UNICORE/X MANUAL

UNICORE Team

Document Version:	1.0.0
Component Version:	8.2.0
Date:	02 08 2021

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 Java (OpenJDK, Oracle or IBM). We recommend using the latest version of the OpenJDK.

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

UNICORE/X has been developed and most extensively tested on Linux-like systems, but runs on 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 Slurm or SGE, you need to install and configure the UNICORE TSI server.
- to make your resources easily 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

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 systemd on Linux, you would do (as root)

```
systemctl start unicore-unicorex.service
```

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 the central component in a UNICORE system and as such has a number of interfaces to other UNICORE components, as well as many of configuration options. This section gives an overview of what can and should be configured. The detailed configuration guide follows in the next sections.

2.1.1 Mandatory configuration

- SSL certificates and basic security: UNICORE uses SSL certificates for all servers. For UNICORE/X these settings are made in the container.properties config file
- Attribute sources: various ways are available to assign local attributes to users, such as Unix user name, groups and role. For details, consult Section 6.
- Backend / target system access: to access a resource manager like Slurm, the UNICORE TSI needs to be installed and UNICORE/X needs to be configured accordingly. Please consult Section 9.
- You can choose to enable/disable certain UNICORE features, for example if you wish to set up a storage-only UNICORE server. Please refer to Section 2.12.

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

Table 2: UNICORE/X configuration files

config file	**usage**	**dynamically reloaded**
startup.properties	Java process settings (e.g. memory), lib/log/conf directories	no
logging.properties	Logging levels, logfiles and their properties	yes

Table 2: (continued)

config file	usage	dynamically reloaded
uas.config	Main server config file. Defines features, storages, AuthN/AuthZ, AIPs/PDPs	no
container.properties	Server address, SSL settings, Web server settings	no
xnjs.properties	Backend properties for the UNICORE TSI	no
simpleidb	Backend, installed applications, resources	yes
simpleuudb	Maps user DNs to local attributes (optional)	yes
rest-users.txt	Usernames/passwords for REST authentication (optional)	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 Unity as an attribute source (optional)	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 basic settings (like maximum memory), see Section 3 for more on these.

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".

You can use system environment variables within property values, e.g.

```
name=${some_systemvariable}
```

Only use this syntax `${...}` to reference UNICORE/X system variables!

To use UNIX system variables e.g. in storage path definitions use the syntax `$VARIABLE`, i.e. WITHOUT curly braces.

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 `container.properties`. 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	-	(deprecated, use *container.externalurl*) Server URL as visible from the outside, usually the gateway's address, including *<sitename>/services*
`container.client.[string]`	string *can have subkeys*	-	Properties with this prefix are used to configure clients created by the container. See separate documentation for details.
`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.externalurl`	string	-	Server URL as visible from the outside, usually the gateway's address, including *<sitename>*
`container.feature.[string]`	string *can have subkeys*	-	Properties with this prefix are used to configure the deployed features. See separate documentation for details.
`container.host`	string	`localhost`	Server interface to listen on.
`container.httpServer.[string]`	string *can have subkeys*	-	Properties with this prefix are used to configure container's Jetty HTTP server. See separate documentation for details.
`container.messageLogging.[string]`	[true, false] *can have subkeys*	`false`	Append service name and set to *true* to enable message logging for that service.

Property name	Type	Default value / mandatory	Description
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.onstartuptest	[true, false]	true	Controls whether to run tests of connections to external services on startup.
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.resources.scheduled.idletime	integer number	10000	The timeout in millis for removing idle threads.
container.resources.scheduled.maxsize	integer number	32	The maximum thread pool size for the scheduled execution service
container.resources.scheduled.minsize	integer number	0	The minimum thread pool size for the scheduled execution service
container.resources.threads.idletime	integer number	60000	Timeout in millis for removing idle threads.
container.resources.threads.size	integer >=1	1	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 federation.

Property name	Type	Default value / mandatory	Description
container.wsrf.expiredelaylimit	integer number *can have subkeys*	120 [.*]	The initial delay for 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.expiredperiod	integer number *can have subkeys*	60 [.*]	The interval for 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.lockingTimeout	integer number *can have subkeys*	10 [.*]	The timeout when attempting to lock 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 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*	-1 [.*]	Maximum lifetime of resources (in seconds). Add dot and service name as a suffix of this property to set a limit per particular service type.
container.wsrf.maximumResourcesPerUser	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	string	de.fzj.unicore.wsrflite.persistence.Persistence	Implementation used to maintain the persistence of resources state.

Property name	Type	Default value / mandatory	Description
container.wsrf.sg.defaultermtime	integer >= 1	1800	The default termination time of service group entries in seconds.

2.6 Integration of UNICORE/X with other parts of 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 hard-coded into CONF/container.properties, using the "container.baseurl" property:

```
container.baseurl=https://Gateway_HOST:Gateway_PORT/SITENAME/ ↩
    services
```

where Gateway_HOST and Gateway_PORT are the host and port of the gateway, and SITENAME is the UNICORE/X site name. The gateway address MUST be accessible from the UNICORE/X node!

On the gateway side, the UNICORE/X address is hard-coded as well, using an entry SITENAME=address in the connections.properties file pointing to the network address of the UNICORE/X container.

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.

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
```

2.6.3 Unity

If you want to support user authentication via Unity, you have to configure UNICORE/X to trust one or more Unity servers. This is done using the `container.security.trustedAssertionIssuers` property. This configures a truststore containing the certificates of all trusted Unity servers (NOT the CA certificates).

For example, to configure a directory containing the trusted certificates in PEM format:

```
# configure trusted Unity certificates
container.security.trustedAssertionIssuers.type=directory
container.security.trustedAssertionIssuers.directoryLocations.1= ↩
    conf/unity/unity.pem
```

All the usual options for configuring truststores are available here, as well, as described in Section .

Note
To enable certificate-less end user access, you will also make sure that the Gateway does not require SSL client-authentication. Please refer to the Gateway manual.

2.7 Startup code

In order to provide a flexible initialization process for UNICORE/X, there is a set of properties 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	Initializes the job management system and the "local" registry; should usually be run on startup	normal UNICORE/X servers

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 UNICORE users. A number of attribute sources exist, 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 6.
- Access control setup (controlling in detail who can do what on which services). Again, several options exist, which are described in Section 16.

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)*
`container.security.accesscontrol.pdp`	Class extending eu.unicore.services.security.pdp.UnicoreXPDP	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.	Controls which Policy
`container.security.accesscontrol.pdpConfig`	filesystem path		Path of the PDP configuration file

Property name	Type	Default value / mandatory	Description
`container.security.additionalServiceIdentities.*`	list of properties with a common prefix	–	List of additional service identifiers (e.g. URLs where this service is accessible) accepted in SAML authentication.
`container.security.attributes[.*]`	string *can have subkeys*	–	Prefix used for configurations of particular attribute sources.
`container.security.attributes.combiner`	string	`MERGE_LAST_WINS`	Which 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.*`	list of properties with a common prefix	*empty*	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.dynamicAttributes[.*]`	string *can have subkeys*	–	Prefix used for configurations of particular dynamic attribute sources.
`container.security.dynamicAttributes.combiner`	string	`MERGE_LAST_WINS`	Which 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.

Property name	Type	Default value / mandatory	Description
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 accept 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	Whether the site should try to autoregister itself with the Gateway. This must be also configured on the Gateway side.
container.security.gateway.registrationSecret	string	-	Required secret when autoregistering with the Gateway. This must match the secret configured on the Gateway side.
container.security.gateway.registrationUpdateInterval	integer >= 10	300	How often the automatic gateway registration should be refreshed.
container.security.gateway.waitOnStartup	[true, false]	false	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).

Property name	Type	Default value / mandatory	Description
`container.security.`	string *can have subkeys*	-	Prefix used to configure REST subsystem security. See separate docs.
`container.security.sessionLifetime`	integer >=1	`28800`	Controls the lifetime of security sessions (in seconds).
`container.security.sessionsEnable`	[true, false]	`true`	Controls whether the server supports security sessions which reduce client/server traffic and load.
`container.security.maxSessionsPerUser`	integer >=1	`5`	Controls the number of security sessions each user can have. If exceeded, some cleanup will be performed.
`container.security.requireSignatures`	[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.
`container.security.samlTrustedAssertionIssuers.`	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 Credential and truststore settings

These properties are used to configure the server's credential (used to make outgoing SSL connections) and truststore. The truststore controls which incoming SSL connections are accepted.

We recommend using a credential in PKCS12 or .pem format, and a directory containing .pem files as truststore.

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*.

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.
`container.security.truststore.directoryEncoding`	[PEM, DER]	`PEM`	For directory truststore controls whether certificates are encoded in PEM or DER. Note that the PEM file can contain arbitrary number of concatenated, PEM-encoded certificates.
`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.

Property name	Type	Default value / mandatory	Description
container.security.truststore.keystorePass	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.opensslNewSigningPolicies	[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.opensslNamespaceMode	[GLOBUS_EUGRIDPMA_FALLBACK, EUGRIDPMA_GLOBUS, GLOBUS, EUGRIDPMA, GLOBUS_EUGRIDPMA_REQUIRE, EUGRIDPMA_GLOBUS_REQUIRE, GLOBUS_REQUIRE, EUGRIDPMA_REQUIRE, EUGRIDPMA_AND_GLOBUS, EUGRIDPMA_AND_GLOBUS_REQUIRE, IGNORE]	GLOBUS_EUGRIDPMA_FALLBACK	In case of openssl truststore, controls which (and in which order) namespace checking rules should be applied. The *REQUIRE* settings will cause that all configured namespace definitions files must be present for each trusted CA certificate (otherwise checking will fail). The *AND* settings will cause to check both existing namespace files. Otherwise the first found is checked (in the order defined by the property).
container.security.truststore.opensslCertificateDir	filesystem path	/etc/grid-security/certificates	Directory to be used for opeenssl truststore.
--- *Revocation settings* ---			
container.security.crl.connectionTimeout	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.crlDiskCacheDirectory`	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.*`	list of properties with a common prefix	-	Optional list of local OCSP responders

Property name	Type	Default value / mandatory	Description
container.security.[REQUIRE, IF_AVAILABLE, IGNORE]	[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.integer number	integer number	10000	Timeout for OCSP connections in miliseconds.
container.security.[CRL, OCSP, OCSP_CRL]	[CRL, OCSP, OCSP_CRL]	OCSP_CRL	Controls overal revocation sources order
container.security.[true, false]	[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.

2.9 Configuring the execution backend (XNJS and TSI)

Information on the configuration of the XNJS and TSI backend can be found in Section 9.

2.10 Configuring storage services

Information on the configuration of the storage factory service, shared storages and per-user storages attached to target systems can be found in Section 13.

2.11 HTTP proxy, timeout and web server settings

A number of settings exist that control the the web server and the HTTPClient library used for outgoing HTTP(s) calls.

The HTTP server options are shown in the following table.

Property name	Type	Default value / mandatory	Description
container.httpServer.CORS_allowedHeaders	string	*	CORS: comma separated list of allowed HTTP headers (default: any)
container.httpServer.CORS_allowedMethods	string	GET,PUT,POST,DELETE,HEAD	CORS: comma separated list of allowed HTTP verbs.

Property name	Type	Default value / mandatory	Description
container.httpServer.CORS_allowedOrigins	string	-	CORS: allowed script origins.
container.httpServer.CORS_chainPreflight	[true, false]	Preflight	CORS: whether preflight OPTION requests are chained (passed on) to the resource or handled via the CORS filter.
container.httpServer.CORS_exposedHeaders	string	Location, Content-Type	CORS: comma separated list of HTTP headers that are allowed to be exposed to the client.
container.httpServer.disabledCipherSuites	string	*cipher string*	Space separated list of SSL cipher suites to be disabled. Names of the ciphers must adhere to the standard Java cipher names, available here: http://docs.oracle.com/-javase/8/docs/technotes/-guides/security/-SunProviders.html#SupportedCipherSuites
container.httpServer.enableCORS	[true, false]	false	Control whether Cross-Origin Resource Sharing is enabled. Enable to allow e.g. accesing REST services from client-side JavaScript.
container.httpServer.enableHsts	[true, false]	false	Control whether HTTP strict transport security is enabled. It is a good and strongly suggested security mechanism for all production sites. At the same time it can not be used with self-signed or not issued by a generally trusted CA server certificates, as with HSTS a user can't opt in to enter such site.

Property name	Type	Default value / mandatory	Description
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.gzipEnable	[true, false]	false	Controls whether to enable compression of HTTP responses.
container.httpServer.gzipMinSize	integer number	100	Specifies the minimal size of message that should be compressed.
container.httpServer.maxConnections	integer number	200	deprecated
container.httpServer.maxIdleTime	integer >= 1	100	deprecated
container.httpServer.maxRequestConnections	integer >= 0	0	Maximum number of incoming connections to this server. If set to a value larger than 0, incoming connections will be limited to that number. Default is 0 = unlimited.
container.httpServer.maxRequestIdleTime	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 number	255	Maximum number of threads to have in the thread pool for processing HTTP connections. Note that this number will be increased with few additional threads to handle connectors.
container.httpServer.minThreads	integer >= 1	1	Minimum number of threads to have in the thread pool for processing HTTP connections. Note that this number will be increased with few additional threads to handle connectors.
container.httpServer.requireClientAuthn	[true, false]	true	Controls whether the SSL socket requires client-side authentication.

Property name	Type	Default value / mandatory	Description
`container.httpServer.clientAuthn`	[true, false]	`true`	Controls whether the SSL socket accepts (but does not require) client-side authentication.
`container.httpServer.xFrameAllowed`	string	`http://localhost`	URI origin that is allowed to embed web interface inside a (i)frame. Meaningful only if the xFrameOptions is set to *allowFrom*. The value should be in the form: *http[s]://host[:port]*
`container.httpServer.xFrameOptions`	[deny, sameOrigin, allowFrom, allow]	`deny`	Defines whether a clickjacking prevention should be turned on, by insertionof the X-Frame-Options HTTP header. The *allow* value disables the feature. See the RFC 7034 for details. Note that for the *allowFrom* you should define also the xFrameAllowed option and it is not fully supported by all the browsers.

The HTTP client options are the following

Property name	Type	Default value / mandatory	Description
`container.client.disableSigningEnabled`	[true, false]	`false`	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).

Property name	Type	Default value / mandatory	Description
container.client.incoming.handlers	string	*empty string*	Space separated list of additional handler class names for handling incoming WS messages
container.client.maximum.number.retries	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.message.logging	[true, false]	false	Controls whether messages should be logged (at INFO level).
container.client.outgoing.handlers	string	*empty string*	Space separated list of additional handler class names for handling outgoing WS messages
container.client.security.sessions	[true, false]	true	Controls whether security sessions should be enabled.
container.client.server.hostname.check	[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.ssl.authn.enabled	[true, false]	true	Controls whether SSL authentication of the client should be performed.
container.client.wsrf.retry.delay	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.

Property name	Type	Default value / mandatory	Description
container.client.http.connectionClose	[true, false]	false	If set to true then the client will send connection close header, so the server will close the socket.
container.client.http.connectionTimeout	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.soTimeout	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.

2.12 Features provided by UNICORE/X

The functionality of the UNICORE/X server is organised into "features", where each feature can combine services, startup code and the like.

Features are enabled by default.

Features can be disabled via configuration. It is also possible to disable single services in a feature.

2.12.1 JobManagement

This feature deals with job submission and management, as well as those storage services required for job processing.

To disable the whole feature

```
container.feature.JobManagement.enable=false
```

Table 4: UNICORE/X JobManagement feature

Service name	**usage**
TargetSystemFactoryService	High level compute service
TargetSystemService	Per-user compute service instances
JobManagement	Per job service instance
ReservationManagement	Make and edit reservations
StorageManagement	Access to storages
ServerServerFileTransfer	Server-server file transfers
ClientServerFileTransfer	Data upload/download

2.12.2 StorageAccess

This feature provides storage access, storage factory service, metadata management and file transfers.

Table 5: UNICORE/X StorageAccess feature

Service name	**usage**
StorageManagement	Access to storages
StorageFactory	Dynamically create new storage endpoints
MetadataManagement	Metadata service
ServerServerFileTransfer	Server-server file transfers
ClientServerFileTransfer	Data upload/download

To disable the whole feature

```
container.feature.StorageAccess.enable=false
```

To disable only one service, e.g. the Storage Factory

```
container.feature.StorageAccess.StorageFactory.enable=false
```

2.12.3 Base

This feature provides low-level services, but also contains the RESTful APIs for jobs and data management.

Table 6: UNICORE/X Base feature

Service name	**usage**
core	RESTful APIs for jobs and data
Enumeration	SOAP/XML service for long lists (jobs, ...)
Task	SOAP/XML service for async tasks (metadata extraction)

2.12.4 Admin

This feature provides the Admin service (see Section 3.3.2)

Table 7: UNICORE/X Admin feature

Service name	**usage**
admin	RESTful API to the admin service
AdminService	SOAP/XML API to the admin service

2.12.5 Registry

This feature provides the Registry service. This covers both the "internal" version running in every UNICORE/X server, as well as the shared Registry that is used to store information about

multiple UNICORE servers.

A setting

```
container.feature.Registry.mode=shared
```

will enable "shared" mode. Don't do this on a "normal" UNICORE/X server.

Table 8: UNICORE/X Registry feature

Service name	**usage**
registries	RESTful API to the Registry service
Registry	Registry service and SOAP/XML API
ServiceGroupEntry	Registry entries service and SOAP/XML API

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/container.properties` and add properties, e.g.

```
container.wsrf.maxInstancesPerUser.JobManagement=200
container.wsrf.maxInstancesPerUser.FileTransfer=20
```

The last part of the property name is the service name, see Section 2.12 for the services in UNICORE/X.

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/2 logging framework. (http://logging.apache.org/log4j/2.x/manual/-configuration.html). The config file is specified with a Java property `log4j.configurationFile`.

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.

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

```
%d [%X{clientName}] [%X{jobID}] [%t] %-5p %c{1} %x - %m%n
```

For more info on controlling the logging we refer to the log4j/2 documentation:

https://logging.apache.org/log4j/2.x/manual/configuration.html

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.

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)
uftp	UFTP client/server communication
org.apache.cxf	Web service toolkit (Apache CXF)

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.

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) for administrative and operations tasks.

To do this you need to configure UCC with administrative privileges. One way is to add the "admin" role to your user account, and select this role when running UCC commands

```
ucc .... -Z role:admin
```

or create a dedicated admin user.

Another way to do this is using the *server* certificate of the UNICORE/X server, which will give UCC administrator rights provided UNICORE/X is configured to accept X509 authentication.

```
# use UNICORE/X keystore
authenticationMethod=X509
credential.path=/path/to/unicorex/keystore
credential.password=...

# (optional) truststore config omitted
```

Also you should connect directly to UNICORE/X, not to the registry as usual. Say your UNICORE/X server is running on *myhost* on port 7777, your preferences file would look like this

```
registry=https://myhost:7777/rest/registries/default_registry
```

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 *rest* commands. Using *list-jobs* you can search for jobs with given properties, whereas the *rest* 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 *rest* command can be used to destroy resources, or look at their properties. Please see "ucc rest -h" for details.

Try

```
ucc rest get https://myhost:7777/rest/core/factories/ ↩
    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
```

To have a look at the internal information about a user job, try

```
ucc admin-runcommand ShowJobDetails jobID=......
```

where `jobID` is the unique ID of the job.

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/container.properties 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 TSI server, 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 first by the SSL layer. In general messages will be relegated through the Gateway, and will not be directly from end user clients.
- extract authentication information from the HTTP headers, such as username/password, OAuth token, a JWT delegation token or even X509 certificate information
- authenticate the message using the configured authentication handlers. This procedure will assign a X500 distinguished name to the current user, which in UNICORE terms is the user identity.
- extract further information used for authorisation from the message sent to the server. This information may include: originator of the message(in case the message passed through a UNICORE gateway), trust delegation tokens, incoming VO membership assertions, etc.
- 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 widely configured.

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, (see Section 2).

A user request is assigned an identity during the authentication process. Identities are X.500 distinguished names. Requests without authentication are *anonymous* and are usually limited to informational endpoints.

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 for the current request.

4.1.3 Resource ownership

Each service is *owned* by some entity identified by an X.500 distinguished name. 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

Messages can be sent from other servers on behalf of an end user. The server will "prove" this by using a JWT token for authentication, which contains the target user's identity (X500 name), and which is signed by the sending server. The receiving server can check the signature with the sender's public key, which will generally be read from the shared registry.

4.1.5 Attributes

UNICORE/X retrieves user attributes using either a local component or a remote service. For example, an XUUDB attribute service can be configured. See Section 6 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 + instance id)
- the activity to be performed (the web method such as GET)

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 16 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.

5 Authentication

5.1 Introduction

UNICORE's RESTful APIs require configuration of the mechanisms for end user authentication, which will check the supplied credentials and map the user to a distinguished name (DN).

This configuration is done in the container config file (typically `uas.config` or `container.properties`).

The enabled authentication options and their order are configured using a list of enabled mechanisms. For example

```
container.security.rest.authentication.order=FILE UNITY-OAUTH X509
```

As you can see, you can use one or more authentication methods, UNICORE will try all configured authentication options in order.

For each enabled option, a set of additional properties is used to configure the details (for example the Unity address)

5.2 Username-password file

The FILE mechanism uses a map file containing username, password and the DN. Required configuration is the location of the file.

```
container.security.rest.authentication.FILE.class=eu.unicore. ↩
    services.rest.security.FilebasedAuthenticator
container.security.rest.authentication.FILE.file=conf/rest-users. ↩
    txt
```

The file format is

```
#
# on each line:
# username:hash:salt:DN
#
demouser:<...>:<...>:CN=Demo User, O=UNICORE, C=EU
```

i.e. each line gives the username, the hashed password, the salt and the user's DN, separated by colons. To generate entries, i.e. to hash the password correctly, the *md5sum* utility can be used. For example, if your intended password is *test123*, you could do

```
$> SALT=$(tr -dc "A-Za-z0-9_$&!=+#" < /dev/urandom | head -c 16 |  ↩
    xargs)
$> echo "Salt is ${SALT}"
$> echo -n "${SALT}test123" | md5sum
```

which will output the salted and hashed password. Here we generate a random string as the salt. Enter these together with the username, and the DN of the user into the password file.

5.3 Unity authentication using OAuth2 Bearer token

This mechanism uses the OAuth2 token sent from the client (HTTP "Authorization: Bearer . . . " header) to authenticate to Unity. In Unity terms, this uses the endpoint of type "SAMLUnicore-SoapIdP" with authenticator of type "oauth-rp with cxf-oauth-bearer".

```
container.security.rest.authentication.UNITY-OAUTH.class=eu.unicore ↩
    .services.rest.security.UnityOAuthAuthenticator
container.security.rest.authentication.UNITY-OAUTH.address=https:// ↩
    localhost:2443/unicore-soapidp-oidc/saml2unicoreidp-soap/ ↩
    AuthenticationService
# validate the received assertions?
container.security.rest.authentication.UNITY-OAUTH.validate=true
```

UNICORE must be configured to trust the assertions issued by the Unity server, please refer to the relevant section on trusted assertion issuers in the manual.

5.4 Unity authentication using username/password

This mechanism takes the username/password sent from the client (HTTP Basic auth) and uses this to authenticate to Unity, retrieving an authentication assertion.

```
container.security.rest.authentication.UNITY.class=eu.unicore. ↩
    services.rest.security.UnitySAMLAuthenticator
container.security.rest.authentication.UNITY.address=https:// ↩
    localhost:2443/unicore-soapidp/saml2unicoreidp-soap/ ↩
    AuthenticationService
# validate the received assertions?
container.security.rest.authentication.UNITY.validate=true
```

UNICORE must be configured to trust the assertions issued by the Unity server, please refer to the relevant section on trusted assertion issuers in the manual.

5.5 X.509 certificate

UNICORE supports X.509 client certificates for authentication.

```
container.security.rest.authentication.order= ... X509 ...

container.security.rest.authentication.X509.class=eu.unicore. ↩
    services.rest.security.X509Authenticator
```

5.6 PAM

This authentication module allows to authenticate users with the username and password that they have on the UNICORE/X system.

```
container.security.rest.authentication.order= ... PAM ...

container.security.rest.authentication.X509.class=eu.unicore. ↩
    services.rest.security.PAMAuthenticator
container.security.rest.authentication.X509.DNTemplate=CN=%s, OU= ↩
    pam-local-users
```

The parameter "DNTemplate" is used to define which DN will be assigned to authenticated users, where the "%s" will be replaced by the user name. In the example above, user "test-user" will have the DN "CN=test-user, OU=pam-local-users".

There is also a PAM attribute source that you can use to automatically assign role="user" as well the Unix login and groups correctly for authenticated users.

```
container.security.attributes.order= ... PAM ...
container.security.attributes.PAM.class=eu.unicore.services.rest. ↩
    security.PAMAttributeSource
```

5.7 OAuth2 authentication using a Bearer token

It is also possible in principle to directly authenticate to an OAuth2 server, contact unicore-support for details.

5.8 Configuring JWT Delegation

Beginning with UNICORE 8.0.0, delegation is fully supported for REST services. The delegating server creates a JWT token containing user authentication information and signs it with its private key. The receiving server can check the signature using the sender's public key.

Public keys are distributed via the shared service Registry.

The lifetime of the issued tokens is 300 seconds by default, which can be changed via

```
container.security.rest.jwt.lifetime=300
```

For very simple cases, e.g. when no shared registry is used, a shared hmac secret can be configured as well. The length of the secret must be at least 32 characters

```
container.security.rest.jwt.hmacSecret=....
```

This secret must be the same on all the UNICORE servers that are supposed to trust each other.

6 Attribute sources

The authorization process in UNICORE/X requires that each UNICORE user (identified by an X.500 DN) is assigned some *attributes* such as her *role*. Attributes are also used to subsequently run tasks for 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 UNICORE 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.

6.1 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 Unity/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.
- 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 federated 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 which local user id (in UNIX called *uid*) should be assigned to the UNICORE user.
- *group* - specifies the primary group (primary gid) that the UNICORE user should get.
- *supplementaryGroups* - specifies all supplementary groups the UNICORE user should get.
- *addDefaultGroups* - boolean attribute saying whether groups assigned to the Xlogin (i.e. the local uid of the UNICORE user) in the operating system should be additionally added for the UNICORE user.
- *queue* - define which BSS queues are allowed for the particular user.

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.

6.2 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 `container.security.attributes.` property prefix, the `container.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 `container.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, `container.security.attributes.combiningPolicy`, allows you to control how attributes from different sources are combined.

For example, the following configuration snippet

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

#
# Combining policy
#
# MERGE_LAST_OVERRIDES (default), FIRST_APPLICABLE,  ↩
    FIRST_ACCESSIBLE or  MERGE
container.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:

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

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

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

6.3 Available attribute sources

6.3.1 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 UNICORE site running multiple service containers.
- Access can be protected by a client-authenticated SSL
- XUUDB can store static mappings of UNICORE 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 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:

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

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

6.3.2 SAML Virtual Organizations aware attribute source (e.g. Unity)

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 Unity 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: Section 7.

6.3.3 File attribute source

This attribute source uses a single map file 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 like the XUUDB, or when you want to locally override attributes for selected users (e.g. to ban somebody).

In contrast to the XUUDB, the File attribute source can store all types of attributes, while the XUUDB only handles role, uid and group.

To use, set

```
container.security.attributes.order=... FILE ...
container.security.attributes.FILE.class=eu.unicore.uas.security. ↩
    file.FileAttributeSource
container.security.attributes.FILE.file=<your map file>
container.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.

6.3.4 PAM

This is a special attribute source which only works in conjunction with the corresponding REST authentication module.

```
container.security.attributes.order=... PAM ...
container.security.attributes.PAM.class=eu.unicore.services.rest. ↩
    security.PAMAttributeSource
```

6.3.5 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 Unity 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:
container.security.attributes.order=UNITY_CLUSTER FILE
container.security.attributes.combiningPolicy=MERGE_LAST_OVERRIDES

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

# The UNITY_CLUSTER is a sub chain:
container.security.attributes.UNITY_CLUSTER.class=de.fzj.unicore. ↩
    uas.security.util.AttributeSourcesChain
container.security.attributes.UNITY_CLUSTER.order=UNITY1 UNITY2
container.security.attributes.UNITY_CLUSTER.combiningPolicy= ↩
    FIRST_ACCESSIBLE

# And configuration of the two real sources used in the sub chain:
container.security.attributes.UNITY1.class=...
...
container.security.attributes.UNITY2.class=...
...
```

7 Virtual Organisations (VO) Support

VO (Virtual Organisation) is a quite broad concept. VO server software (such as Unity) is used to store identities of federated entities along with their attributes. Entities are managed with the usage of groups to help administration. Those attributes can be used e.g. for authorization purposes. It is described here how to take advantage of this approach in any service based on the UNICORE Services Environment such as UNICORE/X, Workflow Service, etc.

In the following we use Unity as our VO service, though in principle other SAML servers can be used.

7.1 Overview

7.1.1 Features

All the below features can be used in any combinations, independently:

- Unity can provide all user attributes to by used for authorization and for accessing resources, also those which are unsupported by the more simple attribute sources (including full support for default and allowed attributes). Therefore it can be used as a central attribute source for multiple sites. Since attributes can be assigned in a group scope, it is possible to use a central service with mappings, still having some of the values (for instance Unix user IDs) which are different for each site. It is simple to assign same attribute for groups of users.
- It is possible to assign non-standard attributes and use them for authorization or for quality of service purposes
- As it is possible (as always in UNICORE) to mix attributes from multiple attribute sources, Unity can provide federation-wide settings (for example, the UNICORE role), while local settings (like Unix gids or uids) are assigned locally by particular sites. This is especially useful when using a dynamic attribute source as a complementary one to the static attribute source: Unity provides federation-wide authorization attributes (such as role) and dynamic source assigns local uids/gids.

The system works in **PULL mode**, i.e. as an attribute source. Attributes are pulled (fetched) by the module from a VO service specified in a configuration file when a new request arrives. This mode is transparent for clients.

7.1.2 VO selection

Some of the VO features (such as authorization), require only information about all VOs the user is a member of and associated attributes. However in many cases it is required to assign user's request to a particular VO and to execute it in the VO scope. This is for instance needed when a special gid is assigned basing on the user's VO or when VOs should be charged for their jobs.

To associate a request with a VO the user has to select one or administrator can define a default which is used when user didn't select a VO. User can select an effective VO using request preference `selectedVirtualOrganisation`. Of course it must be one of the VOs the user is member of.

Administrator can configure a list of preferred VOs. If such a list is provided, then the first VO from the list, where the user is a member is used when user don't provide her own selection. See the general security configuration options for the syntax: Section 2.8.2.

If it is required that all requests should have the effective VO set, then it is possible to deny other requests using an additional rule in the authorization policy. The rule should deny all requests that doesn't have the selectedVO authorization attribute. See Section 17 for details.

7.1.3 Supported VO servers

This module was tested and works well with the Unity system.

There are other possibilities and you can try to use any SAML (2.0) Attribute service. We are interested in all success/failure stories!

7.1.4 VO deployment planning

First of all it must be decided which VO/group (in UNICORE case it doesn't matter whether a VO or VO subgroup is used, all subgroups can be treated as a full-fledged VOs, and VOs are just a nick-name of top-level groups) is used by a site.

In case when a site needs only generic, federation-wide attributes from a VO, a group which is common for all sites should be used. Such a group can provide for instance the *role* attribute for the members. Of course if uids are the same across all sites, then uids can be also assigned in such VO.

In the case when a site needs also some site-specific attributes, a dedicated group should be created for the site, as a subgroup of a VO (e.g /VO1/sites/SiteA). VO administrators should assign VO-scoped attributes in this group and make sure that all universal VO attributes are also replicated there. Please note that Unity allows for outsourcing VO management on a per-group basis, so it is possible to assign administrative permissions to such group, for a site representative.

The next issue is how to handle a situation when there are multiple Unix user IDs or roles available for the user, and how to mark the default one? To overcome this, for every incarnation attribute it is possible to define two VO attributes. The base one can possess many values (e.g. in case of UIDs every value is a different UID) while the additional attribute holds a single default value. When there is no need for multiple values then the base attribute can be used alone. When default attribute is defined then its value is used unless a user provided some preferences. Of course such preferences must be valid, i.e. be included in the allowed values of the base attribute.

Details on what attributes are used for those purposes are presented in the following section.

7.2 Configuration

This sections describes the default configuration file format which is used to configure the VO attribute source. This section provides a detailed and comrehensive information on all configuration options. In the most cases defaults are fine - you can consult the HOWTO (below) for a short *quick start* information.

Some of the configuration options require a value of a VO/GROUP type. Whenever it is needed it should be written in the following way:

```
/VO[/group1[/subgroup2[...]]]
```

where elements in square brackets are optional. E.g. `/Math/users` denotes a group *users* of a VO called *Math*.

In case of UNICORE/X and other USE servers the configuration is provided in a separate file, by default the `vo.config` from the configuration directory of the server (you can change location and name of this file, see below). It holds generic VO configuration which is not dependent to the actual server used - the most of settings is configured there. This file options are described below Section 7.2.1.

To enable the VO subsystem certain settings are also required in the main server's configuration file. You have to define an appropriate Attribute Source. You can use only one or even use multiple instances. The latter situation occurs when you want to support multiple VOs (from one or multiple VO servers) - then you have to define one attribute source per VO (or VO group).

Example with a VO attribute sources and also with local XUUDB. Local data from XUUDB (if it exists) will override attributes received from VOs:

```
container.security.attributes.order=SAML-PULL XUUDB
# ... here comes xuudb configuration

container.security.attributes.SAML-PULL.class=eu.unicore.uas. ↩
    security.vo.SAMLPullAuthoriser
container.security.attributes.SAML-PULL.configurationFile=conf/vo. ↩
    config
```

Before proceeding to fill the VO configuration it is suggested to prepare the VO truststore, which should contain **ONLY** the certificates of the trusted VO servers. Note that this file must not contain any CA certificates, only the trusted VO servers' certificates! This file is optional, but will increase security.

Logging configuration is done by means of standard UNICORE logging configuration file. See Section 7.2.3 section for possible settings related to the VO subsystem.

7.2.1 Main VO configuration file

The following sections provide complete reference of available options for the main configuration file (usually `vo.config`).

Property name	Type	Default value / mandatory	Description
vo.group	string	-	DEPRECATED
vo.localServerURI	string	-	Can contain this, local server SAML identifier, to be used in SAML requests in PULL mode. If unset then DN identity is used for queries, created from the local server's credential.
vo.pull.cacheTtl	integer number	600	Controls pulled attributes cache. Set to negative integer to disable the caching or to positive number - lifetime in seconds of cached entries.
vo.pull.disableIfAttributesWerePushed	[true, false]	true	DEPRECATED
vo.pull.enable	[true, false]	true	DEPRECATED
vo.pull.enableGeneralAttributes	[true, false]	true	If turned on, then not only the recognized UNICORE attributes are processed, but also all others, which can be used for authorization.
vo.pull.verifySignatures	[true, false]	true	Additional security for the pulled assertions (except transport level which is always on) can be achieved by verification of signatures of the received assertions. The key which is used for verification must be present in the VO truststore.
vo.pull.voServerAuthn.password	string	-	If certificate-based authentication to the VO server is disabled, you might be able to use username/password. This sets the password.
vo.pull.voServerAuthn.username	string	-	If certificate-based authentication to the VO server is disabled, you might be able to use username/password. This sets the username.

Property name	Type	Default value / mandatory	Description
`vo.pull.voServerURI`	string	`localhost`	Full address (URL) of SAML VO service. Note that this server's CA cert must be present in the main truststore of the server to create the connection.
`vo.push.enable`	[true, false]	`false`	DEPRECATED
`vo.truststore.[.*]`	string *can have subkeys*	-	Properties starting with this prefix are used to configure validation of VO assertion issuers certificates. Trust anchors should contain only the trusted VO servers certificates. All options are the same as those for other UNICORE truststores.
`vo.unicoreAttribute.[.*]`	string *can have subkeys*	-	Properties starting with this prefix are used to configure mappings of SAML attributes to UNICORE internal ones.
`vo.voServerURI`	string	-	Identification URI of the VO service providing attributes for this source. Only attributes issued by this issuer will be honoured.

The following table shows options, which are used to define mappings of SAML attributes to UNICORE incarnation attributes (the available names of UNICORE incarnation attributes are provided in Section 6.1).

Property name	Range of values	Description
`vo.unicoreAttribute.NAME`	URI	Value must be a SAML attribute name which will be used as a UNICORE internal incarnation attribute NAME.
`vo.unicoreAttribute.NAME.default`	URI	Value must be a SAML attribute name which will be used as a default for UNICORE internal incarnation attribute NAME.

Property name	Range of values	Description
vo.unicoreAttribute.NAME.disabled	ANY, IGNORED	When this attribute is present regardless of its value the NAME attribute won't be mapped.

7.2.2 Example mapping for Unity attributes

Note that your distribution should contain a sensible default for Unity attribute mappings, which does not need to be modified.

```
# standard settings for the xlogin mapping, however let's ignore  ↩
    pushed xlogins
vo.unicoreAttribute.xlogin=urn:unicore:attrType:xlogin
vo.unicoreAttribute.xlogin.default=urn:unicore:attrType: ↩
    defaultXlogin
vo.unicoreAttribute.xlogin.pushDisabled=

#standard role mapping
vo.unicoreAttribute.role=urn:unicore:attrType:role
vo.unicoreAttribute.role.default=urn:unicore:attrType:defaultRole

#supplementary groups are stored in a non standard attribute
vo.unicoreAttribute.supplementaryGroups=urn:ourCompany: ↩
    secondaryGids

#and group - without default
vo.unicoreAttribute.group=urn:unicore:attrType:primaryGid

#queue mapping is defined, but will be ignored (disabled)
vo.unicoreAttribute.queue=urn:unicore:attrType:queue
vo.unicoreAttribute.queue.default=urn:unicore:attrType:defaultQueue
vo.unicoreAttribute.queue.disable=

# addDefaultGroups - is not defined, so won't be mapped

#getting the user's groups is always a good idea
vo.unicoreAttribute.virtualOrganisations=urn:SAML:voprofile:group
```

7.2.3 Logging configuration

All components use the usual log4j/2 logging mechanism. All events are logged with `unicore.security.vo` prefix. The reporting class name is appended.

As an example, a configuration for logging all events for the VO subsystem can be specified as follows:

```
logger.vo.name=unicore.security.vo
logger.vo.level=trace
```

7.3 VO configuration HOWTOs

7.3.1 SAML-PULL and UNICORE - basic case

This section shows all the steps which are required to setup a UNICORE/X server and Unity to work in the SAML-PULL mode. In this scenario we will use Unity to store at a central point mappings of DNs to UNIX logins (Xlogins) and roles of of our users. The UNICORE/X server will then query (pull) attributes from Unity, similar to using an XUUDB.

Note
We write UNICORE/X in the following, but any server based on the UNICORE Services Environment (registry, workflow, etc) works the same way

---

The required steps are:

1. Add Unity's CA certificate to the UNICORE/X truststore (so SSL connections can be established)
2. Add UNICORE/X's CA certificate to the Unity server's truststore (so SSL connections can be established).
3. Add the UNICORE/X server's DN (from its certificate) as a member to the Unity service. You don't have to make it a member of any particular VO (or group). However it must have the **read** permission to all groups where its users will be placed. In Unity, this corresponds to the "Priviledged Inspector" role (check Unity documentation for details).
4. Check that UNICORE/X can properly authenticate to Unity on the SAML endpoint that is used to query attributes. Generally this will be via the UNICORE/X certificate, if that is not possible, you'll need to setup an additional username identity for the entity created in Step 3, and setup password authentication.
5. Create a VO (possibly with subgroups). Add users to the group. Here we will assume this group is `/Math-VO/UUDB/SiteA`. Next assign them *in the scope of the group* attribute `urn:unicore:attrType:xlogin` with the value of Unix UID for the user, and attribute `urn:unicore:attrType:role` with the value of the user's role (usually its just `user`). Note that if you want to assign the same Xlogin/role to multiple users then you can define Unity *group attributes* and set them for the whole `/Math-VO/UUDB/SiteA` group.
6. Enable VO attribute source ("SAML PULL") in the UNICORE server. Here we will configure it as the primary source and leave XUUDB to provide local mappings (which can override data fetched from Unity). You should have the following entries:

```
container.security.attributes.order=SAML-PULL XUUDB
container.security.attributes.combiningPolicy= ↩
    MERGE_LAST_OVERRIDES
# ...  xuudb configuration omitted

container.security.attributes.SAML-PULL.class=eu.unicore.uas. ↩
    security.vo.SAMLPullAuthoriser
```

7. Configure VO attribute source (typically in the `vo.config`) file as follows:

```
vo.group=/Math-VO/UUDB/SiteA

vo.truststore.type=directory
vo.truststore.directoryLocations.1=/opt/unicore/certs/unity/*. ↩
    pem

vo.localServerURI=https://example.org:7777

# ##########################
# PULL mode configuration
# ##########################

vo.pull.enable=true

vo.pull.cacheTtl=20

vo.pull.voServerURL=https://unity.example.org/unicore-soapidp/ ↩
    saml2unicoreidp-soap/AssertionQueryService

vo.pull.verifySignatures=false

# Mapping of Unity attributes (right side) to the special,  ↩
    recognized by UNICORE
#  incarnation attributes (left)
vo.unicoreAttribute.xlogin=urn:unicore:attrType:xlogin
vo.unicoreAttribute.xlogin.default=urn:unicore:attrType: ↩
    defaultXlogin
vo.unicoreAttribute.role=urn:unicore:attrType:role
vo.unicoreAttribute.role.default=urn:unicore:attrType: ↩
    defaultRole
vo.unicoreAttribute.group=urn:unicore:attrType:primaryGid
vo.unicoreAttribute.group.default=urn:unicore:attrType: ↩
    defaultPrimaryGid
vo.unicoreAttribute.supplementaryGroups=urn:unicore:attrType: ↩
    supplementaryGids
vo.unicoreAttribute.supplementaryGroups.default=urn:unicore: ↩
    attrType:defaultSupplementaryGids
vo.unicoreAttribute.addDefaultGroups=urn:unicore:attrType: ↩
    addDefaultGroups
```

```
vo.unicoreAttribute.queue=urn:unicore:attrType:queue
vo.unicoreAttribute.queue.default=urn:unicore:attrType: ↩
    defaultQueue
vo.unicoreAttribute.virtualOrganisations=urn:SAML:voprofile: ↩
    group
```

8. In the VO truststore directory (/opt/unicore/certs/unity/ in this case) put the Unity certificate (NOT the CA certificate) as a PEM file, with pem extension.

7.3.2 Advanced example: Unity and UNICORE - using fine grained authorization

In this scenario we will enhance the first one to use custom authorization attributes in UNICORE policy. To do so ensure that you have this setting in `vo.config` file: `vo.pull.enableGenericAttributes=true` Then you can modify XACML policy to require certain VO attributes.

Important fact to note here (and in case of PUSH mode too) is how the user's group membership is encoded as an XACML attribute. By default it is an attribute of string type (so XACML *DataType="http://www.w3.org/2001/XMLSchema#string"*) with its name (*AttributeId*) equal to *urn:SAML:voprofile:group*. The example policy below uses this attribute.

The following XACML fragment allows for reaching TargetSystemFactory service only for the users which are both members of VO `Example-VO` and a VO group `/Math-VO/UUDB/SiteA`. Moreover those users also must have a standard UNICORE/X attribute role with a value *user*. It means that in Unity, UNICORE users must have `urn:unicore:attrType:role` attribute defined (it is the standard setting) with a value *user*.

```
<Rule RuleId="AcceptTSF" Effect="Permit">
  <Description>
    Accept selected users to reach TSF
    </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">TargetSystemFactoryService</ ↩
              AttributeValue>
          <ResourceAttributeDesignator DataType="http://www.w3.org ↩
              /2001/XMLSchema#anyURI" AttributeId="urn:oasis:names: ↩
              tc:xacml:1.0:resource:resource-id"/>
        </ResourceMatch>
      </Resource>
    </Resources>
  </Target>
  <Condition>
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
      <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">user</AttributeValue>
      </Apply>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:any- ↩
          of-all">
        <Function FunctionId="urn:oasis:names:tc:xacml:1.0:function ↩
            :string-equal"/>
        <SubjectAttributeDesignator DataType="http://www.w3.org ↩
            /2001/XMLSchema#string" AttributeId="urn:SAML:voprofile ↩
            :group"/>
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function: ↩
            string-bag">
          <AttributeValue DataType="http://www.w3.org/2001/ ↩
              XMLSchema#string">/Example-VO</AttributeValue>
          <AttributeValue DataType="http://www.w3.org/2001/ ↩
              XMLSchema#string">/Math-VO/UUDB/SiteA</AttributeValue ↩
              >
        </Apply>
      </Apply>
    </Apply>
  </Condition>
</Rule>
```

8 The UNICORE persistence layer

UNICORE stores its state in data bases. The information that is stored depends on the services that are running in the container, and can include

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

etc.

The job directories themselves reside on the target system, but UNICORE keeps additional information (like, which UNICORE 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 three kinds of storage:

- on the filesystem of the UNICORE/X server (using the H2 database engine), which is generally the default;
- on a database server (MySQL, or the "server mode" of H2);
- in-memory, i.e. all info is lost on server restart.

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 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.

8.1 Configuring the persistence layer

Peristence properties are configured in two files:

- container.properties for all service data
- xnjs.properties 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.size`	integer number *can have subkeys*	`10`	Maximum number of elements in the cache (default: 10).

Property name	Type	Default value / mandatory	Description
`persistence.class[.*]`	string *can have subkeys*	`de.fzj.unicore.persist.impl.H2Persist`	The persistence implementation class, which controls with DB backend is used.
`persistence.clustering.config[.*]`	string *can have subkeys*	-	Clustering configuration file.
`persistence.clustering.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*	`1024`	(H2) Cache size.
`persistence.h2.options[.*]`	string *can have subkeys*	-	(H2) Further options separated by *;*.
`persistence.h2.server[.*]`	[true, false] *can have subkeys*	`false`	(H2) Connect to a H2 server.
`persistence.host[.*]`	string *can have subkeys*	`localhost`	The database host.
`persistence.max_connections[.*]`	integer number *can have subkeys*	`1`	Connection pool maximum size.
`persistence.mysql.table_type[.*]`	string *can have subkeys*	`MyISAM`	(MySQL) Table type (engine) to use.
`persistence.mysql.timezone[.*]`	string *can have subkeys*	`UTC`	(MySQL) Server timezone.
`persistence.mysql.useSSL[.*]`	[true, false] *can have subkeys*	`false`	(MySQL) Connect using SSL.
`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.

Property name	Type	Default value / mandatory	Description
`persistence.port[...]`	integer number *can have subkeys*	`3306`	The database port.
`persistence.user[...]`	string *can have subkeys*	`sa`	The database username.

8.1.1 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
```

8.1.2 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 H2 server mode, the connection URL is constructed as follows

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

8.1.3 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
```

If you want to store data from multiple UNICORE servers, make sure to use a different database for each of them.

8.2 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 (https://hazelcast.org/documentation). A basic TCP based configuration is

```
<hazelcast xmlns="http://www.hazelcast.com/schema/config">
  <group>
      <name>persistence-dev</name>
      <password>dev-pass</password>
  </group>
  <network>
      <port auto-increment="true">5701</port>
      <join>
          <multicast enabled="false"/>
          <tcp-ip enabled="true">
              <!-- list other members of the cluster -->
              <member>127.0.0.1</member>
              <member>some.host.org</member>
          </tcp-ip>
      </join>
  </network>
</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.

9 Configuring the XNJS

The XNJS is an internal UNICORE/X component that deals with the actual job execution and file system access. It is configured using a properties file named *xnjs.properties*. It is include'd from the main config file.

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

Property name	**Type**	**Default value / mandatory**	**Description**
`XNJS.allowUserExecutable`	[true, false]	`true`	Whether to allow user-defined executables. If set to false, only applications defined in the IDB may be run.
`XNJS.autosubmit`	[true, false]	`false`	Automatically submit a job to the BSS without waiting for an explicit client start.

Property name	Type	Default value / mandatory	Description
XNJS.bssResubmitCount	integer >= 1	3	How often should UNICORE/X try to submit a job to the BSS.
XNJS.bssResubmitDelay	integer >= 1	10	Minimum delay (in seconds) between attempts to submit a job to the BSS.
XNJS.filespace	string	-	Directory on the TSI for the job directories. Must be world read/write/executable.
XNJS.filespaceUmask	integer number	0002	Umask to be used for creating the base directory for job directories.
XNJS.idbfile[.*]	string *can have subkeys*	-	IDB configuration.
XNJS.idbtype	string	json	IDB format: *json* (default) or *xml*
XNJS.incarnationTweakerConfig	string	*not set*	Path to configuration file for the incarnation tweaker subsystem. If not set, the subsystem will be disabled.
XNJS.localtsi[.*]	string *can have subkeys*	-	Properties for configuring the embedded Java TSI (if used). See separate docs.
XNJS.numberofworkers	integer >= 0	4	Number of XNJS worker threads.
XNJS.parameterSweepLimit	integer >= 0	200	Upper limit for number of jobs generated in a single parameter sweep.
XNJS.staging[.*]	string *can have subkeys*	-	Properties for configuring the data staging and I/O components. See separate docs.
XNJS.staging.addWaitingLoop	[true,false]	false	Whether to add a waiting loop for files to appear on shared filesystems.
XNJS.staging.filespaceTimeout	integer >= 1	10	Grace time (in seconds) when waiting for files to appear on shared filesystems.

Property name	Type	Default value / mandatory	Description
XNJS.strictUserInputChecking	[true, false]	true	Whether to be restrictive in checking user-supplied arguments and environment variables. Set to true if you do not want ANY user code to run on your TSI node.

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.

9.1 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 Slurm 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.properties
- Start the newly installed TSI (as root in a multiuser setting)
- Restart UNICORE/X

9.1.1 Installation of the correct TSI

The TSI is a service 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.

As the TSI is a central and sensitive service, make sure to read its documentation. This guide serves just as a quick overview of the necessary steps.

- 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 configuration, especially command locations, path settings etc.

9.1.2 Required TSI Configuration

Configuration is done by editing <tsi_conf_dir>/tsi.properties At least check the following settings:

```
# 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
```

9.1.3 UNICORE/X configuration

Edit *unicorex/conf/uas.config* and check that the "xnjs.properties" file is included

```
# read XNJS/TSI config
$include.XNJS conf/xnjs.properties
```

Edit *unicorex/conf/xnjs.properties*. 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. Also, the filespace location has to be read/write/executable for the current user. If you wish to

avoid a world-executable directory, it is possible to use a per-user location, like *$HOME/UNICORE_Jobs*.

Check the *CLASSICTSI* related properties. Set the correct value for the machine and the ports (these can usually be left at their default values). The CLASSICTSI.machine property is a comma separated list of machines names or IP addresses. Optionally, a port number can be added to each entry, separated from the machine by a colon. The XNJS will establish connections to each of these machines and ports in a round-robin fashion to ensure that jobs can be submitted and job statuses retrieved even if one of the TSI instances is unavailable. Should the port not be given along with the machine, CLASSICTSI.port will be used as a default.

Here is an small example.

```
XNJS.filespace=$HOME/UNICORE_Jobs/
XNJS.idbfile=/opt/unicore/unicorex/conf/simpleidb

CLASSICTSI.machine=login.mycluster.com
CLASSICTSI.port=4433
CLASSICTSI.replyport=7654
CLASSICTSI.priveduser=unicore

XNJS.staging.wget=wget --no-check-certificate
```

9.1.4 Communication parameters

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

Table 9: 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

9.1.5 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
CLASSICTSI.statusupdate.grace=2
```

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

Note
When changing TSIs, it's a good idea to remove the UNICORE/X state and any files before restarting. See Section 8 for details

9.1.6 Enabling SSL for the XNJS to TSI communication

The UNICORE/X server can be set up to use SSL for communicating with the UNICORE 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 -
CLASSICTSI.ssl.disable=false
```

To setup the TSI side, please refer to the TSI manual!

9.1.7 Using an SSH tunnel for the XNJS to TSI communication

In the special case that the XNJS callback port is not accessible from the TSI server, you may want to use an SSH tunnel configuration. For example, this case occurs if the TSI is running in a different location (e.g. an Amazon cloud) than the UNICORE/X server.

We recommend using the tool "autossh", and adding the tunnel setup to to your UNICORE/X start script.

Here is an example how to do this

```
killall -g autossh
autossh -M 0 -f -o "ExitOnForwardFailure=yes" -o " ↩
    ServerAliveInterval 30"
  -o "ServerAliveCountMax 3" -4 -N
  -L 4433:localhost:4433
  -R 7654:localhost:7654
  -i path_to_key remoteuser@remote.server.org
```

9.1.8 TSI configuration parameter reference

Here is a full list of TSI-related parameters.

Property name	Type	Default value / mandatory	Description
CLASSICTSI.BUFFERSIZE	integer >= 1	1048576	Buffer size (in bytes) for transferring data from/to the TSI.
CLASSICTSI.CD	string	cd	Unix *cd* command.
CLASSICTSI.CHGRP	string	/bin/chgrp	Unix *chgrp* command.
CLASSICTSI.CHMOD	string	/bin/chmod	Unix *chmod* command.
CLASSICTSI.CP	string	/bin/cp	Unix *cp* command.
CLASSICTSI.FSID	string	-	TSI filesystem identifier which uniquely identifies the file system. The default value uses the *CLASSICTSI.machine* property.
CLASSICTSI.GROUPS	string	groups	Unix *groups* command.
CLASSICTSI.LN	string	/bin/ln -s	Unix *ln* command.
CLASSICTSI.MKDIR	string	/bin/mkdir -p	Unix directory creation command.
CLASSICTSI.MKFIFO	string	/bin/mkfifo	Unix *mkfifo* command.
CLASSICTSI.MV	string	/bin/mv	Unix *mv* command.
CLASSICTSI.PS	string	ps -e	Command to get the process list on the TSI node.
CLASSICTSI.RM	string	/bin/rm	Unix *rm* command.
CLASSICTSI.RMDIR	string	/bin/rm -rf	Unix directory removal command.
CLASSICTSI.UMASK	string	umask	Unix *umask* command.
CLASSICTSI.interactiveExecutionDisable	[true, false]	false	Disable execution of user commands on the TSI node.
CLASSICTSI.limitTSIprocesses	integer number	-1	Limit the total number of TSI worker processes created by this UNICORE/X (*-1* means no limit).
CLASSICTSI.machine	string	localhost	TSI host(s) or IP address(es). Specify multiple hosts in the format *machine1[:port1],machine2[:port2],...*
CLASSICTSI.pooledTSIconnections	integer >= 1	4	How many TSI worker processes per TSI host to keep (even if idle).
CLASSICTSI.port	integer >= 1	4433	TSI port to connect to.

Property name	Type	Default value / mandatory	Description
CLASSICTSI.privedusern	string	unicore	Account used for getting statuses of all batch jobs (cannot be *root*).
CLASSICTSI.replypoint	integer >= 1	7654	Reply port on UNICORE/X server.
CLASSICTSI.reservation	string	AdminUser unicore	Account used for making reservations (cannot be *root*). If null, the current user's login will be used.
CLASSICTSI.reservation	[true, false]	false	Whether to enable the reservation interface.
CLASSICTSI.socket.conn	integer >= 0	10	Connection timeout (seconds) on when establishing (or checking) the TSI connection. Set to *0* for no timeout.
CLASSICTSI.socket.[true, false]		false	Disable checking if IP address(es) of command/data socket callbacks are as expected.
CLASSICTSI.socket.read	integer >= 0	180	Read timeout (seconds) on the TSI connection. Set to *0* for no timeout.
CLASSICTSI.ssl.disable	[true, false]	true	Whether to disable SSL for the TSI-UNICORE/X connection.
CLASSICTSI.statusupdate	integer >= 1	2	How many times the XNJS will re-check job status in case of a *lost* job.
CLASSICTSI.statusupdate	integer >= 1	10000	Interval (ms) for updating job statuses on the batch system.

9.2 Operation without a UNICORE TSI

In some situations (e.g. in a Windows-only environment) you will not use the UNICORE TSI, which is designed for multi-user Unix environments. The XNJS can run code in an "embedded" mode on the UNICORE/X machine. Note that this is without user switching, and inherently not secure as user code can access potentially sensitive information, such as configuration data. Also, there is no separation of users.

Embedded mode is enabled in xnjs.properties file by setting

```
coreServices.targetsystemfactory.tsiMode=embedded
```

The embedded mode can be configured with a set of properties which are listed in the following table.

Property name	Type	Default value / mandatory	Description
XNJS.localtsi.jobLimit	integer number	0	Maximum number of concurrent jobs, if set to a value >0. Default is no limit.
XNJS.localtsi.shell	string	/bin/bash	Default UNIX shell to use (if shell is used).
XNJS.localtsi.useShell	[true|false]	true	Should a UNIX shell be used to execute jobs.
XNJS.localtsi.workerThreads	integer >=1	4	Number of worker threads used to execute jobs.

10 The IDB

The UNICORE IDB (incarnation database) contains information on the target system capabilities (like number of nodes, CPUs etc) and allowing to check client resource requests against these.

The second IDB function is to define abstract application definitions that are then mapped onto concrete executables. This process (called "incarnation") is performed by the XNJS component.

10.1 Defining the IDB location

The IDB file is defined by the property "XNJS.idbfile", which must point to a file or directory on the UNICORE/X machine which is readable by the UNICORE/X process.

10.1.1 Using an IDB directory

While the IDB can be put into a single file, it can be convenient to use multiple files. In this case, the property "XNJS.idbfile" points to a directory. The information from all files in this directory is merged.

When using a directory, you can optionally specify a "main" IDB file containing applications, resources, properties etc. From other files, only Applications will be read. A main IDB file is defined via "XNJS.idbfile.main"

10.1.2 User-specific applications (IDB extensions)

Sometimes it is required to define special applications for (groups of) users, and even let users define their own applications. This means that the set of available applications differs between users.

User specific applications can be defined using additional properties, for example like this:

```
XNJS.idbfile.ext.1=/opt/staff/unicore/*.xml
XNJS.idbfile.ext.2=$HOME/.unicore/*.xml
XNJS.idbfile.ext.3=$WORK/projects/apps/*.xml
```

These paths are resolved on the TSI machine, NOT on UNICORE/X. As you can see, they can contain variables (using `$VARIABLE` syntax WITHOUT curly braces!). Make sure that the numbering is consistent (ext.1,ext.2,...).

Note
Some UNICORE features such as brokering in workflows might not (yet) work with user-specific applications!

10.1.3 Examples for IDB setup

Here are a few common IDB config examples

Single IDB file (default)

```
XNJS.idbfile=/etc/unicore/unicorex/simpleidb
```

IDB directory, all files are merged

```
XNJS.idbfile=/etc/unicore/unicorex/idb/
```

IDB directory, main file defined, read apps from all other files

```
XNJS.idbfile=/etc/unicore/unicorex/applications/
XNJS.idbfile.main=/etc/unicore/unicorex/simpleidb
```

IDB directory, main file defined, user-specific extension

```
XNJS.idbfile=/etc/unicore/unicorex/applications/
XNJS.idbfile.main=/etc/unicore/unicorex/simpleidb
XNJS.idbfile.ext.1=$HOME/.unicore/apps/*.xml
```

10.2 IDB syntax description

Starting with UNICORE 8.0, the IDB is written in JSON. This documentation focuses on the JSON format.

Note
The older XML format is deprecated, but still supported. It is limited to a single partition, i.e. a single set of resource limits (number of nodes / CPUs etc). For reference, the schema for the XML̀ IDB can be found at http://svn.code.sf.net/p/unicore/svn/xnjs/trunk/src/main/schema/-idb.xsd

---

The IDB contains Partitions, Applications, Submit/Excute script templates and Info elements, all of which will be described below. Additionally the administrator can customize the script template that is used to

Applications can also be defined in separate files (if using a directory)

```
{
"Partitions" : {},

"Info" : {},

"Applications" : [],

"ExecuteScriptTemplate" : "...",

"SubmitScriptTemplate"  : "...",

}
```

10.2.1 Partitions

Each Partition corresponds essentially to a batch queue. Each partition may have its own run-time limits, number of CPUs etc.

Let's look at an example first. In the IDB file

```
{
"Partitions": {

 "batch" : {
    "IsDefaultPartition": "true",
    "Description": "Default batch queue",
    "OperatingSystem": "LINUX",
    "OperatingSystemVersion": "4.15.0-62-generic / Ubuntu 18.04",
    "CPUArchitecture": "x86_64",
    "Resources": {
```

```
      "Nodes": "1-64:1",
      "CPUsPerNode": "4",
      "TotalCPUs": "4-256",
      "Runtime": "1-72000:3600",
    },
 },

  "dev" : {
    "Description": "Development queue",
    "OperatingSystem": "LINUX",
    "CPUArchitecture": "x86_64",
    "Resources": {
      "Nodes": "1-4:1",
      "CPUsPerNode": "4",
      "TotalCPUs": "4-16",
      "Runtime": "1-3600:10m",
    },
 },

}
```

If you have more than one Partition, make sure to set one as the default using the element

```
  "IsDefaultPartition": "true",
```

Resources

Here you can specify things like number of nodes, job runtime (wall time!) CPUs per node, total number of CPUs, etc.

Integer-valued capabilities are specified with a range and an optional default, for example:

```
 "Nodes" : "1-64:1",
```

or in a more verbose style:

```
  "Nodes" : {
    "Range": "1-64",
    "Default": "1",
 }
```

If a default is specified, the resource is part of the site's default resource set, and a value will be always be sent to the TSI.

If NO default is specified, the resource request will only be sent to the TSI if the user has requested it in her job.

A number of standard resource names exist, which a system should adhere to, in order to make user jobs as portable as possible. 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", or only "Nodes".

- `Runtime` : The wall clock time (integer). You can use the usual units ("m", "h", "d"), e.g. "12h"
- `Nodes` : The number of nodes (integer)
- `CPUsPerNode` : The number of CPUs per node (integer)
- `TotalCPUs` : The total number of CPUs (integer)
- `MemoryPerNode` (or just `Memory`) : The amount of memory per node in bytes (integer). You can use the usual units ("k", "M", G"), e.g. "128G"
- `NodeConstraints` : Identifiers for requesting specific node types (list of values)
- `QoS` : Quality of service required by the job (list of values)

```
"NodeConstraints" : {
 "Type": "CHOICE",
 "AllowedValues" : ["gpu", "mc"],
}
```

Support for array jobs

Many resource managers support submission of job arrays, i.e. multiple similar jobs are submitted at the same time, where the user can control two things: how many jobs are submitted, and how many jobs run at the same time.

To enable this feature, the site administrator needs to define two resources in the IDB partition(s), named "ArraySize" and "ArrayLimit".

Consider the following example:

```
"ArraySize"  : "1-100:1",
"ArrayLimit" : "1-100:10",
```

The array size and limit are passed to the TSI via

```
#TSI_ARRAY 0-99
#TSI_ARRAY_LIMIT 10
```

The TSI also sets an environment variable in the job script that corresponds to the "task id", i.e. the ID of the current job instance:

```
UC_ARRAY_TASK_ID = "22"; export UC_ARRAY_TASK_ID
```

10.2.2 Other types of resources

Most HPC sites have special settings that cannot be mapped to the generic resource elements shown in the previous section. Therefore UNICORE allows to define custom system settings and allow users to request these in their UNICORE jobs.

Custom resources have a name, and a short specification including their type and range and/or allowed values.

UNICORE/X will send such resource requests to the TSI in upper case, with a "#TSI_SSR_" prefix, e.g.

```
#!/bin/sh
#TSI_SUBMIT
# ...
#TSI_SSR_GPUS 4
# ....
```

Custom resource definitions support the following fields:

- `Type` : int (default), double, string, choice or boolean
- `Range` : (*int*, *float*) allowed range of the form "lower-upper"
- `Default` : optional default value
- `AllowedValues` : (for *choice*) list of strings
- `Description` : optional description

Here are a few examples:

```
"LicenseKey" : {
 "Type": "String"
}

"UserSupportClass" : {
 "Type": "CHOICE",
 "AllowedValues" : ["bronze", "silver", "gold"],
 "Default": "bronze"
}

"ReservedBandwidth" : {
 "Type": "int",
 "Range" "1-100",
}
```

For "int" resources, you can alternatively use the abbreviated definition, as shown above for the standard resources (such as *Nodes*). For example

```
"FPGAs" : "0-1024"
```

10.2.3 Script templates

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

```
"SubmitScriptTemplate" : "#!/bin/sh \n #COMMAND \n#RESOURCES \n# ↩
    SCRIPT \n",

"ExecuteScriptTemplate" : "#!/bin/sh \n#COMMAND \n#RESOURCES \n# ↩
    SCRIPT \n"
```

(JSON requires this as a single-line string)

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:

- "#COMMAND" entry will be replaced by the action for the TSI, e.g. "#TSI_SUBMIT".
- (for submit)the "#RESOURCES" will be replaced by the resource requirements, e.g. "#TSI_NODES=..."
- "#SCRIPT" will be the user script / the executed command

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

```
"SubmitScriptTemplate" :
"#!/bin/bash \n#COMMAND \n#RESOURCES \nLD_LIBRARY_PATH= ↩
    $LD_LIBRARY_PATH:/opt/openmpi-2.1/lib; export LD_LIBRARY_PATH \ ↩
    nPATH=$PATH:/opt/openmpi-2.1/bin; export PATH \n#SCRIPT \n",
```

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

---

For example

```
"ExecuteScriptTemplate" :
"#!/bin/bash \n#COMMAND \nmodule load java-11 > /dev/null 2>&1 \n# ↩
    SCRIPT \n",
```

10.2.4 Info

Simple key-value pairs 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.

Here is an example

```
{
   "Info" : {
     "ssh-host"    : "login.cluster.com",
     "admin-email" : "root@cluster.com",
   },
}
```

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

10.2.5 Summary

Table 10: Translation of standard resource names to TSI parameters

Resource	**TSI parameter**
Name of the selected partition	#TSI_QUEUE
Accounting project (from job)	#TSI_PROJECT
Runtime	#TSI_TIME
Nodes	#TSI_NODES
CPUsPerNode	#TSI_PROCESSORS_PER_NODE
TotalCPUs	#TSI_TOTAL_PROCESSORS
NodeConstraints	#TSI_BSS_NODES_FILTER
QoS	#TSI_QOS
MemoryPerNode (or Memory)	#TSI_MEMORY
ArraySize	#TSI_ARRAY
ArrayLimit	#TSI_ARRAY_LIMIT
Other resources	#TSI_SSR_<name>

10.3 IDB Application definitions

Apart from describing the available queues and their associated resources, the most important functionality of the IDB is defining applications.

Applications can be defined in the main IDB file,

```
{
 Applications: [
   { Name: Date, ...  },
   { Name: "Python script", ... },
 ],
}
```

or in separate files (one application per file).

Here is a quick overview of the available elements, which will be documented in detail below.

Table 11: JSON IDB Application

Tag	**Type**	**Description**	**Optional/mandatory**
Name	String	Application name	Mandatory
Version	String	Application version	Mandatory
Description	String	Application description	Optional
Executable	String	Executable	Mandatory
Arguments	List of strings	Command line arguments	Optional
Environment	Map of strings	Environment values	Optional
PreCommand	String	Pre-processing executed on the login node	Optional
PostCommand	String	Post-processing executed on the login node	Optional
Prologue	String	Pre-processing in the batch script	Optional
Epiloge	String	Post-processing in the batch script	Optional
Parameters	Map	Metadata for application arguments / parameters	Optional
Resources	Map	Application-specific resource requests	Optional
RunOnLoginNode	"true"/"false"	Run job on login node	Optional, default=false

Table 11: (continued)

Tag	Type	Description	Optional/mandatory
IgnoreNonZeroExitCode	"true"/"false"	Don't fail the job if app exits with non-zero exit code	Optional, default=true

Here is an example:

```
{
 Name: "Python script",
 Version: "3.4",
 Description: "Python 3 interpreter",
 Executable: "/usr/bin/python3",
 Arguments: [
    "-d$DEBUG?",
    "-vVERBOSE?",
    "$OPTIONS?",
    "$SOURCE?",
    "$ARGUMENTS",
  ],

 Parameters: {
   "SOURCE": {Type: "filename"},
   "ARGUMENTS": {Type: "string"},
   "DEBUG": {Type: "boolean"},
   "VERBOSE": {Type: "boolean"},
   "OPTIONS": {Type: "string"},
 },

 Prologue: "module load python3",

 Resources: {
  Nodes: 1,
 }
}
```

10.3.1 Basic Application definition

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

```
{
  Name: Date,
  Version: 1.0,
  Executable: "/bin/date",
}
```

Invoking the "Date" application will be simply mapped to an invocation of "/bin/date".

10.3.2 Arguments

Command line arguments are specified using <Argument> tags:

```
{
  Name: LS,
  Version: 1.0,
  Executable: /bin/ls
  Arguments: [ "-l", "-t", ],
}
```

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

10.3.3 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:

```
{
  Name: JAVA,
  Version: "11.0",
  Description: "Java virtual machine",
  Executable: "/usr/bin/java",

  Arguments: [ "-cp$CLASSPATH?", ],

}
```

Here, `-cp$CLASSPATH?` is an optional argument.

If the user's job submission now includes a environment variable named `CLASSPATH` the incarnated commandline will be "/usr/bin/java -cp$CLASSPATH ...", otherwise just "/usr/bin/java ...".

This allows very flexible incarnations.

10.3.4 Environment variables

To set environment variables, add a map

```
{
  Name: LS,
  Version: 1.0,
  Executable: "/bin/ls",

  Environment: {
   "PATH": "/opt/myapp:/usr/bin:$PATH",
   "MYENV": "value",
  },

}
```

10.3.5 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 perform some pre- or postprocessing. These pre/post commands are executed on a login node (i.e. they are not part of the batch job).

```
{
 Name: SomeSimulation,
 Version: "1.0",
 Executable: "/usr/bin/simulation",

 PreCommand: "/opt/licenses/aquire_license",

 PostCommand: "/opt/licenses/release_license",

}
```

10.3.6 Prologue and epilogue

These commands will be executed as part on a batch node of the user's job script, and are placed before / after the application executable command.

```
{
 Name: SomeSimulation,
 Version: "1.0",
 Executable: "/usr/bin/simulation",

 Prologue: "module load some_module"

 Epilogie: "",

}
```

10.3.7 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:

```
{
  Name: SomeApp,
  Version: 1.0,

# instruct UNICORE to run the application on a login node

  RunOnLoginNode: true,

}
```

10.3.8 Exit code handing

By default, a UNICORE job will be set to NOT_SUCCESSFUL if the application exits with a non-zero exit code. If you want to change this behaviour, you can set a flog

```
{
  Name: SomeApp,
  Version: 1.0,

# instruct UNICORE to NOT fail if the application
# exits with non-zero exit code

  IgnoreNonZeroExitCode: true,

}
```

10.4 Application argument metadata

For client components it can be useful to have a description of an application in terms of its arguments. This allows for example the UNICORE Portal to automatically build a nice GUI for the application.

```
{
 Name: SomeApp,
 Version: 1.0,

 Parameters: {

  SOURCE:{
    Type: filename,
    Description: "The input file",
  },
```

```
  VERBOSE:{
    Type: boolean,
    Description: "Verbose mode",
  },

  PRECISION:{
    Type: choice,
    Description: "Computational precision",
    ValidValues: [
         "sloppy", "normal", "pedantic",
    ],
  },

 },

}
```

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 (lower/upper case does not matter) "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 `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.

10.4.1 Per-application resource requirements

If the application requires any default resources, like particular node constraints, or a specific queue, you can add resource requests in the IDB.

For example:

```
{
 Name: SomeSimulation,
 Version: "3.0",

 Resources: {
  Nodes: "2",
  NodeConstraints: "amd",
 }
}
```

Note that the user job can override these, i.e. if the user requests different values for the requested resources, the values from the user job will be used.

10.5 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 the XUUDB.

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.

10.5.1 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.

10.5.2 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>
```

10.5.3 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 10.6.

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 10.6 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 10.5 for details.

10.5.4 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.

10.5.5 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.

10.6 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
- `IncarnationDataBase idb` provides an (read only) access to the contents of the IDB. https://sourceforge.net/p/unicore/svn/HEAD/tree/xnjs/trunk/src/main/java/de/fzj/unicore/xnjs/idb/IDBImpl.java [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.

10.6.1 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"
```

10.6.2 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"})
```

10.6.3 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.

11 Data staging

When executing user jobs, the XNJS also performs data staging, i.e. getting data from remote locations before starting the job, and uploading data when the job has finished. A variety of protocols can be used for data movement, including UNICORE-specific protocols such as BFT or UFTP, but also standard protocols like ftp, scp, and e-mail.

Some of these (like mail) have additional configuration options, which are given in this section.

11.1 SCP support

UNICORE supports file staging in/out using SCP with username/password authentication. The source/target URL schema is "scp://"

11.1.1 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

```
# scp wrapper script
XNJS.staging.scpWrapper=/path/to/scp-wrapper.sh
```

11.1.2 SCP wrapper script

A possible 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 Python.

11.2 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.

The source/target URL scheme is "mailto:". You can append a subject, for example "mailto:user@domain?subject=Your output is ready"

11.2.1 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 XNJS config file). See the next section if you do not want to use an SMTP server directly.

- XNJS.staging.mailHost: the host of the SMTP server
- XNJS.staging.mailPort : the port of the SMTP server
- XNJS.staging.mailUser : the user name of the mail account which sends email
- XNJS.staging.mailPassword : the password of the mail account which sends email
- XNJS.staging.mailSSL : to use SSL, see the XNJS/TSI SSL setup page on how to setup SSL

11.2.2 Email wrapper script

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

```
# mailto wrapper script, defining this will disable JavaMail
XNJS.staging.mailSendScript=/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"
```

11.3 GridFTP

UNICORE can use GridFTP client tools for stage-in/stage-out provided the client uploads the required proxy certificate. The proxy cert is expected in a file ".proxy" in the job's working directory.

GridFTP usage can be customised using two settings in the XNJS config file ("xnjs.properties").

```
# name / path of the executable
XNJS.staging.gridftp=/usr/local/bin/globus-url-copy

# additional parameters for globus-url-copy
XNJS.staging.gridftpParameters=
```

11.4 Configuration reference

The configuration settings related to data staging are summarized in the following table.

Property name	Type	Default value / mandatory	Description
`XNJS.staging.curl`	string	-	Location of the *curl* executable used for FTP stage-ins. If null, Java code will be used for FTP.
`XNJS.staging.filesystemGraceTime`	integer >= 0	`10`	Grace time (in seconds) when waiting for files to appear on shared filesystems.
`XNJS.staging.gridftp`	string	`globus-url-copy`	Location of the *globus-url-copy* executable used for GridFTP staging.
`XNJS.staging.gridftpParameters`	string	*empty string*	Additional options for *globus-url-copy*.
`XNJS.staging.mailEnableSSL`	[true, false]	`false`	Outgoing mail (SMTP): enable SSL connection.
`XNJS.staging.mailHost`	string	`localhost`	Outgoing mail host (SMTP) used for staging-out via email.
`XNJS.staging.mailPassword`	string	-	Outgoing mail (SMTP) password.

Property name	Type	Default value / mandatory	Description
`XNJS.staging.mailPort`	integer number	`25`	Outgoing mail (SMTP) port number.
`XNJS.staging.mailScript`	string	-	Script to be used for sending outgoing mail (instead of using SMTP).
`XNJS.staging.mailUser`	string	-	Outgoing mail (SMTP) user name.
`XNJS.staging.scpWrapper`	string	`scp-wrapper.sh`	Location of the wrapper script used for scp staging.
`XNJS.staging.threads`	integer >= 1	`4`	Number of worker threads to use for data staging.
`XNJS.staging.wget`	string	-	Location of the *wget* executable used for HTTP stage-ins. If null, Java code will be used for HTTP.
`XNJS.staging.wgetParameters`	string	-	Additional options for *wget*.

12 UFTP setup

UFTP is a high-performance file transfer protocol. For using UFTP as a data staging and file upload/download solution in UNICORE, a separate server (uftpd) is required. This is installed on a host with direct access to the file system, usually this is a cluster login node, but it can also be a separate host.

In a UFTP transfer, one side acts as a client and the other side is the uftpd server. UNICORE/X will run the client code via the TSI (recommended) or in-process (with lower performance)

For details on how to install the uftpd server please refer to the separate UFTPD manual available on unicore.eu, which provides all information required to install and configure the UFTPD.

Note
If UFTPD is not running on the same host(s) as the TSI, you will need to copy the UTFPD libs and client executable to the TSI machine(s).

The minimal required UNICORE/X configuration consists of the listen and command addresses of the UFTPD server and the location of the client executable on the TSI host.

```
# Listener (pseudo-FTP) socket of UFTPD
coreServices.uftp.server.host=uftp.yoursite.edu
coreServices.uftp.server.port=64434
```

```
# Command socket of UFTPD
coreServices.uftp.command.host=uftp.yoursite.edu
coreServices.uftp.command.port=64435

# Full path to the 'uftp.sh' client executable
# installed on the TSI node
coreServices.uftp.client.executable=/usr/share/unicore/uftpd/bin/ ↩
    uftp.sh
```

If you want to run the client code in the UNICORE/X process, set

```
coreServices.uftp.client.local=true
```

The following table shows all the available configuration options for UFTP.

Property name	Type	Default value / mandatory	Description
`coreServices.uftp.buffersize`	integer >= 1	`128`	File read/write buffer size in kbytes.
`coreServices.uftp.client.executable`	string	`uftp.sh`	Configures the path to the client executable (location of *uftp.sh*) on the TSI.
`coreServices.uftp.client.host`	string	*not set*	Client host. If not set and UFTP client is set to local, then the local interface address will be determined at runtime. If not set and non-local mode is configured, then the TSI machine will be used.
`coreServices.uftp.client.ip_addresses`	string	*not set*	Client IP address(es) to send to UFTPD. If not set, the client.host value will be used.
`coreServices.uftp.client.local`	[true, false]	`false`	Controls whether, the Java UFTP client code should be run directly within the JVM, which will work only if the UNICORE/X has access to the target file system, or, if set to false, in the TSI.
`coreServices.uftp.command.host`	string	`localhost`	The UFTPD command host.
`coreServices.uftp.command.port`	integer [1..65535]	`64435`	The UFTPD command port.

Property name	Type	Default value / mandatory	Description
coreServices.uftp.commandSocketTimeout	integer [0 - 300]	10	The timeout (in seconds) for communicating with the command port.
coreServices.uftp.commandSslDisable	[true, false]	false	Allows to disable SSL on the command port (useful for testing).
coreServices.uftp.forceSessionMode	[true, false]	false	Controls multi-file transfers should be done one by one (NOT recommended).
coreServices.uftp.enable	[true, false]	true	Controls whether UFTP should be enabled for this server.
coreServices.uftp.encryption	[true, false]	false	Controls whether encryption should be enabled by default for server-server transfers.
coreServices.uftp.rateLimit	integer number	0	Limit the bandwidth (bytes per second) used by a single transfer (0 means no limit).
coreServices.uftp.server.host	string	-	UFTPD listen host. If this is not set, UFTP is disabled.
coreServices.uftp.server.port	integer [1 - 65535]	64434	UFTPD listen port.
coreServices.uftp.streams	integer number	1	Requested number of parallel data streams.
coreServices.uftp.streamsLimit	integer >= 1	4	Server limit for number of streams (per client connection).

12.1 Configuring multiple UFTPD servers

Since UNICORE 8.1, you can optionally configure multiple UFTPD servers that will then be used in a round-robin fashion, to increase performance and scalability.

The configuration is similar to the simple case, but you can have multiple "blocks" of servers.

As an example, consider this configuration of two UFTPD servers:

```
coreServices.uftp.1.server.host=uftp.yoursite.edu
coreServices.uftp.1.server.port=64434
coreServices.uftp.1.command.host=uftp.yoursite.edu
coreServices.uftp.1.command.port=64435

coreServices.uftp.2.server.host=uftp-2.yoursite.edu
```

```
coreServices.uftp.2.server.port=64434
coreServices.uftp.2.command.host=uftp-2.yoursite.edu
coreServices.uftp.2.command.port=64435

# Full path to the 'uftp.sh' client executable
# installed on the TSI node
coreServices.uftp.client.executable=/usr/share/unicore/uftpd/bin/ ↩
    uftp.sh
```

Use consecutive numbers (1,2, ...) to define servers.

13 Configuration of storages

A UNICORE/X server can make storage systems (e.g. file systems) accessible to users in several ways.

- storages endpoints can be defined which are available even if there is no compute service;
- storages can be "attached" to compute services;
- each job has a working directory, which is exposed as a storage instance and can be freely accessed using a UNICORE client.
- the "StorageFactory" service allows users to create dynamic storage instances, which is very useful if the UNICORE workflow system is used;

Storages have additional features which are covered in other sections of this manual.

- Metadata management is covered in Section 14
- Data-triggered processing is described in Section 15

13.1 Configuring storage services

Storage services are created on server startup and published in the registry. They are independent of any compute services and accessible for all users.

Note
Service accessibility does not imply file system accessibility. The file system access control is still in place, so users must have the appropriate Unix permissions to access a storage.

The basic property controls which storages are enabled

```
coreServices.sms.storage.enabledStorages=HOME WORK SHARE2 ...
```

Each enabled storage is configured using a set of properties.

Property name	Type	Default value / mandatory	Description
coreServices.sms.storage.N.allowUserDefinedDir	[true,false]	true	Whether the allow the user to set the storage base directory when creating the storage via the StorageFactory.
coreServices.sms.storage.N.checkExistence	[true,false]	true	Whether the existence of the base directory should be checked when creating the storage.
coreServices.sms.storage.N.class	Class extending de.fzj.unicore.uas.impl.sms.SMSBaseImpl	de.fzj.unicore.uas.impl.sms.SMSBaseImpl	Storage implementation class to be used (and mandatory) in case of the CUSTOM type.
coreServices.sms.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.sms.storage.N.defaultUmask	integer number	077	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.sms.storage.N.description	string	Filesystem	Description of the storage. It will be presented to the users. *(runtime updateable)*
coreServices.sms.storage.N.disableMetadata	[true,false]	false	Whether the metadata service should be disabled for this storage.

Property name	Type	Default value / mandatory	Description
coreServices.sms.storage.N.enableTrigger	[true,false]	false	Whether the triggering feature should be enabled for this storage.
coreServices.sms.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.sms.storage.N.infoProvider	Class extending de.fzj.unicore.uas.impl.sms.StorageInfoProvider	DefaultStorageInfoProvider	(Very advanced setting, only) Provider information about storages produced by the SMS factory.
coreServices.sms.storage.N.name	string	–	Storage name. If not set then the internal unique identifier is used.
coreServices.sms.storage.N.path	string	–	Denotes the storage base path.
coreServices.sms.storage.N.protocols	string	–	(DEPRECATED, ignored) *(runtime updateable)*
coreServices.sms.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.sms.storage.N.triggerUserID	string	–	For data triggering on shared storages, use this user ID for the controlling process.

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

For example to define a storage for accessing the user's HOME and some shared path

```
coreServices.sms.storage.HOME.name=HOME
coreServices.sms.storage.HOME.type=HOME
coreServices.sms.storage.HOME.description=User's HOME

coreServices.sms.storage.WORK.name=WORK
coreServices.sms.storage.WORK.description=Shared projects workspace
coreServices.sms.storage.WORK.path=/mnt/gpfs/projects
coreServices.sms.storage.WORK.defaultUmask=07
```

The *name* parameter will be used as the storage's service ID. This means that the URL to access these storages will be something like

```
https://<site_address>/rest/core/storages/HOME

https://<site_address>/rest/core/storages/WORK
```

and via the SOAP/XML interfaces

```
https://<site_address>/services/StorageManagement?res=HOME

https://<site_address>/services/StorageManagement?res=WORK
```

Usually, the "name" property is not needed, if you set it it should match the ID to avoid confusion.

The other storage properties (see the previous section) are also accepted!

13.2 Configuring storages attached to TargetSystem instances

Each TargetSystem instance can have one or more storages attached to it. Note that this is different case from the shared storages which are 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.allowUserDefinedPath`	[true, false]	`true`	Whether the allow the user to set the storage base directory when creating the storage via the StorageFactory.
`coreServices.targetsystem.storage.N.checkExistence`	[true, false]	`true`	Whether the existence of the base directory should be checked when creating the storage.
`coreServices.targetsystem.storage.N.class`	Class extending de.fzj.unicore.uas.impl.sms.SMSBaseImpl	-	Storage implementation class to load (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).

Property name	Type	Default value / mandatory	Description
`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.enableTriggers`	[true, false]	`false`	Whether the triggering feature should be enabled for this storage.
`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.infoProvider`	Class extending de.fzj.unicore.uas.impl.sms.StorageInfoProvider	`de.fzj.unicore.uas.impl.sms.DefaultStorageInfoProvider`	(Very) advanced setting, allows to provide information about storages produced by the SMS factory.
`coreServices.targetsystem.storage.N.name`	string	–	Storage name. If not set then the internal unique identifier is used.
`coreServices.targetsystem.storage.N.path`	string	–	Denotes the storage base path.
`coreServices.targetsystem.storage.N.protocols`	string	–	(DEPRECATED, ignored) *(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)*

Property name	Type	Default value / mandatory	Description
coreServices.targetsystem.storage.N.triggerUserID	string	-	For data triggering on shared storages, use this user ID for the controlling process.
coreServices.targetsystem.storage.N.type	string	-	Storage type. FIXEDPATH: mapped to a fixed directory, VARIABLE: resolved using an environmental variable lookup, CUSTOM: specified class is used.
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.2.name=DEISA_HOME
coreServices.targetsystem.storage.2.type=VARIABLE
coreServices.targetsystem.storage.2.path=$DEISA_HOMES

# example for a custom SMS implementation
coreServices.targetsystem.storage.3.name=MyStorage
coreServices.targetsystem.storage.3.type=CUSTOM
coreServices.targetsystem.storage.3.path=/
coreServices.targetsystem.storage.3.class=my.custom.sms. ↩
    ImplementationClass
```

13.2.1 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 than 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 because resource owners are expressed as grid user names (certificate DNs) and not xlogins. To have 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
```

13.3 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 S3 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.[type].allowUserDefinedDir`	[true, false]	`true`	Whether the allow the user to set the storage base directory when creating the storage via the StorageFactory.
`coreServices.sms.factory.[type].checkExistence`	[true, false]	`true`	Whether the existence of the base directory should be checked when creating the storage.
`coreServices.sms.factory.[type].class`	Class extending de.fzj.unicore.uas.impl.sms.SMSBaseImpl	—	Storage implementation class to be used (and mandatory) in case of the CUSTOM type.
`coreServices.sms.factory.[type].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)*

Property name	Type	Default value / mandatory	Description
coreServices.sms.factory.N.defaultUmask	integer number	077	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.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.enableTrigger	[true,false]	false	Whether the triggering feature should be enabled 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.infoProvider	Class extending de.fzj.unicore.uas.impl.sms.StorageInfoProvider	DefaultStorageInfoProvider	(Very) advanced setting, allowing to provide information about storages produced by the SMS factory.
coreServices.sms.factory.N.name	string	-	Storage name. If not set then the internal unique identifier is used.

Property name	Type	Default value / mandatory	Description
coreServices.sms.factory.N.path	string	-	Denotes the storage base path.
coreServices.sms.factory.N.protocols	string	-	(DEPRECATED, ignored) *(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)*
coreServices.sms.factory.N.triggerUserID	string	-	For data triggering on shared storages, use this user ID for the controlling process.
coreServices.sms.factory.N.type	string	-	Storage type. FIXEDPATH: mapped to a fixed directory, VARIABLE: resolved using an environmental variable lookup, CUSTOM: specified class is used.
coreServices.sms.factory.N.work-dir	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

# 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
```

```
# allow the user to pass in a path on storage creation. Defaults to ↩
    "true"
coreServices.sms.factory.TYPE1.allowUserDefinedPath=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.

It is also possible to let the user control the path of the dynamic storage, by sending a "path" parameter when creating the storage. For example, the user can use UCC to create a storage:

```
$> ucc create-sms path=/opt/projects/shared-data
```

This will create a storage resource for accessing the given directory. In this case, there will be no cleanup, and no appended storage ID.

The normal storage properties (see the previous section) are also accepted: "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.jclouds.s3. ↩
    S3StorageImpl
```

13.4 Configuring the job working directory storage services

For each UNICORE job instance, a storage instance is created, corresponding to the job's working directory. In some cases you might wish to control this storage in detail, e.g. configure a special storage backend.

The working directory storages are configured using a set of properties, which is the same as for the other storage types, except for the prefix.

Note
The "path", "name", "description", "enableTrigger" and "disableMetadata" properties are ignored, they are set by the server.

For example

```
coreServices.sms.jobDirectories.type=CUSTOM
coreServices.sms.jobDirectories.class=your.custom.SMSImpl
```

14 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

14.1 Configuring metadata support

By default, metadata support is enabled on all storages (except job directories).

You can disable it on a per-storage basis, see Section 13 for the relevant config settings.

You can also control which implementation should be used. This is done in *<CONF>/uas.config*.

```
#
# 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/
```

14.2 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
```

15 Data-triggered processing

UNICORE can be set up to automatically scan storages and trigger processing steps (e.g. submit batch jobs or run processing tasks) according to user-defined rules.

15.1 Enabling and disabling data-triggered processing

By default, data-triggered processing is disabled on all storages.

Explicit control is available via the configuration properties for storages, as listed in Section 13 Set the *enableTrigger* property to "true" to enable the data-triggered processing for the given storage.

15.2 Controlling the scanning process

To control which directories should be scanned, a file named `.UNICORE_Rules` at the top-level of the storage is read and evaluated. This file can be (and usually will be) edited and uploaded by the user.

The file must be in JSON format, and has the following elements:

```
{
"DirectoryScan": {

  "IncludeDirs": [
      "project.*",
  ],
  "ExcludeDirs": [
      "project42",
  ],
  "Interval": "30",

},

"Rules": [ ]

}
```

The "IncludeDirs" and "ExcludeDirs" are lists of Java regular expression strings that denote directories (as always relative to the storage root) that should be included or excluded from the scan.

The "Rules" section controls which files are to be processed, and what is to be done (actions). This is described below.

15.3 Special case: shared storages

Since shared storages are "owned" by the UNICORE server and used by multiple users, data-triggered processing requires a valid Unix user ID in order to list files independently of any actual user. Therefore the *triggerUserID* property is used to configure which user ID should be used (as always in UNICORE, this cannot be *root*, and multiuser operation requires the TSI!).

For example, you might have a project storage configured like this:

```
#
# Shares
#
coreServices.sms.storage.enabledStorages=PROJECTS

coreServices.sms.storage.PROJECTS.name=projects
coreServices.sms.storage.PROJECTS.description=Shared projects
coreServices.sms.storage.PROJECTS.path=/opt/shared-data
coreServices.sms.storage.PROJECTS.defaultUmask=007
coreServices.sms.storage.PROJECTS.enableTrigger=true
coreServices.sms.storage.PROJECTS.triggerUserID=projects01
```

Here the scanning settings are only evaluated top-level.

For each included directory, a separate scan is done, controlled by another `.UNICORE_Rules` file in that directory. So the directory structure could look like this:

```
&#x251c;&#x2500;&#x2500; dir1
&#x2502;~~ &#x251c;&#x2500;&#x2500; ...
&#x2502;~~ &#x2514;&#x2500;&#x2500; .UNICORE_Rules
&#x251c;&#x2500;&#x2500; dir2
&#x2502;~~ &#x251c;&#x2500;&#x2500; ...
&#x2502;~~ &#x2514;&#x2500;&#x2500; .UNICORE_Rules
&#x251c;&#x2500;&#x2500; dir3
&#x2502;~~ &#x251c;&#x2500;&#x2500; ...
&#x2502;~~ &#x2514;&#x2500;&#x2500; .UNICORE_Rules
&#x2514;&#x2500;&#x2500; .UNICORE_Rules
```

The top-level `.UNICORE_Rules` file must list the included directories. Processing the included directories is then done using the owner of that directory.

15.4 Rules

The "Rules" section in the `.UNICORE_Rules` file is a list of file match specifications together with a definition of an "action", i.e. what should be done for those files that match.

The general syntax is

```
{
"DirectoryScan": {
  "IncludeDirs": [...],
  "ExcludeDirs": [...]
},

"Rules": [
  {
    "Name": "foo",
    "Match": ".*incoming/file_.*",
```

```
    "Action": { ... }
   }
]
}
```

The mandatory elements are

- `Name` : the name of the rule. This is useful when checking the logfiles,
- `Match` : a regular expression defining which file paths (relative to storage root) should be processed,
- `Action` : the action to be taken.

15.4.1 Variables

The following variables can be used in the `Action` description.

- `UC_BASE_DIR` : the storage root directory
- `UC_CURRENT_DIR` : the absolute path to the parent directory of the current file
- `UC_FILE_PATH` : the full path to the current file
- `UC_FILE_NAME` : the file name

15.4.2 Scripts

This type of action defines a script that is executed on the cluster login node (TSI node).

```
"Action":
{
 "Name": "local_example",
 "Type": "LOCAL",
 "Command": "/bin/md5sum ${UC_FILE_PATH}",
 "Outcome": "output_directory",
 "Stdout": "${UC_FILE_NAME}.md5",
 "Stderr": "${UC_FILE_NAME}.error"
}
```

15.4.3 Batch jobs

This type of action defines a batch job that is submitted to the resource management system of your cluster.

```
"Action":
{
 "Name": "batch_example",
 "Type": "BATCH",
 "Job": { ... }
}
```

The `Job` element is a normal UNICORE job in the same syntax as used for the UCC commandline client.

15.4.4 Automated metadata extraction

```
"Action":
{
 "Name": "extract_example",
 "Type": "EXTRACT",
 "Settings": { ... }
}
```

This action will extract metadata from the file. The `Settings` element is currently unused.

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

16.1 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:

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

16.2 Available PDP modules

16.2.1 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:

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

16.2.2 Remote SAML/XACML 2.0 PDP with Argus PAP

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:

```
container.security.accesscontrol.pdpConfig=<CONFIG_DIR>/argus. ↩
    config
container.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.

17 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 Unity. 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 uses version 2.0.

UNICORE allows to choose one of several authorization back-end implementations called Policy Decision Points (PDP). The authorization section Section 16 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 PDP. 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.

17.1 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 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
```

17.2 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.

17.3 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>
```

17.4 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:

```
logger.sec.name=unicore.security
logger.sec.level=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.

18 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.

18.1 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`:

```
coreServices.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

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

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

19 Cloud storages support (S3, Swift, CDMI)

UNICORE/X can use S3, Swift or CDMI storages as backend. These storages can be configured both as a normal storage (shared or attached to target systems) and as storage backend for the StorageFactory service (see also Section 13)

19.1 Basic configuration

Configuring a cloud storage as a shared storage works exactly as described in Section 13, you just have to make sure to use the required properties.

The following sections list the required properties for all of the supported cloud storages. Note that the prefix depends on what type of storage (shared, dynamic, TSS, job working directory) is being configured.

19.1.1 S3

```
<prefix>.type=CUSTOM
<prefix>.class=de.fzj.unicore.uas.jclouds.s3.S3StorageImpl
<prefix>.infoProviderClass=de.fzj.unicore.uas.jclouds.s3. ↩
    S3InfoProvider
```

```
# provider is "s3" or "aws-s3"
<prefix>.settings.provider=s3

# http(s) URL of the S3 storage
<prefix>.settings.endpoint=...

# authentication keys
<prefix>.settings.accessKey=...
<prefix>.settings.secretKey=...

# may the user set the endpoint (default: false)
<prefix>.settings.allowUserDefinedEndpoint=false
```

19.1.2 Swift

```
<prefix>.type=CUSTOM
<prefix>.class=de.fzj.unicore.uas.jclouds.swift.SwiftStorageImpl
<prefix>.infoProviderClass=de.fzj.unicore.uas.jclouds.swift. ↩
    SwiftInfoProvider

# http(s) URL of the Swift storage
<prefix>.settings.endpoint=...

# authentication username/password
<prefix>.settings.username=...
<prefix>.settings.password=...

# allow the user to set the endpoint
<prefix>.settings.allowUserDefinedEndpoint=true
```

19.1.3 CDMI

```
<prefix>.type=CUSTOM
<prefix>.class=de.fzj.unicore.uas.cdmi.CDMIStorageImpl
<prefix>.infoProviderClass=de.fzj.unicore.uas.cdmi.CDMIInfoProvider

# http(s) URL of the CDMI storage
<prefix>.settings.endpoint=...

# authentication username/password
<prefix>.settings.username=...
<prefix>.settings.password=...

# Openstack Keystone token endpoint
# if not set, HTTP basic authentication will be used
<prefix>.settings.tokenEndpoint=...
```

```
# allow the user to set the endpoint
<prefix>.settings.allowUserDefinedEndpoint=true
```

Compare the examples below! The authentication keys can be handled flexibly, as detailed in the next section.

19.2 Authentication credentials

There are several ways to configure the required credentials for authenticating the user to the cloud store. Two of them are done server-side (i.e. by the UNICORE administrator) and the third uses the credentias provided by the user.

UNICORE/X looks for credentials in the following order

- credentials provided by the user
- per-user credentials provided via UNICORE's attribute sources
- fixed credentials provided in the server config

It is always possible for the user to pass in credentials when creating the storage using the StorageFactory service. Of course this mechanism does not apply when using a cloud store for a different type of storage service.

The second option (using attribute sources) allows to configure per-user credentials, but managing everything server-side, so the user has a nice single-sign-on experience when using UNICORE.

If you use the map file attribute source, an example entry looks like this:

```
<entry key="CN=Demo User,O=UNICORE,C=EU">

   <!-- standard UNICORE attributes -->

   <attribute name="role">
      <value>user</value>
   </attribute>
   <attribute name="xlogin">
      <value>somebody</value>
   </attribute>
   <attribute name="group">
      <value>users</value>
   </attribute>

    <!-- S3 specific attributes -->

    <attribute name="s3.accessKey">
       <value> ... access key data omitted ... </value>
```

```
      </attribute>
      <attribute name="s3.secretKey">
         <value> ... secret key data omitted ... </value>
      </attribute>

   </entry>
```

Last not least, the keys can also be hardcoded into the config, using the accessKey and secretKey properties.

```
# authentication keys
<prefix>.settings.accessKey=...
<prefix>.settings.secretKey=...
```

19.3 Examples

19.3.1 Dynamic storage using the StorageFactory

If configured as a dynamic storage, a new directory will be created corresponding to each storage.

In the following example we configure S3 in addition to the "DEFAULT" storage type.

```
#
# Available storage types
#
coreServices.sms.factory.storagetypes=DEFAULT S3

#
# NOTE
#
# the configuration for the "DEFAULT" storage type
# is OMITTED in this example!
#

#
# S3 storage configuration
#
coreServices.sms.factory.S3.description=S3 interface
coreServices.sms.factory.S3.type=CUSTOM
coreServices.sms.factory.S3.class=de.fzj.unicore.uas.jclouds.s3. ↩
    S3StorageImpl
coreServices.sms.factory.S3.infoProviderClass=de.fzj.unicore.uas. ↩
    jclouds.s3.S3InfoProvider
coreServices.sms.factory.S3.path=/dynamic-storages
coreServices.sms.factory.S3.cleanup=false
coreServices.sms.factory.S3.protocols=BFT
```

```
#
# the next four settings depend on your S3 backend
#

# provider is "s3" or "aws-s3"
coreServices.sms.factory.S3.settings.provider=s3
# endpoint of the S3
coreServices.sms.factory.S3.settings.endpoint=...
# OPTIONAL access key and secret key
coreServices.sms.factory.S3.settings.accessKey=...
coreServices.sms.factory.S3.settings.secretKey=...

# optional: may user overwrite endpoint and provider?
# this defaults to 'false'!
coreServices.sms.factory.S3.settings.allowUserdefinedEndpoint=true
```

19.3.2 Shared storage

```
# add 'S3' to the list of enabled shares
coreServices.sms.storage.enabledStorages=S3 ...

# S3 configuration
coreServices.sms.storage.S3.description=S3 interface
coreServices.sms.storage.S3.type=CUSTOM
coreServices.sms.storage.S3.class=de.fzj.unicore.uas.jclouds.s3. ↩
    S3StorageImpl
coreServices.sms.storage.S3.infoProviderClass=de.fzj.unicore.uas. ↩
    jclouds.s3.S3InfoProvider
coreServices.sms.storage.S3.path=/
coreServices.sms.storage.S3.protocols=BFT

coreServices.sms.storage.S3.settings.provider=s3
coreServices.sms.storage.S3.settings.endpoint=...
coreServices.sms.storage.S3.settings.accessKey=...
coreServices.sms.storage.S3.settings.secretKey=...
```

Configuring as a TSS storage works accordingly.