:

To only include pdf and jpg files, you would use

```
include=*.pdf,*.jpg
```

To exclude all doc and ppt files,

```
exclude=*.doc,*.ppt
```

To include all pdf files except those whose name starts with 2011,

```
include=*.pdf
exclude=2011*.pdf
```

10 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 11. You can also review this guide if you want to have a deeper understanding of the authorization process.

10.1 Basic configuration

Note
The full list of options related to PDP is available here: Section 2.8.2.

There are three options which are relevant to all PDPs:

- `use.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.
- `use.security.accesscontrol.pdp` (value: full class name) This property is used to choose which PDP module is being used.
- `use.security.accesscontrol.pdpConfig` (value: file path) This property provides a location of a configuration file of the selected PDP.

10.2 Available PDP modules

10.2.1 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:

```
use.security.accesscontrol.pdpConfig=<CONFIG_DIR>/xacml2.conf
use.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
```

10.2.2 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:

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

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

10.2.3 Remote SAML/XACML 2.0 PDP with Argus PAP

Note
Releases 6.5.x of UNICORE offered an other Argus PDP implementation which allows for off-sourcing authorisation decision to a remote Argus PDP daemon. While this implementation was working, in the Argus policy language it is impossible to express any rules using the resource owner. Therefore creation of a functional policy for UNICORE with Argus is barely possible and this implementation was dropped in UNICORE 6.6.0.

This PDP allows for mixing local policies with policies downloaded from a remote server using SAML protocol for XACML policy query. This protocol is implemented by Argus PAP server Argus PAP. Please note that under the name Argus there is a whole portfolio of services, but for purpose of UNICORE integration Argus PAP is the only one required.

Usage of Argus PAP together with UNICORE policies is useful as Argus PAP allows for a quite easy editing of authorization policies with its Simplified Policy Language. It is less powerful then XACML but allows for performing all the typical tasks like banning selected users or VOs. Also if Argus is used to provide authorization rules for other middleware installed at the site (as gLite or ARC), it might be desirable to have a single place to store site-wide policies.

Unfortunately as Argus policy can not fully take over the UNICORE authorization (see the above note for details), the Argus policy must be combined with the classic UNICORE XACML 2 policy, stored locally.

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:

```
use.security.accesscontrol.pdpConfig=<CONFIG_DIR>/argus.config
use.security.accesscontrol.pdp=eu.unicore.uas.pdp.argus.ArgusPAP
```

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

```
# The directory where XACML 2.0 policy files are stored
#  (both local and downloaded from Argus PAP)
localpdp.directory=conf/xacml2PoliciesWithArgus

# 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.
#  This algorithm will be used to combine the Argus and local policies.
localpdp.combiningAlg=first-applicable

# Address of the Argus PAP server. Typically only the hostname needs to be changed,
#  rarely the port.
argus.pap.serverAddress=https://localhost:8150/pap/services/ProvisioningService

# What is the name of a file to which a downloaded Argus policy is saved.
#  Note that name of this file is very important as it determines policies evaluation order.
#  Here the Argus policy will be evaluated first.
argus.pap.policysetFilename=00argus.xml

# How often (in ms) the Argus PAP should be queried for a new policy
argus.pap.queryInterval=3600000

# What is the Argus query timeout in ms.
argus.pap.queryTimeout=15000

# If Argus PAP is unavailable for that long (in ms) the PDP will black all users
# assuming that the policy is outdated. Use negative value to disable this feature.
argus.pap.deny.timeout=36000000
```

You can use both http and https addresses. In the latter case server's certificate is used to make the connection. Note that all `localpdp.*` settings are the same as in case of the default, local XACML 2.0 PDP.

Using the available configuration options, it is possible to merge Argus policies in many different ways. Here we present a simple pattern, which is good for cases when Argus is used to ban users (it was also applied to the example above):

- Argus policy should be saved to a file which will be evaluated first, e.g. `00argus.xml`
- Default XACML 2.0 policies of UNICORE local PDP should be added to the directory, without any changes.
- The policy combining algorithm should be `first-applicable`
- Argus PAP policies should include a series of deny statements (see Argus documentation for details) and no final permit (or deny) fall-trough rule.

Then Argus policy will be evaluated first. If any banning rule matches the user then it will be denied by the Argus policy. Otherwise it will be non-applicable and the local, default UNICORE policy will be evaluated. Note that if it is problematic for other (non-UNICORE) services using Argus, to remove the final fall-through permit or deny rule, then you can add such rule, but with a proper `resource` statement so it will be applicable only for non-UNICORE components.

Of course it is also possible to creatively design other patterns, when for instance Argus policy is evaluated as a second one.

11 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 in xref:use_policies-11.

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 10 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:

XACML attribute name	XACML category	XACML type	Description
`urn:oasis:names:tc:xacml:1.0:resource:resource-id`	Resource	AnyURI	WS service name
`urn:unicore:wsresource`	Resource	String	Identifier of the WSRF resource instance (if any).
`owner`	Resource	X.500 name	The name of the VO resource owner.
`voMembership-VONAME`	Resource	String	For each VO the accessed resource is a member, there is such attribute with the *VONAME* set to the VO, and with the value specifying allowed access type, using the same action categories as are used for the `actionType` attribute.
`actionType`	Action	String	Action type or category. Currently *read* for read-only operation and *modify* for others.
`urn:oasis:names:tc:xacml:1.0:action:action-id`	Action	String	WS operation name.
`urn:oasis:names:tc:xacml:1.0:subject:subject-id`	Subject	X.500 name	User's DN.
`role`	Subject	String	The user's role.
`consignor`	Subject	X.500 name	Client's (consignor's) DN.
`vo`	Subject	Strings	Bag with all VOs the user is member of (if any).
`selectedVo`	Subject	String	The effective, selected VO (if any).

Note that the above list is valid for the default local XACML 2 and legacy XACML 1.x PDPs. For others the attributes might be different - see the respective documentation.

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.

11.1 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
```

11.2 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 services.

```
<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.

11.3 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>
```

11.4 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.

11.5 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>
```

12 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.

12.1 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
```

12.2 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.

12.2.1 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>
```

12.2.2 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=""/>
```

13 XtreemFS support

XtreemFS is a distributed filesystem (see http://www.xtreemfs.org).

XtreemFS can be mounted locally at more than one UNICORE site, making it desirable to have an optimized way of moving files used in UNICORE jobs into and out of XtreemFS.

To achieve this, UNICORE supports a special URL scheme "xtreemfs://" for data staging (i.e. moving data into the job directory prior to execution, and moveing data out of the job directory after execution).

As an example, in their jobs users can write (using a UCC example):

```
{

  Imports:
  [
   { From: "xtreemfs://CN=test/test.txt", To: "infile", },
  ]

}
```

to have a file staged in from XtreemFS.

13.1 Site setup

At a site that wishes to support XtreemFS, two ways of providing access are possible. If XtreemFS is mounted locally and accessible to the UNICORE TSI, it is required to define the mount point in `CONF/uas.config`:

```
xtreemfs.mountpoint=...
```

In this case, data will simply be copied by the TSI.

If XtreemFS is not mounted locally, it is possible to define the URL of a UNICORE Storage which provides access to XtreemFS

```
xtreemfs.url=https://...
```

In this case, data will be moved using the usual UNICORE file transfer mechanism.

14 SCP support

UNICORE supports file staging in/out using SCP, as defined in the Open Grid Forum's "HPC File staging profile" (GFD.135).

In the JSDL job description, an scp stage in is specified as follows:

```
<?xml version="1.0"?>
<p:JobDefinition xmlns:p="http://schemas.ggf.org/jsdl/2005/11/jsdl"
                 xmlns:jsdl-posix="http://schemas.ggf.org/jsdl ↩
                     /2005/11/jsdl-posix">
  <p:JobDescription>
    <p:Application>
```

```
      <jsdl-posix:POSIXApplication>
        <jsdl-posix:Executable>/bin/ls</jsdl-posix:Executable>
        <jsdl-posix:Argument>-l</jsdl-posix:Argument>
      </jsdl-posix:POSIXApplication>
    </p:Application>
    <p:DataStaging>
      <p:FileName>input</p:FileName>
      <p:CreationFlag>overwrite</p:CreationFlag>
      <p:Source>
        <p:URI>scp://HOST:PORT:filepath</p:URI>
      </p:Source>
      <ac:Credential xmlns:ac="http://schemas.ogf.org/hpcp/2007/11/ac">
        <wsse:UsernameToken xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
          <wsse:Username>***</wsse:Username>
          <wsse:Password>***</wsse:Password>
        </wsse:UsernameToken>
      </ac:Credential>
    </p:DataStaging>
  </p:JobDescription>
</p:JobDefinition>
```

As you can see, username and password required to invoke SCP are embedded into the job description, and the URL schema is "scp://"

14.1 Site setup

At a site that wishes to support SCP, the UNICORE server needs to be configured with the path of an scp wrapper script that can pass the password to scp, if necessary.

If not already pre-configured during installation, you can configure this path manually in the XNJS config file (or simpler in the IDB)

```
    <!-- scp wrapper script -->
    <eng:Property name="scp-wrapper.sh" value="/path/to/scp-wrapper.sh"/>
```

14.2 SCP wrapper script

The TSI 6.4.2 and later includes a script written in Perl (scp-wrapper.pl), depending on how you installed UNICORE it is probably already pre-configured for you.

An alternative scp wrapper script written in TCL is provided in the "extras" folder of the core server bundle, for your convenience it is reproduced here. It requires TCL and Expect. You may need to modify the first line depending on how Expect is installed on your system.

```
#!/usr/bin/expect -f

# this is a wrapper around scp
#
# it automates the interaction required to enter the password.
#
# Prerequisites:
# The TCL Expect tool is used.
#
# Arguments:
# 1: source, 2: target, 3: password

set source [lindex $argv 0]
set target [lindex $argv 1]
set password [lindex $argv 2]
set timeout 10

# start the scp process
spawn scp "$source" "$target"

# handle the interaction
expect {
   "passphrase" {
     send "$password\r"
     exp_continue
    } "password:" {
     send "$password\r"
     exp_continue
    } "yes/no)?" {
     send "yes\r"
     exp_continue
    } timeout {
      puts "Timeout."
      exit
    } -re "." {
      exp_continue
    } eof {
      exit
    }
}
```

Similar scripts may also be written in other scripting languages such as Perl or Python.

15 Mail support

UNICORE supports file staging out using email. An existing SMPT server or some other working email mechanism is required for this to work.

In the JSDL job description, a stage out using email is specified as follows:

```
<?xml version="1.0"?>
<p:JobDefinition xmlns:p="http://schemas.ggf.org/jsdl/2005/11/jsdl"
                 xmlns:jsdl-posix="http://schemas.ggf.org/jsdl ↩
                     /2005/11/jsdl-posix">
  <p:JobDescription>

    <!-- example stage-out using email -->
    <p:DataStaging>
      <p:FileName>stdout</p:FileName>
      <p:Target>
        <p:URI>mailto:user@domain?subject=Your output is ready</p: ↩
            URI>
      </p:Target>
    </p:DataStaging>

  </p:JobDescription>
</p:JobDefinition>
```

The "mailto" URI consists of the email address and an OPTIONAL user-defined subject.

15.1 Site setup

Without any configuration, UNICORE will use JavaMail and attempt to use an SMTP server running on the UNICORE/X host, expected to be listening on port 25 (the default SMTP port).

To change this behaviour, the following properties can be defined (in the IDB or XNJS config file). See the next section if you do not want to use an SMTP server directly.

- mail.smtp.host: the host of the SMTP server
- mail.smtp.port : the port of the SMTP server
- mail.smtp.user : the user name of the mail account which sends email
- mail.smtp.password : the password of the mail account which sends email
- mail.smtp.ssl : to use SSL, see the XNJS/TSI SSL setup page on how to setup SSL

15.2 Email wrapper script

As an alternative to using JavaMail, the site admin can define a script which is executed (as the current grid user) to send email.

```
    <!-- mailto wrapper script, defining this will disable JavaMail ↩
        -->
    <eng:Property name="mail-wrapper.sh" value="/path/to/mail- ↩
        wrapper.sh"/>
```

This is expected to takes three parameters: email address, file to send and a subject. An example invocation is

```
mail-wrapper.sh "user@somehost.eu" "outfile" "Result file from your ↩
    job"
```

16 EMIR support

The EMI Registry (EMIR) is a new product developed in the EMI project. It is a service registry and can be used from different middlewares such as ARC. UNICORE/X supports publishing service information to EMIR (in addition to the usual UNICORE registries).

To enable publishing to EMIR, the UNICORE/X configuration file `CONF/uas.config` needs to be adapted with the following entries.

```
# enable publishing to EMIR
emiregistry.publishing.enable=true

# set the publishing interval (seconds)
emiregistry.publishing.interval=120

# initialise EMIR publishing at server start
uas.onstartup.99=eu.emi.emir.unicore.PublishingOnStartup

# URL of the EMIR server
emiregistry.server.url=http(s)://<hostname>:<port>
```

17 The CIP (Infoprovider)

The CIP is a component that provides information in GLUE2 format about a UNICORE/X server to interested clients or other services such as CIS or even non-UNICORE components such as BDII.

It consists of two parts: the basic infoprovider maintains information and generates the required GLUE2 document and a web service that can be queried by clients or other services.

To setup CIP, a startup entry in the main config file `CONF/uas.config` is required:

```
# add CIP setup to startup tasks, 'NNN' should be a high-enough  ↩
    integer
container.onstartup.NNN=de.fzj.unicore.cisprovider.impl. ↩
    InitOnStartup
```

There are a number of configuration parameters that control how CIP is set up.

Property name	Type	Default value / mandatory	Description
coreServices.cip.dataPath	filesystem path	conf/site-info.json	Path to the JSON file with constant information about the service.
coreServices.cip.glue2.generate	[true, false]	false	Whether to auto-generate a GLUE2 document
coreServices.cip.glue2.refreshPeriod	integer number	3600	Refresh period (in seconds) for the GLUE2. If <0, no refresh will be done
coreServices.cip.glue2.targetPath	string	/var/run/unicore/unicorex_glue2.xml	Path for the auto-generated GLUE2 document
coreServices.cip.publish	[true, false]	true	Controls whether CIP service endpoint information should be published to registry.
coreServices.cip.site.cpus	integer number	1	Number of CPUs the site provides
coreServices.cip.site.description	string	Linux server	Site description
coreServices.cip.site.latitude	floating point number	50.94545	Geographical latitude of the site
coreServices.cip.site.longitude	floating point number	6.377907	Geographical longitude of the site
coreServices.cip.site.name	string	-	Site name
coreServices.cip.site.url	string	-	Site web URL

18 The Application service (GridBean service)

The Application service (aka GridBean service) provides application specific plugins for the UNICORE Rich client (URC). On the server side, it is good practice to provide such an Application plugin for every configured application whenever applicable.

The availability of the service is advertised through the Registry, if an appropriate "onstartup" setting is used:

```
container.onstartup.11=com.intel.gpe.gridbeans. ↩
    PublishGridBeanService
```

A second property allows to set the directory where GridBean *.jar files are located on the server. These files will be served by the GridBean service.

```
#
# GridBeanService: directory on the UNICORE/X machine where the
# server looks for gridbeans
#
coreServices.gridbean.directory=/usr/local/unicore/gridbeans
```

Only files ending with ".jar" will be served, it is customary to use the suffix "GridBean.jar"