UNICORE Docs

2023 UNICORE

Apr 11, 2024

UNICORE DOCUMENTATION

UNICORE (UNiform Interface to COmputing REsources) offers a ready-to-run system including client and server software. It makes distributed computing and data resources available in a seamless and secure way in intranets and the internet.

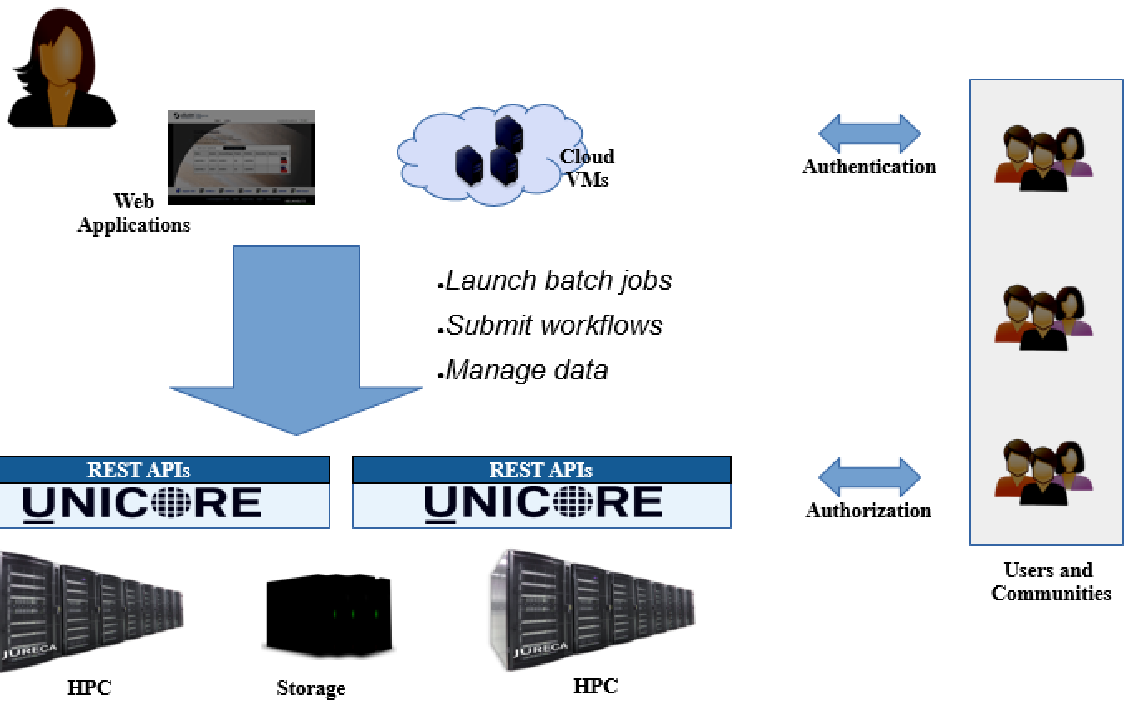


Fig. 1: Federating HPC with UNICORE

- *Overview* gives an overview of the UNICORE features and the UNICORE architecture.
- *Getting started* shows how to get going quickly.
- *How to setup UNICORE for a single HPC cluster* covers the steps required to install a minimal set of UNICORE services for a single HPC cluster that is running Slurm.

CHAPTER

ONE

OVERVIEW

UNICORE (UNiform Interface to COmputing REsources) provides tools and services for building federated systems, making high-performance computing and data resources accessible in a seamless and secure way for a wide variety of applications in intranets and the internet.

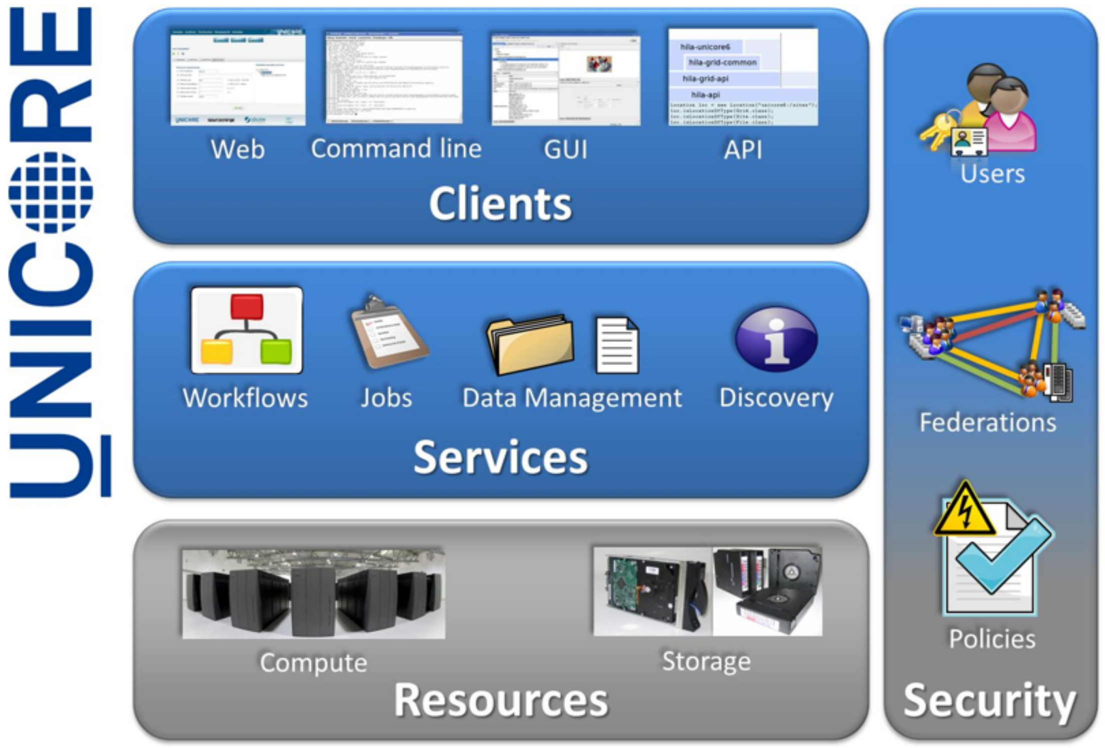


Fig. 1.1: UNICORE Architecture

1.1 UNICORE Features

UNICORE provides a comprehensive set of *RESTful APIs* for HPC access and workflows, dealing with user authentication, user account mapping and authorization in a highly flexible way.

1.1.1 Services and APIs

- Batch jobs with pre- and post-processing
- Support for common resource managers such as SLURM or LSF
- File system access and file transfer
- Site-to-site file transfer
- Cross-site workflows featuring graphs, loops, conditions, variables, hold/continue, workflow data management
- Direct access to applications running on HPC (e.g. for steering or visualisation)
- Metadata
- Rule based file processing
- Service Registry

1.1.2 Security

- Flexible user authentication: username/password, OpenID Connect, SSH keys, X.509, …
- Flexible mapping of users to local accounts and groups
- Based on open standards: X.509 Public Key Infrastructure, TLS, SAML, OIDC, XACML, …

1.1.3 Clients

- *Commandline client*: Job execution, data transfer, workflows, scripting, batch mode, extensible
- Dedicated client for UFTP high performance file transfer and data management features
- pyUNICORE Python client library

1.1.4 Add-ons

- Standalone UFTP suite for high-performance data transfer (can be used independently of UNICORE)
- Unity Identity Management system, supports LDAP, OAuth, SAML, federated AAI and a lot more

CHAPTER

TWO

GETTING STARTED

2.1 Using UNICORE

If you are an end-user or application developer who wishes to use an existing UNICORE installation, have a look at the user documentation for the *UNICORE Commandline Client*, the PyUNICORE client library or the *REST API documentation*.

2.2 Evaluating UNICORE

If you wish to experiment with a UNICORE server installation or wish to quickly evaluate UNICORE's features, you can try our UNICORE Docker image.

Also, you can download the Core Server Bundle which can be installed very quickly on a single test machine or even your laptop.

2.3 Deploying UNICORE

Full production deployments of UNICORE range from minimalistic to rather complex, depending on your requirements, use cases and existing infrastructure.

A few starting points:

1. for each target resource (e.g. a compute cluster) you need a *TSI* and a *UNICORE/X* The TSI is deployed on the cluster login node(s), while UNICORE/X requires a VM or server, UNICORE/X should NOT be run on a machine where users can log in.
2. we strongly recommend running a *Gateway*, one for all of an institution/company's UNICORE services is enough. This will shield the services from direct external access for added security.
3. For the services (except TSI where this is optional), you will need server certificate(s) from a CA (similar to a web server)
4. for multi-site workflows, you will need a *Registry* and a *Workflow service*

You can always *contact us* for advice in your specific situation.

CHAPTER

THREE

HOW TO SETUP UNICORE FOR A SINGLE HPC CLUSTER

3.1 Overview

This How-To covers in detail the steps required to install a minimal set of UNICORE services for a single HPC cluster that is running Slurm.

The following steps will be described:

- setup the Slurm TSI on a HPC login node
- deploy UNICORE Gateway and UNICORE/X on one VM or physical server
- connect UNICORE/X and TSI
- add test user(s)
- make a Slurm queue accessible via UNICORE
- test the installation via `curl`
- replace the *demo* certificate by a more secure self-signed certificate for Gateway and UNICORE/X

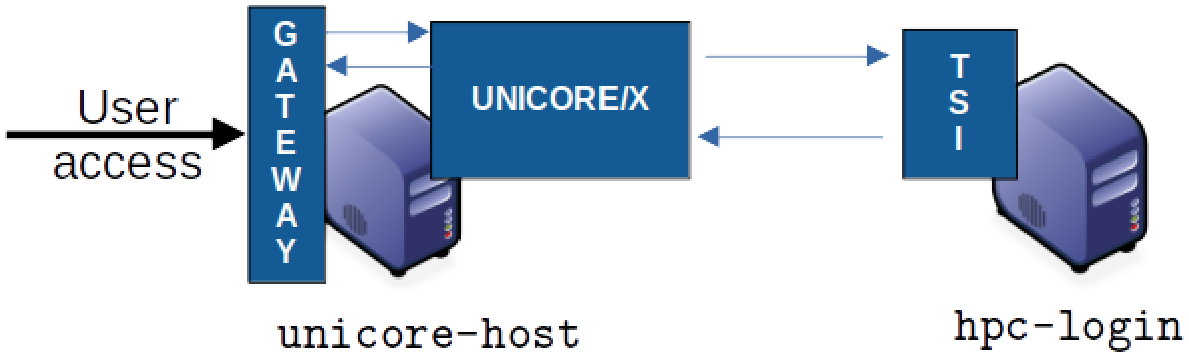


Fig. 3.1: Example deployment for a single cluster

3.2 Prerequisites

- a server or VM with Java 11 or later and Python3 installed
- port 8080 on this server must be accessible from the Internet if you want to let external users access your cluster.

In the following, we will refer to this machine as `unicore-host`. Adapt the following code examples according to your actual machine name.

- access to the HPC login node(s) with Python3 installed. Since UNICORE requires a TCP connection from the `unicore-host` to the login node(s), and another TCP connection from the login node(s) to the `unicore-host`, the local firewall rules might need to be adapted accordingly. Details will be given below.

3.3 Installing the TSI

The TSI is a server daemon written in Python and is installed on (one or more) HPC login nodes.

The HPC login node will be named `hpc-login` in the following.

3.3.1 Preparations

- SSH as `root` into the login node:
- Add a `unicore` user and a directory for the UNICORE TSI:

```
/usr/sbin/groupadd -r unicore
/usr/sbin/useradd -c "UNICORE" -g unicore -s /bin/false -r -d /tmp unicore
mkdir -p /opt/unicore
chown unicore:unicore /opt/unicore
```

3.3.2 Download and install the Slurm TSI

```
cd /opt/unicore

wget https://sourceforge.net/projects/unicore/files/Servers/Core/9.3.1/unicore-tsi-9.3.1.
↪tgz -O unicore-tsi-9.3.1.tgz

tar xf unicore-tsi-9.3.1.tgz
rm unicore-tsi-9.3.1.tgz
cd unicore-tsi-9.3.1

./Install.sh slurm /opt/unicore/tsi

chown -R unicore:unicore /opt/unicore/tsi

cd /opt/unicore/tsi/conf

# log to file in /opt/unicore/tsi/logs/ instead of syslog
sed -i "s/use_syslog=.*/use_syslog=0/" tsi.properties
```

(continues on next page)

(continued from previous page)

```
# configure the hostname of the UNICORE/X machine
sed -i "s/unicorex_machine=.*/unicorex_machine=unicore-host/" tsi.properties
```

You can start the TSI now and see if there are any errors in the log file:

```
rm -f /opt/unicore/tsi/logs/*
/opt/unicore/tsi/bin/start.sh
cat /opt/unicore/tsi/logs/TSILog_*
```

3.4 Installing the UNICORE Gateway and UNICORE/X

The two Java-based server components will be installed on the host named `unicore-host`. We assume that Java 11 or later is installed.

Check that

```
java --version
```

works and shows the correct version.

After this step, the UNICORE installation will be accessible at

`https://unicore-host:8080/TEST/rest/core`.

3.4.1 Preparations

- SSH into `unicore-host`
- Add a `unicore` user and a directory for the UNICORE components:

```
/usr/sbin/groupadd -r unicore
/usr/sbin/useradd -c "UNICORE" -g unicore -s /bin/false -r -d /tmp unicore
mkdir -p /opt/unicore
chown unicore:unicore /opt/unicore
```

3.4.2 Download and extract the UNICORE Server bundle

```
cd /opt/unicore

wget https://sourceforge.net/projects/unicore/files/Servers/Core/9.3.1/unicore-servers-9.
↪3.1.tgz -O unicore-servers-9.3.1.tgz

tar xf unicore-servers-9.3.1.tgz
chown -R unicore:unicore unicore-servers-9.3.1
rm unicore-servers-9.3.1.tgz
```

3.4.3 Installing UNICORE Gateway and UNICORE/X

We will install the Gateway to `/opt/unicore/gateway` and UNICORE/X to `/opt/unicore/unicorex` using the *configure.py* and *install.py* scripts.

Here we can already set the TSI host (`hpc-login`) and configure the Gateway to listen on all addresses and route requests to UNICORE/X under the *TEST* alias.

We also tell UNICORE/X what the public hostname of the UNICORE installation will be (`unicore-host`) and where the job directories should be created on the HPC cluster.

```
cd unicore-servers-9.3.1

# TSI is running on 'hpc-login'
sed -i "s/uxTSIHost=.*/uxTSIHost=hpc-login/" configure.properties

# Public address is 'unicore-host'
sed -i "s/uxGatewayHost=.*/uxGatewayHost=unicore-host/" configure.properties

sed -i "s/uxName=.*/uxName=TEST/" configure.properties

sed -i 's%uxTSIWorkingDirectoriesBasedir=.*%uxTSIWorkingDirectoriesBasedir=$HOME/UNICORE_
↪Jobs%' configure.properties

sed -i "s/tsi=true/tsi=false/" configure.properties

sed -i "s/gwHost=.*/gwHost=0.0.0.0/" configure.properties

sed -i "s%INSTALL_PATH=currentdir%INSTALL_PATH=/opt/unicore%" configure.properties

# setup the configuration files and copy the required files to '/opt/unicore'

sudo -u unicore ./configure.py
sudo -u unicore ./install.py
```

3.4.4 Starting the Gateway

The Gateway files can now be found in `/opt/unicore/gateway` and the server is started like this:

```
cd /opt/unicore/gateway

sudo -u unicore bin/start.sh
```

Logs are in `/opt/unicore/gateway/logs`.

To check for any errors:

```
cat /opt/unicore/gateway/logs/gateway.log | grep ERROR
```

The Gateway should now be accessible. A simple test using `curl` would be:

```
curl -k -i https://unicore-host:8080
```

(which will return some HTML)

3.5 UNICORE/X configuration

The UNICORE/X files are now in `/opt/unicore/unicorex`.

UNICORE/X is the central component in a UNICORE installation, and consequently has quite a few configuration options.

Here we focus on a very basic setup, and refer to the *full manual* for more information.

3.5.1 Connecting UNICORE/X and TSI

This part is configured in the file `/opt/unicore/unicorex/conf/tsi.config`.

UNICORE/X and TSI communicate via TCP. There are two connections:

1. From the UNICORE/X host to the TSI (HPC login node) on port `4433`
2. From the HPC login node to the UNICORE/X host on port `7654`

Make sure your firewall(s) allow both these connections.

3.5.2 Starting UNICORE/X

The UNICORE/X server is started like this:

```
cd /opt/unicore/unicorex

sudo -u unicore bin/start.sh
```

Logs are in `/opt/unicore/unicorex/logs`.

To check for any errors:

```
cat /opt/unicore/unicorex/logs/startup.log | grep ERROR
cat /opt/unicore/unicorex/logs/unicorex.log | grep ERROR
```

As a first check via the REST API, you can run

```
curl -k -H "Accept: application/json" https://unicore-host:8080/TEST/rest/core | python3␣
↪-m json.tool
```

3.5.3 User authentication

To understand the security concepts in UNICORE, please read *this section* in the UNICORE/X manual.

In the configuration we have set up so far, UNICORE will authenticate users via username/password, which are configured in a file

`/opt/unicore/unicorex/conf/rest-users.txt`

A default user *demouser* with password *test123* is pre-configured, you can add others.

Many other options for authentication exist, and we can only refer to the *Authentication section* in the UNICORE/X manual.

3.5.4 User account mapping

In the configuration we have set up so far, users are mapped to HPC accounts in the file

`/opt/unicore/unicorex/conf/simpleuudb`

Make sure to add account mappings for your users there.

Other options for account mapping exist, we refer to the *Attribute sources section* in the UNICORE/X manual.

3.5.5 Setting up batch queues

The available batch system queues are configured in the file

`/opt/unicore/unicorex/conf/idb.json`

A partition named `batch` is already in there, make sure to have a look and adapt it to your needs.

For more information, we refer to the *IDB syntax section* in the UNICORE/X manual.

3.6 Testing

3.6.1 Authentication and user mapping

To check that the authentication and user mapping works as intended, you can run the following

```
export BASE=https://unicore-host:8080/TEST/rest/core

curl -k -u demouser:test123 -H "Accept: application/json" $BASE?fields=client | python3 -
↪m json.tool
```

where the output will look approximately like this

```
{
  "client": {
      "role": {
          "selected": "user",
          "availableRoles": [
              "user"
          ]
      },
      "authenticationMethod": "PASSWORD_FILE",
      "dn": "CN=Demo User, O=UNICORE, C=EU",
      "xlogin": {
          "UID": "demouser",
          "availableGroups": [],
          "availableUIDs": [
              "demouser"
          ]
      }
  }
}
```

3.6.2 Batch queue setup

To check the available batch queues,

```
export BASE=https://unicore-host:8080/TEST/rest/core

curl -k -H "Accept: application/json" $BASE/factories/default_target_system_factory?
↪fields=resources | python3 -m json.tool
```

which will look similar to this

```
{
  "resources": {
      "batch": {
          "CPUsPerNode": "1-4",
          "Runtime": "10-86000",
          "MemoryPerNode": "1048576-1073741824",
          "Nodes": "1-16",
          "TotalCPUs": "1-64"
      }
  }
}
```

3.6.3 Test job

Create a file *test1.json* with the following content

```
{
  "Executable": "date"
}
```

and submit it using

```
export BASE=https://unicore-host:8080/TEST/rest/core

curl -k -i -u demouser:test123 -H "Content-Type: application/json" --data-ascii @test1.
↪json  $BASE/jobs
```

which should result in something like this

```
HTTP/1.1 201 Created
Date: Tue, 04 Jul 2023 09:59:38 GMT
X-Frame-Options: DENY
Content-Type: application/json;charset=utf-8
X-UNICORE-SecuritySession: 192ae773-650b-45bf-93fb-5552739f5460
X-UNICORE-SecuritySession-Lifetime: 28799354
Location: https://unicore-host:8080/TEST/rest/core/jobs/78b1a586-3f66-4f5b-bb8d-
↪7fe1d8fe7b87
Transfer-Encoding: chunked
```

Check the UNICORE/X logs in case of errors. To check whether the job runs properly, check the logs. You can also access the job via the REST API, the URL to use is given in the `Location` field above

```
export JOB=https://unicore-host:8080/TEST/rest/core/jobs/78b1a586-3f66-4f5b-bb8d-
↪7fe1d8fe7b87

curl -k -u demouser:test123 -H "Accept: application/json" $JOB | python3 -m json.tool
```

3.6.4 Further testing

We recommend downloading the *UNICORE commandline client*, or using PyUNICORE for further tests.

3.7 Server certificate

Up to now, the so-called *demo certificates* that come with the download have been used. While this is OK for testing and setup, it is VERY BAD to expose such a server to the outside world, since anyone who knows what they are doing can easily get access to your installation.

Ideally you will get an SSL certificate from a **CA** (**C**ertification **A**uthority) for your machine and use that. It's however beyond the scope of this how-to to give a full introduction to SSL certificates.

As an improvement over the demo certificates, we will create a so-called *self-signed certificate* and use that, which is secure enough to expose the system to outside users, but is usually not good enough when integrating UNICORE access with external applications, or integrating your UNICORE installation into a bigger setup or federation.

3.7.1 Generating the self-signed certificate

The following uses OpenSSL to create a self-signed certificate

```
cd /opt/unicore/certs

openssl req -x509 -newkey rsa:4096 -sha256 -nodes -days 3650 \
    -keyout server-key.pem   \
    -out server-cert.pem     \
    -subj "/C=EU/O=Test/CN=unicore-host"

chown unicore:unicore server-*.pem

cat server-cert.pem >> server-key.pem
```

The file *server-key.pem* is now suitable as server credential, and the *server-cert.pem* will be used as the server truststore. We will use the same key and cert for both UNICORE/X and Gateway.

3.7.2 Gateway config

We configure our new credential and trusted certificate in the file */opt/unicore/gateway/conf/gateway.properties*:

```
cd /opt/unicore/gateway/conf

sed -i "s%credential.path=.*%credential.path=/opt/unicore/certs/server-key.pem%" gateway.
↪properties
sed -i "s%credential.password=.*%credential.password=%" gateway.properties

sed -i "s%directoryLocations.1=.*%directoryLocations.1=/opt/unicore/certs/server-cert.pem
↪%" gateway.properties
```

Restart via:

```
cd /opt/unicore/gateway
bin/stop.sh
sudo -u unicore bin/start.sh
```

Check the logs for any errors!

3.7.3 UNICORE/X config

We configure our new credential and trusted certificate in the file */opt/unicore/unicorex/conf/container.properties*:

```
cd /opt/unicore/unicorex/conf

sed -i "s%credential.path=.*%credential.path=/opt/unicore/certs/server-key.pem%"␣
↪container.properties
sed -i "s%credential.password=.*%credential.password=%" container.properties

sed -i "s%directoryLocations.1=.*%directoryLocations.1=/opt/unicore/certs/server-cert.pem
↪%" container.properties
```

Restart via:

```
cd /opt/unicore/unicorex
bin/stop.sh
sudo -u unicore bin/start.sh
```

Check the logs for any errors!

3.8 Getting support

You can always *contact us* for advice in your specific situation.

CHAPTER

FOUR

USER DOCUMENTATION

- *Introduction* Introduction for UNICORE users.
- *UNICORE Commandline Client* A full featured commandline client for UNICORE.
- *REST API* REST-API for the *UNICORE/X server* (job submission and management, data access and data transfer) and the *Workflow server* (workflow submission and management).
- *Job description format* The job description format that allows you to specify the application or executable you want to run, arguments and environment settings, any files to stage in from remote servers and any result files to stage out.
- *Workflow description* The workflow description language that is supported by the *Workflow engine*.
- *Data-triggered processing* Reference for the data-triggered processing in UNICORE/X.

4.1 Introduction

This document gives an introduction for new UNICORE users - what can UNICORE do for you?

4.1.1 Why use UNICORE?

The main functionality offered by UNICORE is a secure and flexible programmatic access to HPC compute and storage.

You can use UNICORE for all those tasks where the “usual” SSH access is not flexible or not secure enough:

- automation tasks
- multi-step and even multi-site workflows
- hybrid cloud/HPC applications
- integration into third party (web) applications

4.1.2 What client should you use?

For most of the common end-user tasks, like interactive use, scripting, automation etc, the *UNICORE Commandline client* will be the most convenient. It provides a fully-fledged commandline tool.

For integration into third party applications, we provide a Python library PyUNICORE.

If you are not using Python (or Java), you can always directly use the *REST API*.

4.1.3 Use case: job execution

Running jobs is the backbones of UNICORE - you can use it as a fancy job submission and job monitoring tool, or simply as a "replacement" for SSH.

4.1.3.1 Executing commands via UNICORE

To simply run a command on a login node of the HPC cluster, you can use the `ucc exec` command

```
ucc exec -- whoami
```

This will run the command and get standard output and error back to the client and display them.

For more complex cases, you can create a JSON job description and run that via UCC.

```
{
  "Job type": "ON_LOGIN_NODE",
  "Executable": "/bin/bash ./myscript.sh",
  "Imports": [
    {
      "From": "inline://dummy", "To": "myscript.sh",
      "Data": [
        "whoami",
        "hostname",
        "date"
      ]
    }
  ]
}
```

which you can then run via `ucc run`.

For simplicity, this example puts the script directly in the job via an "inline" data transfer. There's a number of other options available to deal with file transfers.

4.1.3.2 Running batch jobs

The most common use case on a HPC system is the batch job - your job gets submitted to the cluster's queueing system (e.g. Slurm), and is executed whenever the required resources become available.

You as the user need to provide the required resources - which queue, how long do you need the job to run, how many nodes etc, as well as the command to execute.

For example to run 4 instances of "date" on one node of a cluster, the job would look like this:

```
{
  "Executable": "srun -ntasks=4 date",
  "Resources": {
    "Nodes": 1,
    "Runtime": 30
  }
}
```

Running this job via `ucc run` will submit and monitor the job, waiting for its completion and then download the standard output and error files. UCC has many options to modify this behaviour, and you will often submit the job without waiting for it to finish.

The `ucc list-jobs` command is used to list all your jobs (that were submitted via UNICORE), and you can use other ucc commands to interact with the job or download results.

4.1.3.3 Advanced batch jobs

If you prefer to use a more low level way to allocate resources, you can provide a file containing resource requests, e.g. for Slurm, and tell UNICORE to use that via special "Job type" and "BSS file" elements in your job:

```
{
  "Job type": "RAW",
  "BSS file": "sbatch.request",

  "Executable": "srun -ntasks=4 date",

  "Imports": [
    {
      "From": "inline://dummy", "To": "sbatch.request",
      "Data": [
        "#!/bin/bash",
        "#SBATCH --account=yourproject",
        "#SBATCH --nodes=1",
        "#SBATCH --output=stdout",
        "#SBATCH --error=stderr",
      ]
    }
  ]
}
```

For simplicity, this example contains the script directly in the job description via an "inline" data transfer.

Note that this only needs to contain resource requests, the actual execution part will be document by UNICORE. UNICORE will then track this batch job as usual.

4.2 UNICORE Commandline Client

The **U**NICORE **C**ommandline **C**lient (UCC) is a client for *UNICORE*, supporting all of UNICORE's features, such as:

- *Job submission and management*
- *Data access and management* (upload, download, server-to-server copy, etc) using the UNICORE storage management functions and the available data transfer protocols
- Storage functions (*ls*, *mkdir*, ...) including creation of storage instances via storage factories
- *Workflow execution* using UNICORE's own *workflow description*

and much more.

UCC Manual User Manual with detailed instructions and examples for using the UNICORE Commandline Client.

Building the UCC Building the UNICORE Commandline Client distribution packages.

4.2.1 UCC Manual

4.2.1.1 Overview

The **U**NICORE **C**ommandline **C**lient (UCC) is a full-featured client for the UNICORE middleware. UCC has client commands for all the UNICORE basic services and the *UNICORE workflow system*.

It offers the following functions

- *Job submission and management*
- *Batch mode* job submission and processing with many *performance tuning* options
- *Data movement* (upload, download, server-to-server copy, etc) using the UNICORE storage management functions and available data transfer protocols
- Storage functions (*ls*, *mkdir*, ...) including *creation of storage instances* via storage factories
- Support for UNICORE *workflow submission and management*
- Support for the *UNICORE metadata* system
- Support for *sharing UNICORE resources* via ACLs
- Information about the available services is provided via the *system-info* command
- Various utilities like a *shell mode*, *low-level REST API* operations and others
- Extensibility through custom commands and the possibility to *run scripts* written in the Groovy programming language
- *Built-in help*

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

4.2.1.2 Installation and configuration

4.2.1.2.1 Prerequisites

To run UCC, you need a Java runtime version 11 or later (OpenJDK preferred).

4.2.1.2.2 Download

You can get the UCC latest version from SourceForge UNICORE download page.

4.2.1.2.3 Installation and configuration

To install, unpack the distribution in a directory of your choice. It's a good idea to add the `bin/` directory to your `PATH` variable,

```
$ export PATH=$PATH:<UCC_HOME>/bin
```

where *UCC_HOME* is the directory you installed UCC in.

Note: *Windows only*

Please do not install UCC into a directory containing spaces such as *Program files*.

Also avoid long path names, this can lead to errors due to the Windows limit on command line length.

Setting environment variables can be done (as *administrator*) using the *Control panel* → *System* → *Extras panel*.

Though you can specify many parameters on the commandline, it is easiest to use a config file, so that you do not have to key in this information repeatedly.

4.2.1.2.4 Preferences file

UCC checks by default whether the file `<userhome>/.ucc/preferences` exists, and reads it.

A minimal example that specifies username, password and your preferred UNICORE registry URL would look as follows:

```
registry=<your registry>

authentication-method=username
username=demouser
password=test123

truststore.type=directory
truststore.directoryLocations.1=<path to CA file(s)>

client.serverHostnameChecking=NONE
```

Please refer to *Common options to UCC* for a full description of available options.

Note: If you are worried about security, and do not want specify the password: UCC will ask for it if it is not given in the preferences or on the commandline.

Note: *Windows only*

The preferences are usually searched in the `c:\Users\<user_name>\.ucc` folder.

To create the `.ucc` folder, you might have to use the command prompt `mkdir` command.

When specifying paths in the preferences file, the backslash \ character needs to be written using an extra backslash \\.

For example, if you are using a local UNICORE installation for testing, you could use

```
registry=https://localhost:8080/DEMO-SITE/rest/core/registries/default_registry
```

Tip: If you wish to change the default property file location, you can set a Java VM property in the UCC start script, for example by editing the command that starts UCC

```
$ java .... -Ducc.preferences=<preferences location> ....
```

4.2.1.2.5 Logging

UCC writes some messages to the console, more if you choose the verbose mode (`-v` option). If you need real logging (e.g. when using the batch mode), you can edit the `<UCC_HOME>/conf/logging.properties` file, which configures the Log4J logging infrastructure used in UNICORE.

4.2.1.2.6 Installing UCC extensions

UCC can be extended with additional commands. It is enough to copy the libraries (`.jar` files) of the extension into a directory that is scanned by UCC: in general these are the UCC `lib` and the `${HOME}/.ucc/lib` directory.

4.2.1.2.7 Testing the installation

To test your UCC installation and to get information about the services available in the UNICORE system you're connecting to, do

```
$ ucc system-info -l -v
```

4.2.1.3 Getting started with UCC

Assuming you have successfully installed UCC, this section shows how to get going quickly.

4.2.1.3.1 Getting help

Calling UCC with the `-h` option will show the available options. To get a list of available commands, type:

```
$ ucc -h
```

To get help on a specific command, type:

```
$  ucc <command> -h
```

See also *Common options to UCC* for a list of common options.

4.2.1.3.2 Connecting

First, contact UNICORE and make sure you have access to some target systems:

```
$ ucc connect
```

4.2.1.3.3 List available sites

Then, list the sites available to you using:

```
$ ucc list-sites
```

4.2.1.3.4 Running your first job

The UCC distribution contains samples that you can run. Let's run the date.u sample. The `-v` switch prints more info so you can see what's going on.

```
$ ucc run -v <UCC_HOME>/samples/date.u
```

This will run *date* on a randomly chosen site, and retrieve the output. To run on a particular site, use the `-s` option to specify a particular target system.

Note: Look for UCC samples in the `/usr/share/doc/unicore/ucc/samples` directory.

4.2.1.3.5 Listing your jobs

The command

```
$ ucc list-jobs -l
```

will print a list of job URLs with their respective status (`RUNNING`, `SUCCESSFUL`, etc).

4.2.1.4 Common options to UCC

The following table lists the options understood by most UCC commands. Most commands have additional options. You can always get a summary of all available options for a command by calling UCC with the `-h` or `--help` option, for example,

```
$ ucc run --help
```

Since it is not possible to give all the required options on the commandline, it is mandatory to create a preferences file containing e.g. your settings for keystore, registry, etc.

Table 4.1: Common options for the UCC

Option (short and long form)	Description
-c,–configuration <Properties_file>	Properties file containing your preferences. By default, a file `$HOME/.ucc/preferences` is checked.
-k,–authentication-method <auth>	Authentication method to use (default: USERNAME)
-o,–output <Output_dir>	Directory for any output produced (default is the current directory)
-r,–registry <List_of_Registry_URLs>	The comma-separated list of URLs of UNICORE registries
-v,–verbose	Verbose mode
-h,–help	Print help message
-y,–with-timing	Timing mode

4.2.1.4.1 User preferences

If you have multiple user IDs or are a member of multiple Unix Groups on the target system, you may wish to control the user attributes that are used when invoking UCC.

Here is a list of options related to user attributes:

Table 4.2: User attribute options

Option (short and long form)	Description
-Z, –preference	Select from your remote attributes (e.g. xlogin)

The `preference` option accepts multiple arguments of the form <*name*>:<*value*> where *name*:

Table 4.3: User attribute options

Name	Description
uid	Remote login
pgid	Primary group ID
supgids	Secondary group IDs (comma-separated)
role	UNICORE role (user, admin, . . .)
vo	virtual organisation

4.2.1.4.2 Configuration file

By default, UCC checks for the existence of a file `<userhome>/.ucc/preferences` and reads settings from there. As shown above, you can use a different file by specifying it on the commandline using the `-c` option.

The configuration file can contain default settings for many commandline options, which are given in the form <*option name*>=<*value*> where <*option name*> is the long form of the option. The property values may contain variables in the form `${VAR_X}`, which are automatically replaced with the environmental variable values with the same name. Additionally a special variable `${UCC_CONFIG}` is recognized and is replaced with the absolute path of your configuration file.

The most important part of configuration is how UCC should authenticate you to the UNICORE server(s) and what server(s) should be trusted.

An overview of the available authentication options can be retrieved using:

```
$ ucc help-auth
```

A minimal example for using the *quickstart* installation would be:

```
registry=https://localhost:8080/DEMO-SITE/services/Registry?res=default_registry

authentication-method=username
username=demouser
password=test123

truststore.type=directory
truststore.directoryLocations.1=<path to CA file(s)>
```

Important: To protect your passwords, you should make the file non-readable by others, for example on Unix using a command such as `chmod 600 preferences`.

Note: If required passwords are not given in the properties file, they will be queried interactively.

4.2.1.4.3 Username and password authentication

To authenticate with username and password, set the following:

```
authentication-method=username
username=<your remote username>
password=<your remote password>
```

4.2.1.4.4 Support for token based authentication

UCC has three different options for using token-based authentication:

- via oidc-agent
- directly contact an OIDC server as an OIDC client (requires client ID and secret)
- specify the token directly as a config property

OIDC-Agent

UCC supports the oidc-agent tool that allows to interact with common OIDC servers to retrieve new access tokens.

To configure oidc-agent, UCC supports the following properties:

Table 4.4: Options for oidc-agent

Property name	Type	Default value / mandatory	Description
oidc-agent.account	string	*mandatory*	Account short name.
oidc-agent.lifetime	integer >= 1		Minimum lifetime of the issued access token.
oidc-agent.scope	string		OpenID scope(s) to request.

Your *config file* would require at least:

```
authentication-method=oidc-agent
oidc-agent.account=<oidc-agent account to be used>
```

OIDC Server

This is a low-level approach that requires the details on how to act as an OIDC client, you'll need at least an OIDC token endpoint, client ID and secret.

Table 4.5: Options for oidc-server

Property name	Type	Default value / mandatory	Description
oidc.authentication	[BASIC, POST]	BASIC	How to authenticate (i.e. send client id/secret) to the OIDC server (BASIC or POST).
oidc.clientID	string		Client ID for authenticating to the OIDC server.
oidc.clientSecret	string		Client secret for authenticating to the OIDC server.
oidc.endpoint	string	*mandatory*	The OIDC server endpoint for requesting a token
oidc.grantType	string	client_credentials	Grant type to request.
oidc.otp	string		Additional one-time password for two-factor authentication. Set this to 'QUERY' to query it interactively.
oidc.password	string		Password used to log in. It is suggested not to use this option for security reasons. If not given in configuration, it will be asked interactively.
oidc.refreshInterval	integer number	300	Interval (seconds) before refreshing the token.
oidc.refreshTokenFile	string		(internal) Filename for storing the refresh token between UCC invocations.
oidc.request_key_for_otp	string	otp	(internal) How to send the OTP value to the server.
oidc.username	string		Username used to log in. If not given in configuration, it will be asked interactively.

```
authentication-method=oidc-server
oidc.endpoint=<oidc server token endpoint>
oidc.username=...
oidc.password=...
```

UCC also supports sending an OTP (one-time password) to Keycloak. To enable, add

```
oidc.otp=QUERY
```

to your config. The OTP token is queried from the command-line (the OTP value can also be placed verbatim in the preferences as `oidc.otp=your_otp_value`).

UCC stores the refresh token (if any) and tries to use it, before using the username/password again, also accross UCC invocations. (The token is stored in a file "$HOME/.ucc/refresh-tokens", this default can be changed via a config variable)

Bearer token in config

Last not least, if you have a Bearer token via some other means, you can directly put the token into your *config file*:

```
authentication-method=bearer-token
token=...
```

4.2.1.4.5 Certificate-based authentication

For UNICORE installations that support (or even require) client certficates for authentication, set:

```
authentication-method=X509

credential.path=<your keystore>
credential.password=XXXXXXX
```

4.2.1.4.6 Truststore options

In most cases you only need a truststore directory containing trusted certificates:

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

A full list of options related to truststore management is available in the following table:

Table 4.6: Truststore properties

Property name	Type	Default value / mandatory	Description
trust-store.allowProxy	[ALLOW, DENY]	ALLOW	Controls whether proxy certificates are supported.
truststore.type	[keystore, openssl, directory]	*mandatory*	The truststore type.
trust-store.updateInterval	integer number	600	How often the truststore should be reloaded, in seconds. Set to negative value to disable refreshing at runtime.(runtime updateable)
trust-store.directoryConnectionTimeout	integer number	15	Connection timeout for fetching the remote CA certificates in seconds.
trust-store.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.
trust-store.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.
trust-store.directoryLocations.*	list of properties with a common prefix		List of CA certificates locations. Can contain URLs, local files and wildcard expressions.(runtime updateable)
trust-store.keystoreFormat	string		The keystore type (jks, pkcs12) in case of truststore of keystore type.
trust-store.keystorePassword	string		The password of the keystore type truststore.
trust-store.keystorePath	string		The keystore path in case of truststore of keystore type.
trust-store.opensslNewStoreFormat	[true, false]	false	In case of openssl truststore, specifies whether the trust store is in openssl 1.0.0+ format (true) or older openssl 0.x format (false)
trust-store.opensslNsMode	[GLOBUS_EUGRIDPMA, EU-GRIDPMA_GLOBUS, GLOBUS, EU-GRIDPMA, GLOBUS_EUGRIDPMA_REQUIRE, EU-GRIDPMA_GLOBUS_REQUIRE, GLOBUS_REQUIRE, EU-GRIDPMA_REQUIRE, EU-GRIDPMA_AND_GLOBUS, EU-GRIDPMA_AND_GLOBUS_REQUIRE, IGNORE]	EU-GRIDPMA_GLOBUS	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).
trust-store.opensslPath	filesystem path	/etc/grid-security/certificates	Directory to be used for opeenssl truststore.
trust-store.crlConnectionTimeout	integer number	15	Connection timeout for fetching the remote CRLs in seconds (not used for Openssl truststores).
trust-store.crlDiskCachePath	filesystem path		Directory where CRLs should be cached, after downloading them from remote source. Can be left undefined if no disk cache should be used. Note that directory should be secured, i.e. normal

4.2.1.4.7 Truststore examples

Here are some examples for commonly used trust store configurations.

Most commonly used is a directory (with a minimal set of options):

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

OpenSSL trust store:

```
truststore.type=openssl
truststore.opensslPath=/etc/grid-security/
truststore.opensslNsMode=EUGRIDPMA_GLOBUS_REQUIRE
truststore.updateInterval=1200
truststore.crlMode=IF_VALID
```

Java keystore used as a trust store:

```
truststore.type=keystore
truststore.keystorePath=/some/dir/truststore.jks
truststore.keystoreFormat=JKS
truststore.keystorePassword=xxxxxx
```

4.2.1.4.8 Client options

The configuration file may also contain low-level options, for example, if you need to specify connection timeouts, http proxies, etc.

Table 4.7: Client options

Property name	Type	Default value / mandatory	Description
client.digitalSigningEnabled	[true, false]	true	Controls whether signing of key web service requests should be performed.
client.httpAuthnEnabled	[true, false]	false	Whether HTTP basic authentication should be used.
client.httpPassword	string	*empty string*	Password for use with HTTP basic authentication (if enabled).
client.httpUser	string	*empty string*	Username for use with HTTP basic authentication (if enabled).
client.maxWsCallRetries	integer number	3	Controls how many times the client should try to call a failing web service. Note that only the transient failure reasons cause the retry. Note that value of 0 enables unlimited number of retries, while value of 1 means that only one call is tried.
client.messageLogging	[true, false]	false	Controls whether messages should be logged (at INFO level).
client.securitySessions	[true, false]	true	Controls whether security sessions should be enabled.
client.serverHostnameChecking	[NONE, WARN, FAIL]	WARN	Controls whether server's hostname should be checked for matching its certificate subject. This verification prevents man-in-the-middle attacks. If enabled WARN will only print warning in log, FAIL will close the connection.
client.sslAuthnEnabled	[true, false]	true	Controls whether SSL authentication of the client should be performed.
client.sslEnabled	[true, false]	true	Controls whether the SSL/TLS connection mode is enabled.
client.wsCallRetryDelay	integer number	10000	Amount of milliseconds to wait before retry of a failed web service call.
client.http.allow-chunking	[true, false]	true	If set to false, then the client will not use HTTP 1.1 data chunking.
client.http.connection-close	[true, false]	false	If set to true then the client will send connection close header, so the server will close the socket.
client.http.connection.timeout	integer number	20000	Timeout for the connection establishing (ms)
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.
client.http.maxRedirects	integer number	3	Maximum number of allowed HTTP redirects.
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.
client.http.socket.timeout	integer number	0	Socket timeout (ms)
client.http.nonProxyHosts	string		Space (single) separated list of hosts, for which the HTTP proxy should not be used.
client.http.proxy.password	string		Relevant only when using HTTP proxy: defines password for authentication to the proxy.
client.http.proxy.user	string		Relevant only when using HTTP proxy: defines username for authentication to the proxy.
client.http.proxyHost	string		If set then the HTTP proxy will be used, with this hostname.
client.http.proxyPort	integer number		HTTP proxy port. If not defined then system property is consulted, and as a final fallback 80 is used.
client.http.proxyType	string	HTTP	HTTP proxy type: HTTP or SOCKS.

4.2.1.4.9 Other options

The following table lists other options, that are more rarely used:

Table 4.8: Other options for the UCC

Property name	Description
blacklist	Comma separated *patterns* for sites / URLs to ignore
contact-registry	Do not attempt to contact the registry, even if one is configured

4.2.1.5 Running jobs

4.2.1.5.1 Introduction

The UCC can run jobs specified in the JSON job description format that is used by the UNICORE *REST API*, plus a few extensions related to handling of local files, submission options, etc. See *Job description format* for all the details.

In the following it is assumed that you have UCC installed *Installation and configuration* and tried some examples *Getting started with UCC*.

For example, assume the file *myjob.u* looks as follows:

```
{
  "ApplicationName": "Date",
  "ApplicationVersion": "1.0"
}
```

To run this through UCC, issue the following command:

```
$ ucc run myjob.u
```

This will submit the job, wait for completion, download the stdout and stderr files, and place them in your default output directory. The `run` command has a number of options, to see all the possibilities use the built-in help:

```
$ ucc run -h
```

Controlling the output location and file names

Output files will be placed in the directory given by the `-o` option, if not given, the current directory is used. Also, file names will be put into a subdirectory named as the job id, to prevent accidental overwriting of existing files. This behaviour can be changed using the `-b` option. When `-b` is given on the command line, no subdirectory will be created.

Specifying the site

In the example above, a random site will be chosen to execute the job. To control it, you can use the `-s` option. This will accept the name of a target system. The target systems available to you can be listed by

```
$ ucc list-sites
```

Accessing a job's working directory

Using the UCC's data management functions, the job working directory can be accessed at any time after job submission. Please see section *Data management functions* for details.

4.2.1.5.2 Options overview

The following options are available when running jobs (see also the general options overview in *Common options to UCC*):

Table 4.9: Job submission options for UCC

Option (Short and long form)	Description
-a,–asynchronous	Run asynchronously
-b,–brief	Do not create a sub-directory for output files
-B,–broker	Select the type of resource broker to use (see `run -h` for a list)
-s,–sitename <SITE>	Site where the job shall be run
-S,–schedule <Time>	Schedule the submission of the job at the given time
-o,–output <Output_dir>	Directory for any output produced (default is the current directory)

4.2.1.5.3 Resource selection

In general the user selects the execution site.

If no site is specified upon submission, UCC will select a matching site, where the requirements (resources, applications) are met.

In case there are other types of brokers available, they can be selected using the `-B` or `--broker` option.

- LOCAL (default): brokering is done by UCC itself

To see if other brokers exist, execute `ucc run -h`, the available options will be listed in the help for the `-B` option.

4.2.1.5.4 Processing jobs asynchronously

In case of long-running jobs, you will want to run the job asynchronously, i.e. just submit the job, stage in any files and start it, in order to get the results later.

Asynchronous submission

Use the `-a` flag when submitting a job:

```
$ ucc run -a <job file>
```

This will submit the job, stage-in any local files, start the job and exit. A job file will be written that contains information about the job endpoint and any exports that should be performed once the job has finished. You can use this file later with the `get-output` and `job-status` commands.

Get the status of particular jobs

The command

```
$ ucc job-status <job_file_or_url> <job_file_or_url_2> ...
```

will retrieve the status of the given jobs. If not given on the command line, a job URL will be read from the console.

The arguments can be either a job URL, or the name of a job file (as written by the `run -a` command).

Download results

To get stdout and stderr as well as other files marked for export, do:

```
$ ucc get-output -o <outdir> <job_file_or_url> <job_file_or_url_2> ...
```

Here, the option `-o` specifies the directory where to put the output, by default the current directory is used. As before, a job address can also be read from the console.

Referencing a job by its URL (endpoint address)

In case you want to check on a job not submitted through UCC, you can refer to a job by its URL. The *list-jobs* command will produce a list of all job URLs that you can access.

Note that in this case UCC will only retrieve stdout and stderr files. To download other result files, you'll have to use the data movement functions described in *Data management functions*.

Scheduling job submission to the batch system

Sometimes a user wishes to control the time when a job is submitted to the batch queue, for example, because he/she knows that a certain queue will be empty at that time.

To schedule a job, you can either use the `-S` option to the ucc `run` command:

```
$ ucc run -S "12:24" ...
```

Alternatively, you can specify the start time in your job file using the `Not before` keyword:

```
{
  "Not before": "12:30",
}
```

In both cases, the specified start time can be given in the brief “HH:mm” (hours and minutes) format shown above, or in the full ISO 8601 format including year, date, time and time zone:

```
{
 "Not before": "2011-12-24T12:30:00+0200",
}
```

4.2.1.5.5 Executing a command

If you just want to execute a simple command remotely (i.e. without data staging, resource specifications etc), you can use the `exec` command.

This will run the given command remotely (similarly to `ssh`), and print the output to the console. You can specify the site with the `-s` option. If you do not specify the site, a random site will be chosen.

UNICORE will run the command on the login node, it will not be submitted to the batch system.

For example, try

```
$ ucc exec /bin/date
```

To safely pass arguments to the executable, use “–” to end the UCC part of the command line, for example

```
$ ucc exec -- date --rfc-email
```

4.2.1.6 Job description format

UCC uses the JSON *Job description format* that is used by the UNICORE *REST API*, adding support for handling local files.

Several complete job samples can be found in the samples directory of the distribution. On Linux, check also the `/usr/share/unicore/ucc/samples` directory.

To view an example job showing most of the available options, run:

```
$ ucc run -H
```

(most of the options shown are not mandatory, of course).

4.2.1.6.1 UCC extensions to the UNICORE job description format

It is often the case that your job requires files from your local workstation, or you want UCC to download result files once the job has finished.

UCC achieves this by allowing paths to local files in the `To` and/or `From` directives of the data staging part(s) in your job.

Local files can be given as an absolute or relative path; in the latter case the configured output directory will be used as base directory.

Importing local files into the job workspace

To import files from your local computer, you can use the usual `Imports` keyword, with a path as the `From` argument.

You can of course mix local and remote files. This example shows some of the possibilities:

```
{

      "Imports": [

      #
      # import a local file from the client machine
      # into the job workspace
      #

        { "From": "/work/data/fileName", "To": "fileName" },

      #
      # import a set of local files from the client machine
      # into the job workspace
      #

        { "From": "/work/data/pdf/*.pdf", "To": "/" },

      #
      # import a remote file from a UNICORE storage using the UFTP protocol
      #

        { "From": "UFTP:https://gw:8080/DEMO-SITE/rest/core/storages/Home/files/testfile
↪",

          "To": "testfile" },

      # create a symlink from a file on the compute machine to the job workspace

        { "From": "link:/work/data/testfile", "To": "linked-file" },

      # copy a file on the compute machine to the job workspace

        { "From": "file:/work/data/testfile", "To": "copied-file" },

      ],

}
```

If for some reason an import fails, but you want the job to run anyway, there is a flag `FailOnError` that can be set to `false`:

```
"Imports": [

     { "From": "/work/data/fileName",
       "To": "fileName",
       "FailOnError": "false"
     },

```

(continues on next page)

(continued from previous page)

```
],
```

Note: UCC supports simple wild cards (* and ?) for importing and exporting files.

Exporting result files from the job workspace

To export files from the job's working directory to your local machine, use the normal `Exports` keyword, with a file path as the `To` argument. Here is an example `Exports` section that specifies two exports:

```
{
  "Exports": [

    # this exports all png files to a local directory

      { "From": "*.png", "To": "/home/me/images/" },

    # this exports a single file to a to local directory
    # failure of this data transfer will be ignored

      { "From": "error.log",
        "To": "/home/me/logs/error.log",
        "FailOnError": "false" },

    # this exports to a UNICORE storage

    { "From": "stdout",
      "To": "https://gw:8080/DEMO-SITE/rest/core/storages/Home/files/results/myjob/stdout
↪"
    },
  ]
}
```

As a special case, UCC also supports downloading files from other UNICORE storages (after the job has finished), using the `Exports` keyword:

```
{
   "Exports": [

    # this exports a file from a UNICORE storage

      { "From": "https://gw:8080/DEMO-SITE/rest/core/storages/Work/files/somefile",
        "To": "/home/me/somefile"
      },
  ]
}
```

4.2.1.7 Data management functions

UCC offers access to all the data management functions in UNICORE. You can upload or download data from a remote server, initiate a server-to-server transfer, create directories and so on.

4.2.1.7.1 Specifying remote locations

Remote locations are specified via URIs that includes protocol, storage server (*host/port*), site name, and filename. For example,

```
BFT:https://mygateway:8080/SITE/rest/core/storages/HOME/files/my_file
```

specifies a file named */my_file* on the storage instance *https://mygateway:8080/SITE/rest/core/storages/HOME*, using the BFT protocol.

Paths are always **relative** to the storage root, not the root of the actual file system.

The protocol is optional, and will default to `BFT` if not given.

4.2.1.7.2 Data movement

cp

The `cp` command is a generic command for copying source file(s) to a target destination, where source and target can be remote locations or files on the local machine. Wild card characters * and ? are supported.

Examples for client-server transfers:

```
$ ucc cp data/*.pdf https://server/rest/core/storages/SHARE/files/pdfs
$ ucc cp https://server/rest/core/storages/SHARE/files/pdfs .
```

The `-R` option allows to choose whether subdirectories are to be copied too.

The `-X` option allows to resume a previous transfer. Missing data will be appended to an existing target file (if the chosen protocol supports it).

Examples for server-server transfer:

```
$ ucc cp https://server/rest/core/storages/SHARE/files/*.pdf  \
     https://otherserver/rest/core/storages/WORK/data/
```

For server-to-server transfers, the `cp` command supports several additional options.

The `-S` option allows to schedule a transfer for a certain time. For example,

```
$ ucc cp -S "23:00" ...
```

The format is simply *HH:mm* (hours and minutes). Alternatively, you can give the time in the full ISO 8601 format including year, date, time and time zone:

```
$ ucc cp -S "2011-12-24T12:30:00+0200" ...
```

Another useful option is `-a` which will execute the server-server transfer asynchronously, i.e. the client will not wait for the transfer to finish.

copy-file-status

This will print the status of the given data transfer. As argument, it expects a file name containing the transfer reference, or directly the reference.

Example (for Unix) which captures the reference into a shell variable:

```
$ export ID=$(ucc cp -a ...
$ ucc copy-file-status $ID
```

Specifying the file transfer protocol

To use a different protocol from the default BFT, you can use the `-P` option to specify your preferred protocol. UCC will try to match them with the capabilities of the storage and use the first match. Your preferred protocol can also be listed in your preferences file using the `protocols` key:

```
protocols=UFTP
```

Note: If necessary, you can specify additional filetransfer options in your preferences file as well. For example, to use the UFTP protocol you may need to specify the client host address and the number of parallel streams explicitely:

```
uftp.client.host=your_client_ip_address
uftp.streams=2
# encrypt data (at the cost of performance)
uftp.encryption=true
# compress data
uftp.compression=true
```

Use the special value `all` to enable all available client IP addresses for UFTP.

```
uftp.client.host=all
```

You can also override the UFTP server host, which can be useful in case the UFTP server is accessible via multiple network interfaces:

```
uftp.server.host=myhost.com
```

UCC will try to use reasonable defaults for any missing parameters.

4.2.1.7.3 General commands

mkdir

This will create a directory (including required parent directories) remotely.

Example:

```
$ ucc mkdir https://mygateway:8080/SITE/rest/core/storages/HOME/files/pdfs
```

rm

This will remove a file or directory remotely. By default, UCC will ask for a confirmation. Use the `--quiet` or `-q` option to disable this confirmation (e.g. when using this command in scripts).

Example:

```
$ ucc rm https://mygateway:8080/SITE/rest/core/storages/HOME/files/pdfs
```

rename

This will rename/move a remote file/directory on the same storage.

Example:

```
$ ucc rename https://mygateway:8080/SITE/rest/core/storages/HOME/files/data/foo1.pdf /
↪files/data/foo2.pdf
```

will rename the file *foo1.pdf* to *foo2.pdf*.

stat

This command shows full information on a certain file or directory. Add the `-m` flag to also print user-defined metadata.

Example:

```
$ ucc stat -m https://mygateway:8080/SITE/rest/core/storages/HOME/files/foo.txt
```

4.2.1.7.4 Finding data

ls

This will list a remote directory. Useful options are: `-l` (detailed output), `-H` (human-friendly) and `-R` (recurse).

Example:

```
$ ucc ls -l -H https://mygateway:8080/SITE/rest/core/storages/HOME/
```

If the storage supports metadata, you can get the metadata of a single file using `ls -l -m`:

```
$ ucc ls -l -m https://mygateway:8080/SITE/rest/core/storages/HOME/.bashrc
```

4.2.1.7.5 Using the StorageFactory service

UNICORE sites may allow users to dynamically create storage resources, which even can be linked to special back-end systems like Apache HDFS, iRODS, or cloud storage like Amazon S3.

You can find out if there are sites supporting this *StorageFactory* service either by running the `system-info -l` command, or better using

```
$ ucc create-storage -i
```

This will list the available *StorageFactory* services and also show which types of storage are supported and how much space is left on each of them.

UCC supports creating storages via the `create-storage` command. The simple

```
$ ucc create-storage
```

will create a new storage resource using the default storage type at some site.

Usually you want to control at least where the storage is created. Additionally, the type of storage and some parameters can be passed to UCC.

As an example, creating a storage of type `S3` would look like this:

```
$ ucc create-storage -t S3 accessKey=... secretKey=...
```

You can also read parameters from a file. Say you have your S3 keys in a file *s3.properties*, then you can use the following syntax:

```
$ ucc create-storage -t S3 @s3.properties
```

You can also mix this with the normal *key=value* syntax, or mix it like this:

```
$ ucc create-storage -t S3 accessKey=@s3.accessKey secretKey=@s3.secretKey
```

The last version *key=@file* causes just the value to be read from the named file.

4.2.1.8 Metadata management functions

UCC offers a simple interface to access the metadata management service in UNICORE.

4.2.1.8.1 Basics

The metadata functions are all accessed via a single UCC command `metadata`. The actual operation to be performed is given with the `-C` (i.e. `command`) option.

The storage to be operated upon is given using the `-s` option.

In addition to the URL, the name of the target file on the storage is required.

Metadata is represented in JSON format. The metadata operations usually read metadata from a file (or write results to a file), which is specified using the `-f` option.

In the following examples, <`STORAGE`> denotes the URL of a storage capable of handling metadata.

4.2.1.8.2 Available commands

Creating metadata

To create metadata, a file in JSON format is required containing *key-value* pairs. For example, edit the file *meta.json* to contain:

```
{
  "foo": "bar"
}
```

Say we have a file *test* on our storage, then you can create metadata as follows:

```
$ ucc metadata -C create -f meta.json -s <STORAGE> /test
```

If you now look at the file with `ls -l -m`,

```
$ ucc ls -l -m  <STORAGE>/test
```

you should get something like this:

```
-rw-           3344 2011-06-27 22:32 /test
{
  "foo": "bar",
  "resourceName": "/test"
}
```

Reading metadata

Apart from the `ls -l -m` used above, there is also an explicit `read` command, which can write the metadata to a file as well:

```
$ ucc metadata -C read -s <STORAGE> /test -f out.json
```

The `-f` option is optional.

Updating metadata

Using `update`, the given metadata is merged with any existing metadata. Say we have a file *x.json* containing:

```
{
 x: y
}
```

We can append this to the existing metadata:

```
$ ucc metadata -C update -s <STORAGE> /test -f x.json
```

Check that the metadata has indeed been appended.

Deleting metadata

Explicitely deleting is also possible:

```
$ ucc metadata -C delete -s <STORAGE> /test
```

Check that the metadata has indeed been deleted.

Searching

Searching requires a search string (according to the rules of Apache Lucene), and is triggered by the `search` command:

```
$ ucc metadata -C search -q "foo" -s <STORAGE> /
```

Triggering metadata extraction

To trigger the extraction of metadata on the server, use the `start-extract` command:

```
$ ucc metadata -C start-extract -s <STORAGE> /
```

In this case the `/` denotes the base path from which to start the extraction process.

4.2.1.9 Workflows

4.2.1.9.1 Introduction

UCC supports the UNICORE *Workflow service* and allows to submit workflows and manage them.

The workflows are executed server-side, and UCC is used only for submitting, managing data and getting results. UCC also provides helper features for dealing with the workflows' input/output data and parametrised workflow templates.

Note: Version 8.x of the Workflow service has changed a lot, and existing 7.x XML workflows will need to be converted / refactored.

4.2.1.9.2 Command overview

The following commands are provided:

- `workflow-submit`: submit a workflow file
- `workflow-control`: abort or resume a running workflow
- `list-workflows`: list information about workflows

More details and examples follow below.

4.2.1.9.3 Basic use

To check the availability of the Workflow service in the configured registry, issue the following command:

```
$ ucc system-info -l
```

This should show at least an accessible Workflow service.

The distribution contains some example workflow files in the `samples/workflows` directory that you can edit and submit.

```
$ ucc workflow-submit yourworkflow.json
```

which will submit the workflow and print the address of the workflow to standard output.

To get the workflow status:

```
$ ucc list-workflows <workflow_address>
```

To list all your workflows, you can use the `list-workflows` command without an explicit workflow address:

```
$ ucc list-workflows -l
```

4.2.1.9.4 Workflow description format

The JSON format used by that the Workflow service can be found *here*.

4.2.1.9.5 Managing workflow data

Importing local data for use by a workflow

If you have local files that need to be imported before starting the workflow, you can use a normal UCC job file that contains only an `Imports` section:

```
{
 "Imports":
 [
   { "From": "local_file_1", "To": "wf:workflow_file_name_1", },
   { "From": "local_file_2", "To": "wf:workflow_file_name_2", },
 ...
 ],
}
```

UCC will upload the local files to a remote storage (which you can specify) and automatically register them with the workflow upon submission.

Your workflow JSON can then reference them as `wf:...` in the workflow activities.

You can also *manually* register files by adding in `inputs` section to your workflow JSON.

```
{
 "inputs": {
   "wf:infile1" : "remote_url_1",
```

(continues on next page)

(continued from previous page)

```
   "wf:infile2" : "remote_url_2",
 },
}
```

Workflow templates

If the workflows contains a `Template parameters` section, the corresponding replacement will be done by reading parameter values from the `.u` file. These so-called *workflow templates* can be a very simple and safe way to make adjustments in complex workflows before submission. As an example, consider the following workflow:

```
{
  "Template parameters": {
      "SLEEPTIME": {
         "type" : "INTEGER",
         "default": "10",
      }
  },

  "activities" : [
      {
          "id": "sleep1",
          "job": {
              "Executable": "sleep",
              "Arguments": ["${SLEEPTIME}"],
          },
      },
  ],
}
```

This introduces a parameter *SLEEPTIME* with default value *10*.

When the workflow is submitted, you can specify a JSON file with the `-u` option, which will be checked for a parameter named *SLEEPTIME*

```
{
  "SLEEPTIME": "1",
}
```

and if present, the value will be replaced in the workflow.

4.2.1.9.6 Resuming a held workflow

A workflow in status `HELD` can be resumed using the workflow-control `resume` command. If the workflow has variables/parameters, updated values can be sent with the `resume` command.

4.2.1.10 Batch processing

The `batch` command allows you to run many jobs without having to start UCC each time. You can control how many jobs should go to which site. This allows efficient job processing, while putting some load on the client machine. If you need to take the client offline, you should consider using the workflow system instead, which also allows efficient high-throughput processing.

Assume you have a bunch of jobs in *UCC's job description format* stored in a directory *jobs*. The output should go to a directory *out*. You can run them all through UCC using a single invocation as follows:

```
$ ucc batch -i jobs -o out
```

As job files, UCC will accept files ending in `.u`.

4.2.1.10.1 Options

You can run in *follow* mode, where UCC will watch the input directory, and will process new files as they arrive:

```
$ ucc batch -f -i jobs -o out
```

4.2.1.10.2 Performance tuning options

Getting the most performance out of UCC and the UNICORE installation can be a challenging task. Sending too many jobs to a site might decrease throughput, sometimes the client machine can be the limiting factor, etc.

You should experiment a bit to get the best performance for your specific setup. UCC has many options available for tuning. Here is an overview:

Table 4.10: Tuning options for the UCC batch mode

Option (short and long form)	Description
-K,–keep	Do not delete finished jobs on the server. By default, finished jobs are destroyed.
-m,–max <MaxRunningJobs>	Limit on jobs submitted by UCC at one time (default: 100)
-t,–threads <NumThreads>	Number of threads to be used for processing (default: 4)
-u,–update <UpdateInterval>	Minimum time in milliseconds between status requests on a single job (Default: 1000)
-R,–no-resource-check	Do not check if the necessary application is available on the target system (will increase performance a bit)
-X,–no-fetch-outcome	Do not fetch standard output and error
-S,–submit-only	Only submit the jobs, do not wait for them to finish
-M,–max-new-jobs	Limit the number of job submissions (default: 100)
-s,–sitename	Specify which site to use
-W,–site-weights	Specify a file containing site weights

4.2.1.10.3 Resource selection in batch mode

By default, the UCC batch mode will select a random site for running a job. You can modify the selection in different ways:

- using the `-s` option or a `Site:  <sitename>,` entry in the job file, you can specify the site directly
- use the `-W` option to specify a file containing site weights

Say you have two sites where one site is a big cluster and the other a small cluster. To send more jobs to the big cluster, you can use the site weights file:

```
#example site weights file for use with "ucc batch -W ..."

BIG-CLUSTER = 100
SMALL-CLUSTER = 10

#send no jobs to this site
BAD-CLUSTER = 0

# set default weight (for any sites not specified here)
UCC_DEFAULT_SITE_WEIGHT = 10
```

This would tell UCC to send 10 times more jobs to the *BIG-CLUSTER* site, and send no jobs to the *BAD-CLUSTER*. All other sites would get weight *10*, i.e. the same as *SMALL-CLUSTER*.

4.2.1.11 >_ The UCC shell

If you want to run a larger number of UCC commands, the overhead of starting the Java VM or checking the registry may become annoying. For this scenario, UCC offers a `shell` that allows the user to enter UCC commands interactively.

It is started by

```
$ ucc shell <options>
```

If you want to process a list of commands from a file instead of typing them, you can start the shell like this:

```
$ ucc shell -f commandsfile
```

or on Unix you can use the redirection features:

```
$ ucc shell < commandsfile
```

4.2.1.11.1 Changing property settings

To change a property setting in shell mode, you can use the `set` command. Without additional arguments, current properties are listed:

```
ucc> set registry=https://...  output=/tmp  ...
```

To set one or more properties, add space separated *key=value* strings:

```
ucc> set output=/work registry=https://....
```

You can also clear a property (set it to null) by using `unset`:

```
ucc> unset registry
```

Variables referenced via `$var_name` or `${var_name}` will be replaced in commands. You can use this to make commands shorter and more readable. It's also possible and useful to pre-set certain things in your preferences file.

For example,

```
ucc> set S1=https://myserver/my_site/rest/core/storages/HOME
ucc> ls -l $S1
```

A special variable `$_` exists that is set by various commands to the *last thing* that was created or accessed.

For example,

```
ucc> run -a date.u
ucc> job-status $_
```

4.2.1.11.2 Running an external command

You can run an external command via the `system` (or simply `!`) shell command. For example,

```
ucc> system vi job.u
```

or simply

```
ucc> ! cat job.u
```

4.2.1.11.3 Exiting the shell

To exit, type `exit` or press `Control-D`.

4.2.1.12 Sharing resources

Accessing UNICORE resources (jobs, storages, ...) is usually only possible when you *own* the resource or when there are special server-side policies in place that allow you access.

UNICORE supports ACLs on a *per-service* instance basis. This means, that you can give other users access to your jobs, storages, etc.

For example, you might want to allow others to check your jobs' status, or even allow them to abort jobs.

Note that to access actual files the permissions on file system level still need to match. Usually this is achieved by using Unix groups.

4.2.1.12.1 Editing ACLs

The ACLs are managed via the `share` command. Use the basic

```
$ ucc share <URL>
```

to share the current ACL for the given resource, where *URL* is the full URL of the resource, e.g.

```
$ ucc share https://localhost:8080/DEMO-SITE/rest/core/storages/HOME
```

To add an ACL entry use:

```
$ ucc share ACE1 ACE2 ... <URL>
```

where *ACE* is an access control entry expressed in a simple format:

```
[read|modify]:[DN|VO|GROUP|UID]:[value]
```

For example, to give *modify* permission to a user whose UNIX user id on the target system is *test*, you would use:

```
$ ucc share modify:UID:test <URL>
```

To delete entries use the `-d` option:

```
$ ucc share -d modify:UID:test <URL>
```

To delete **all** entries use the `-b` option:

```
$ ucc share -b <URL>
```

4.2.1.12.2 Permission levels

The permissions controlled by ACLs are as follows:

- `read`: access resource properties
- `modify`: perform actions e.g. job submission or creating a file export

Only the **owner** of a resource can edit the ACL or destroy the resource.

4.2.1.13 Port forwarding / tunneling

Starting with UNICORE 9.1.0, it is possible to open a tunnel (TCP socket connection) from the client to a service running on the HPC cluster. The service can run on a login node or even on a compute node.

Since this mechanism uses only the established UNICORE communication channels, it will work in any situation, unhindered by firewalls.

Traffic is forwarded from the client through the UNICORE HTTPS stack *down* to the cluster login node, where a (*TSI*) process is in charge of connecting to the backend service and forwarding data back through the UNICORE stack to the client. So there is chain of connections forwarding data through the following stack

- Client application
- Client-side listener (e.g. UCC)

- Gateway
- UNICORE/X
- Server-side listener (TSI process) on the login node
- Service

(in both directions).

That is quite a number of *hops*, so latency and throughput will be limited accordingly.

To establish the client side, UCC has a command `open-tunnel`, which behaves similarly to an SSH tunnel (`ssh -L ...`)

It is started by

```
$ ucc open-tunnel -L <listen-port> <endpoint>
```

The `listen-port` is the port where a local application can connect. You can use "0" to use any free port.

The `endpoint` is a UNICORE job endpoint URL, with a few extra parameters added:

`/forward-port?port=<service_port>&host=<service_host>&loginNode=<tsinode`

The `port` parameter is mandatory, and denotes the port where the backend service is listening.

The `host` and `loginNode` are optional:

- `host` is the host where the service is running, must be reachable from the TSI (login node). It defaults to `localhost` (as seen from the login node!).
- `loginNode` is useful in cases where there are multiple login nodes, and you wish to control on which login node the forwarding process is launched.

Upon connection, the tunneling process is initiated, and the forwarding of data is started. To stop listening and forwarding, press `Control-C`, or use some other method to stop the UCC process.

4.2.1.13.1 Example

While usually the backend service is also started via UNICORE, that is not strictly necessary. Any of your job endpoints will do.

In this example, however, we launch a Python web server via UNICORE, and then connect to that Python service via a tunnel.

Launch a *UCC shell* with `ucc shell ...` and run the following job to start the service, which will be listening on port 8877:

```
run -a

{ "Executable" : "python3 -m http.server 8877" }

(type CTRL-D to to launch the job)
```

Make sure to wait until this job is running, i.e.

```
job-status $_
```

shows it as **RUNNING**. The UCC shell special variable `$_` automatically contains the last URL, i.e. the new job's URL.

To open the tunnel:

```
open-tunnel -L 4321 $_/forward-port?port=8877
```

this will open a local listener on port 4321.

To test your tunnel, run something like the following (from ANOTHER terminal, **don't kill UCC**):

```
wget http://localhost:4321/stdout
```

You might also try and open "http://localhost:4321" in a browser.

4.2.1.13.2 Final notes

Attention: USE RESPONSIBLY! This tool is not intended for high volume data streaming or a very high number of concurrent connections, since it does incur some overhead on the UNICORE infrastructure.

4.2.1.14 UCC for site administrators

UCC can be used for administrative and user support tasks, like checking server status, or getting the full details of a user job.

4.2.1.14.1 Security considerations

Usually, each UNICORE user has only access to his or her own resources (such as jobs). For administrative use, you will need to aquire administrator privileges. There are two ways to achieve this:

- create dedicated user credentials (e.g. a certificate) and map them to the role *admin* (in the *XUUDB*, or whatever attribute source you are using). This method is recommended if you want to remotely administrate *UNICORE/X*.
- use the server keystore (of the UNICORE/X server you want to administrate) as UCC keystore. This will also give you administrator privileges. For this you will need to be logged on to the UNICORE/X server, and UNICORE/X must accept certificate authentication.

4.2.1.14.2 Admin commands

UCC has dedicated commands for accessing the *AdminService* of a UNICORE/X container. To get started, try:

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

UCC will try to access the admin service on each availabe UNICORE/X server. For each server, a list of statistical and performance data will be listed.

It will also list the available admin commands for each server, with a short description of their parameters. For example, here is a sample output:

```
https://localhost:8080/DEMO-SITE/services/AdminService?res=default_admin
  Services:
    TargetSystemFactoryService[1]
    ...
```

(continues on next page)

(continued from previous page)

```
  Monitors:
    use.externalConnectionStatus.REST_UnitySAMLAuthenticator: OK
    use.security.overview: ServerIdentity: CN=Demo UNICORE/X,O=UNICORE,C=EU;Expires: Thu␣
↪Sep 09 12:01:19 CEST 2032;IssuedBy: CN=Demo CA,O=UNICORE,C=EU
    ....
  Metrics:
    use.externalConnectionStatus.REST_UnitySAMLAuthenticator: OK
    use.rest.callFrequency: 0.016677196376660174
    ...
  Available commands:
    ShowJobDetails : parameters: jobID, [xnjsReference]
    ShowServerUsageOverview : parameters: [clientDN]
    ToggleResourceAvailability : 'resources' - comma separated list of IDs
    ToggleJobSubmission : parameters: [message]
    ToggleBESJobSubmission :
```

To invoke a command, the `admin-runcommand` is used. It can take optional parameters.

Disabling/enabling job submission

For example, it is possible to disable/enable job submission to the server, using the `ToggleJobSubmission` command, which can take an optional message:

```
$ ucc admin-runcommand ToggleJobSubmission message="Maintenance"
```

The service will reply:

```
$> SUCCESS, service reply: OK - job submission is disabled
```

If a user now tries to submit, she will receive an error message on submission. Running the command again will re-enable the service:

```
$ ucc admin-runcommand ToggleJobSubmission message="Maintenance"
$> SUCCESS, service reply: OK - job submission is now enabled
```

Getting job details

To get the full job details (for example in user support), try:

```
$ ucc admin-runcommand ShowJobDetails jobID=<unique_jobid>
```

For example,

```
ucc admin-runcommand ShowJobDetails jobID=461f78c7-82a6-4887-9c33-6f538a4b0cb2
SUCCESS, service reply: Job information for 461f78c7-82a6-4887-9c33-6f538a4b0cb2
{Info=Action ID       : 461f78c7-82a6-4887-9c33-6f538a4b0cb2
Action type     : JSON
Status          : DONE (trans.: none)
Result          : SUCCESSFUL [Success.]
Owner           : CN=Demo User, O=UNICORE, C=EU
```

(continues on next page)

(continued from previous page)

```
Job Definition: {"Job type":"interactive","DetailedStatusDisplay":"true","KeepFinishedJob
↪":"true","Output":"/tmp","IDLocation":"/tmp","Executable":"date","haveClientStageIn":
↪"false","Tags":["testing"]}
Processing context: de.fzj.unicore.xnjs.ems.ProcessingContext@17f4b0b6
Application Info: Application <unnamed>
Job log:
Thu Feb 25 16:25:07 CET 2021: Created with ID 461f78c7-82a6-4887-9c33-6f538a4b0cb2
Thu Feb 25 16:25:07 CET 2021: Created with type 'JSON'
Thu Feb 25 16:25:07 CET 2021: Client: Name: CN=Demo User,O=UNICORE,C=EU
Xlogin: uid: [schuller], gids: [schuller:audio, active=schuller, addingOSgroups: true]
Role: user: role from attribute source
Security tokens: User name: CN=Demo User,O=UNICORE,C=EU
Delegation to consignor status: true, core delegation status: false
Message signature status: UNCHECKED
Client's original IP: 127.0.0.1
Thu Feb 25 16:25:07 CET 2021: No staging in needed.
Thu Feb 25 16:25:07 CET 2021: Status set to READY.
Thu Feb 25 16:25:07 CET 2021: Status set to PENDING.
Thu Feb 25 16:25:08 CET 2021: Execution on login node
Thu Feb 25 16:25:08 CET 2021: Command is:
Thu Feb 25 16:25:08 CET 2021: #!/bin/bash -l
export PROJECTS_DIR=/opt/shared-data
#TSI_EXECUTESCRIPT

#RESOURCES
#TSI_DISCARD_OUTPUT true
#TSI_SCRIPT
#TSI_UMASK 77
umask 77
cd /opt/shared-data/UNICORE-Jobs//461f78c7-82a6-4887-9c33-6f538a4b0cb2/
 { date > /opt/shared-data/UNICORE-Jobs//461f78c7-82a6-4887-9c33-6f538a4b0cb2/stdout 2> /
↪opt/shared-data/UNICORE-Jobs//461f78c7-82a6-4887-9c33-6f538a4b0cb2/stderr; echo $? > /
↪opt/shared-data/UNICORE-Jobs//461f78c7-82a6-4887-9c33-6f538a4b0cb2//UNICORE_SCRIPT_
↪EXIT_CODE ; } & echo $! > /opt/shared-data/UNICORE-Jobs//461f78c7-82a6-4887-9c33-
↪6f538a4b0cb2//UNICORE_SCRIPT_PID
Thu Feb 25 16:25:08 CET 2021: TSI reply: submission OK.
Thu Feb 25 16:25:11 CET 2021: Submitted to classic TSI as [schuller schuller:DEFAULT_
↪GID] with PID=30112 on [localhost]
Thu Feb 25 16:25:11 CET 2021: Exit code 0
Thu Feb 25 16:25:11 CET 2021: Job completed on BSS.
Thu Feb 25 16:25:11 CET 2021: Status set to DONE.
Thu Feb 25 16:25:11 CET 2021: Result: Success.
Thu Feb 25 16:25:11 CET 2021: Total: 3 sec., Stage-in: 0 sec., Queued: 0 sec., Main: 0␣
↪sec., Stage-out: 0 sec.}
```

Thus you can get a full view of what the user submitted and what was executed.

4.2.1.14.3 Listing jobs, sites, . . .

You can also use all normal UCC commands to access the server. Note however that due to the authentication and authorisation system in UNICORE, this may not always work as expected: the *admin* user might not have the required Unix permissions to access files, list directories, etc.

The UCC commands that list server-side things (*list-jobs*, etc.) accept a filtering option, that can be used to limit the results of the operation. Filtering works on the XML resource properties of the resource in question.

Filtering is enabled by the `-f` or `--filter` option of the form:

```
-f NAME OPERATOR VALUE
```

where *NAME* is the name of an element from the JSON resource properties.

For example, to list all jobs:

```
$ ucc list-jobs -f status equals RUNNING
```

To list all jobs submitted on Nov 13, 2007:

```
$ ucc list-jobs -f submissionTime contains 2007-11-13
```

Table 4.11: Filtering options

Operator (long and short form)	Description
equals, eq	String equality (ignoring case)
notequals, neq	String inequality (ignoring case)
contains, c	Substring match
notcontains, nc	substring non-match
greaterthan, gt	Lexical comparison
lessthan, lt	Lexical comparison

4.2.1.14.4 Low-level operations

UCC supports low-level access to REST API endpoints using the `rest` command, specifically you can execute HTTP GET, PUT, POST and DELETE requests with JSON content.

For example, to delete (destroy) a resource:

```
$ ucc rest delete <Address>
```

To get a complete property listing (i.e. print the JSON resource property document):

```
$ ucc rest get <Address>
```

To change properties, use the `put` command with JSON content:

```
$ ucc rest put '{"Tags": ["tests", "hpc" ]}'
```

These commands can be abbreviated, e.g. `ucc rest d <Address>`

4.2.1.15 Scripting

UCC can execute Groovy scripts. Groovy is a dynamic scripting language similar to Python or Ruby, but very closely integrated with Java. The scripting facility can be used for automation tasks or implementation of custom commands, but it needs a bit of insight into how UCC and UNICORE work.

4.2.1.15.1 Script context

Your Groovy scripts can access some predefined variables that are summarized in the following table:

Table 4.12: Variables accessible for scripts

variable	description	Java type
registry	A preconfigured client for accessing the registry	eu.unicore.client.registry.IRegistryClient
configurationProvider	Security configuration provider (truststore, etc)	de.fzj.unicore.ucc.authn.UCCConfigurationProvider
auth	REST authentication mechanism	eu.unicore.services.rest.client.IAuthCallback
registryURL	the URL of the registry	java.lang.String
messageWriter	for writing messages to the user	de.fzj.unicore.ucc.MessageWriter
commandLine	the command line	org.apache.commons.cli.CommandLine
properties	defaults from the user's properties file	java.util.Properties

4.2.1.15.2 Examples

Some example Groovy scripts can be found in the samples directory of the UCC distribution.

4.2.1.16 Frequently asked questions

4.2.1.16.1 Configuration

Do I really have to store my password in the preferences file? Isn't this insecure?

Putting the password in a file or giving it as a commandline parameter can be considered insecure. The file could be read by others, and the commandline parameters may be visible in for example in the output of the `ps` command. Thus, UCC will simply ask for the password in case you did not specify it.

How can I enable more detailed logging?

UCC uses Log4j 2, by default the configuration is done in `<UCC_HOME>/conf/logging.properties`. You can edit this file and increase the logging levels, choose to log to a file or to the console, etc.

4.2.1.16.2 Usage

Can I use multiple registries with UCC?

Yes. Simply use a comma-separated list of URLs for the `-c` option. However, you may only use a single key/truststore, so all registries (and sites listed in them) must accept the same security credentials.

Can I upload and execute my own executable?

Yes. Check *Running jobs*.

Can I use UCC to list the contents of the registry?

Using the *rest command* (and optionally the UNIX `jq` utility for formatting the output), this is very easy. For example,

```
$ ucc rest get https://localhost:8080/DEMO-SITE/rest/core/registries/default_
↪registry | jq
```

will list the content of the registry.

4.2.2 Building the UCC

4.2.2.1 Prerequisites

You need Java and Apache Maven. Check the versions given in the pom.xml file.

4.2.2.2 Building Java code

Clone the git repo and build the jars from the *root* dir:

```
$ git clone https://github.com/UNICORE-EU/commandline-client.git
$ cd commandline-client
$ mvn clean install -DskipTests
```

4.2.2.3 Creating distribution packages

The following commands create the distribution packages in *tgz*, *deb* and *rpm* formats. The versions are taken from the pom.xml.

4.2.2.3.1 tgz

```
$ cd distribution
$ mvn package -DskipTests -Ppackman -Dpackage.type=bin.tar.gz
```

4.2.2.3.2 deb

```
$ cd distribution
$ mvn package -DskipTests -Ppackman -Dpackage.type=deb -Ddistribution=Debian
```

4.2.2.3.3 rpm redhat

```
$ cd distribution
$ mvn package -DskipTests -Ppackman -Dpackage.type=rpm -Ddistribution=RedHat
```

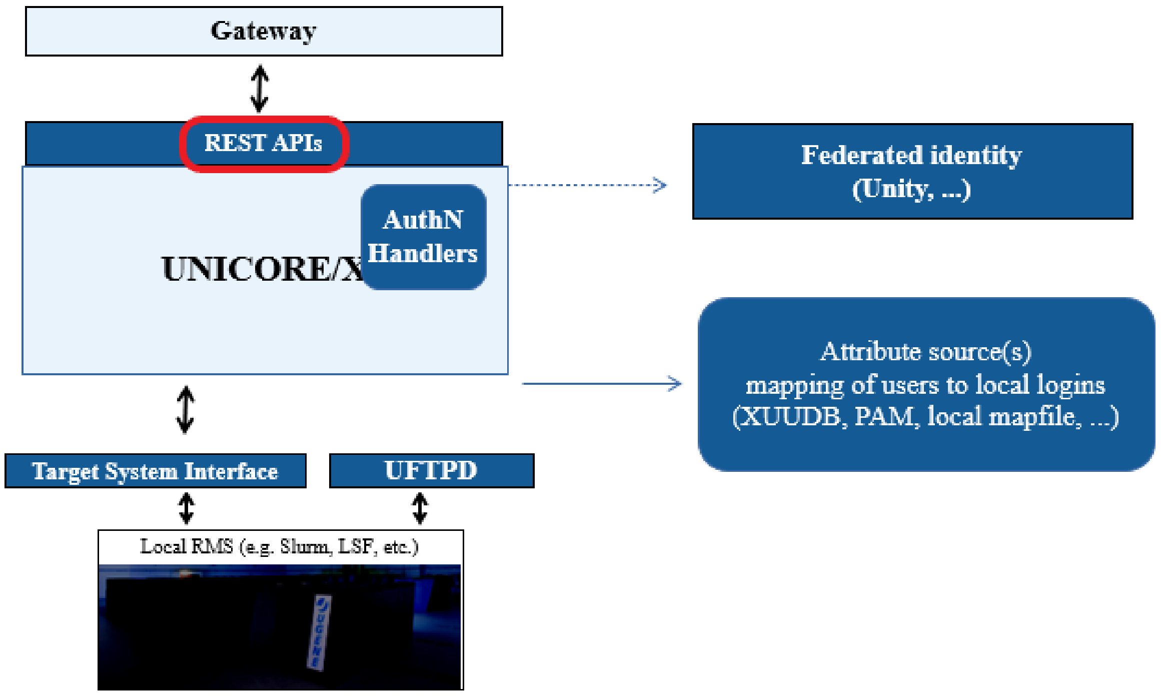
4.3 REST API

This document describes and documents the REST APIs for the *UNICORE/X* server (job submission and management, data access and data transfer) and the *Workflow* server (workflow submission and management).

The documentation generally refers to the latest released version.

A Python client library is under development on GitHub and can be installed from PyPI via `pip install pyunicore`.

Also, have a look at *Rest API Examples* for some examples using PyUNICORE.


Gateway
REST APIs
UNICORE/X
AuthN Handlers
Federated identity (Unity, ...)
Attribute source(s) mapping of users to local logins (XUUDB, PAM, local mapfile, ...)
Target System Interface
UFTPD
Local RMS (e.g. Slurm, LSF, etc.)

4.3.1 Basics

The REST API supports both the JSON (*application/json*) and HTML (*text/html*) content types.

4.3.1.1 Base URL

The base URL of the the REST API for a single UNICORE/X server is `https://gateway_url/SITENAME/rest/core`.

In the following, we will abbreviate this URL as `BASE` Authentication.

You need a user account on the *UNICORE/X* server, which is typically configured to use a password (or using an OAuth2 bearer token). The supported authentication methods depend on the UNICORE server.

For example, when username/password are enabled you can use `curl` to access the base URL above:

```
$ curl -k -u user:pass -X GET -H "Accept: application/json" BASE
```

In the following examples, we leave out the authentication details!

4.3.1.2 User preferences

In some special cases, you as a user might have more than one available unix account, or you might have more than just the *user* role on the server. It is possible to select from the available attributes. Available attributes are: `role`, `uid`, `group`.

For example, to execute a call using the Unix ID *some.user*, you can specify this in a HTTP header as follows:

```
$  curl -k -u user:pass -X GET -H "Accept: application/json"  \
       -H "X-UNICORE-User-Preferences: uid:some.user" BASE
```

Or, to select role *admin* (if you are worthy):

```
$  curl -k -u user:pass -X GET -H "Accept: application/json"  \
       -H "X-UNICORE-User-Preferences: role:admin" BASE
```

You can find out the available values for these attributes with a `GET` to the `BASE` URL!

To give more than one user preference, you can separate the values via commas, for example:

```
$  curl -k -u user:pass -X GET -H "Accept: application/json"  \
       -H "X-UNICORE-User-Preferences: uid:myuser,group:mygroup" BASE
```

4.3.1.3 Security session handling

If authentication was successful, the server reply will include a security session ID and a security session lifetime as HTTP headers:

```
X-UNICORE-SecuritySession: ....
X-UNICORE-SecuritySession-Lifetime: ...
```

The lifetime is given in milliseconds.

You can use the session ID in place of authentication info, i.e.

```
$ curl -k -H "X-UNICORE-SecuritySession: ..." BASE
```

If the session is no longer valid the server will reply with a `HTTP 432` error code, and you must re-send the authentication information.

Using security sessions is recommend especially when third-party IdPs for used for authentication, because it reduces the load on the servers and improves throughput and turnaround time for your API requests.

4.3.2 General API features

A few common operations and principles apply to all the REST resources.

- `GET` is used to retrieve information (resource properties). Depending on the `Accept:` header the format can be JSON or HTML (**JSON is recommended!**).
- `PUT` is used to modify resource properties (JSON format).
- `POST` creates new resources (e.g. job submission). The URL of new resource is returned in the response `Location` header. Some resources support `POST` also for triggering actions (e.g., *job abort*).
- `DELETE` removes resources.

4.3.2.1 Media types

The REST API uses the following media types:

- `application/json` is the commonly used media type for all sorts of tasks.
- `application/octet-stream` : used for upload/download of file data.

Make sure to add the proper headers e.g. `Accept: application/json` and/or `Content-Type: application/json` to your messages.

4.3.2.2 Error handling

As usual, a HTTP error code that is not in the *2xx* range signifies some sort of problem. For example,

- `401 Unauthorized` - your credentials are wrong or you do not have the right to access
- `404 Not found` - the resource does exist, or you have a typo in the URL
- `500 Internal server error` - a server-side error occurred

In many cases a JSON document containing an error message is returned.

4.3.2.3 Paging mechanism, using tags and controlling the output

Since lists of things can get long (e.g. files or jobs), there is a paging mechanism that is available on all the *list-like* things like job lists.

Let's look at an example. To list 5 jobs starting at an offset of 2, you'd do a GET for

```
BASE/sites/{id}/jobs?offset=2&num=5
```

The result would be something like:

```
{
  "_links": {
        "self": {
                "href": "BASE/sites/4q0b44VfBP2/jobs?offset=2&num=5"
        },
        "next": {
                "href": "BASE/sites/4q0b44VfBP2/jobs?offset=7&num=5"
        },
        "previous": {
                "href": "BASE/sites/4q0b44VfBP2/jobs?offset=0&num=5"
        }
  },
  "jobs": [
        "BASE/jobs/DZBiAG5O0kH",
        ... rest of job URLs omitted ...
  ]
}
```

Note that you get links to the next and previous *page* of results.

Another query parameter is `tags` which works on all lists of resources (jobs, storages, … but not files) and which allows you to limit results to those elements that have the specified tag(s).

```
BASE/sites/{id}/jobs/?tags="hpc,test"
```

Last not least you can control which fields you want in the output of a GET request. This is done via another query parameter `fields` which works on all resources (jobs, storages, …) and which allows you to limit results to the named fields.

```
BASE/sites/{id}/jobs/DZBiAG5O0kH?fields=status
```

Parameter	Format	Description
offset	integer (0, 1, 2, …)	How many elements of the list to skip
num	integer (0, 1, 2, …)	How many elements you want
tags	comma-separated strings	Only list elements with this tag
fields	comma-separated strings	Only return the named properties

4.3.2.4 Modifying resource properties

Resource properties can be modified by a PUT request in JSON format, which contains the properties and their new values, e.g.

```
{
  "tags": ["test", "hpc" ],
  "foo" : "bar"
}
```

to modify the umask of a storage. The server replies with a JSON indicating which properties were modified, or if errors occurred:

```
{
  "foo": "Property not found or cannot be modified!",
```

(continues on next page)

(continued from previous page)

```
  "tags": "OK"
}
```

4.3.3 REST resources for jobs and data management

The base URL of the the REST API for a single UNICORE/X server is `https://gateway_url/SITENAME/rest/core`.

URL	Method	Description
/	GET	Get info about the client, the server and links to other resources
/certificate	GET	Get the server's public key (needs "Accept: text/plain" or "Accept: *" header)

4.3.3.1 Synopsis

A `GET` request to the base URL retrieves some information about the current client, e.g. its security attributes. Also, links to the other resources (sites, jobs, etc) are given, as well as some information about the server.

4.3.3.2 API Summary

URL	Method	Description
/	GET	Get info about the client, the server and links to other resources
/certificate	GET	Get the server's public key (needs "Accept: text/plain" or "Accept: *" header)

4.3.3.3 Site factories

4.3.3.3.1 Synopsis

In UNICORE there are several *factory services* that are used to get general site information even as a non-authorized (but authenticated) user.

The `factories` resource is used to access the target system factory services available to the user and create new site resources.

4.3.3.3.2 API Summary

URL	Method	Description
/factories	GET	Get a list of all site factories
/factories	POST	Create a new site
/factories/id	GET	Get the representation of the given factory
/factories/id	POST	Create a new site
/factories/id/applications	GET	Get a list of all applications
/factories/id/ applications/appID	GET	Get the representation of the given applications

4.3.3.3.3 Site properties

A site factory exposes its capablilities (number of CPUs, etc) via its JSON or HTML representation. This includes the applications that are configured in the UNICORE server's IDB file.

Applications are listed in the form

```
NAME---vVERSION
```

e.g.

```
Date---v1.0
```

You can access more details about an application at the URL

```
BASE/factories/{id}/applications/{applicationID}
```

where the application ID is again composed of the application name and version as above.

4.3.3.4 Sites

4.3.3.4.1 Synopsis

The sites resource is used to access the target system services available to the user, create new ones, and to submit jobs.

4.3.3.4.2 Creating a site

Using POST, a new site resource can be created.

```
$ curl -X POST -H "Content-Type: application/json" --data "{}"  https://localhost:8080/
↪DEMO-SITE/rest/core/sites
```

Note that if necessary, a *default* site is created automatically when you submit a job.

4.3.3.4.3 Site properties

A site exposes its capabilities (number of CPUs, etc) via its JSON or HTML representation. This includes the applications that are configured in the UNICORE server's IDB file.

Applications are listed in the form

```
NAME---vVERSION
```

e.g.

```
Date---v1.0
```

You can access more details about an application at the URL

```
BASE/sites/{id}/applications/{applicationID}
```

where the *application ID* is again composed of the application name and version as above.

4.3.3.4.4 Listing jobs

You can list all the jobs submitted to a site using the `BASE/sites/{id}/jobs` endpoint, by doing a GET for

```
BASE/sites/{id}/jobs
```

Usually it is simpler to use the `BASE/jobs` endpoint.

4.3.3.4.5 API Summary

URL	Method	Description
/sites	GET	Get a list of all sites
/sites	POST	Create a new site
/sites/id	GET	Get a representation of the given site
/sites/id	DELETE	Destroy a site (but not the jobs)
/sites/id	POST	Submit a job to the site
/sites/id/jobs?offset=	GET	List num jobs, starting at offset, with the given tags
/sites/id/applications	GET	Get a list of all applications
/sites/id/applications/ appID	GET	Get a representation of the given application

4.3.3.5 Storages and files

4.3.3.5.1 Synopsis

The storages resource is used to access the storages available to the user, create new ones, access files, and initiate file transfers. Storages may support metadata management as well (depending on server configuration).

4.3.3.5.2 Listing storages

To get a list of storages accessible to you, simply do a `GET` for `BASE/storages`.

Each storage has a `files` subresource which is used for accessing files on a particular storage.

To list files on a storage with id {id}, do a `GET` for `BASE/storages/{id}/files/{filePath}`.

The same way as for jobs, there is a paging mechanism.

4.3.3.5.3 Data upload and download

There are two ways to transfer data to/from a UNICORE storage via the REST API. The simpler, more RESTful way is to use `HTTP GET` and `PUT` requests to download or upload data. This should be straightforward, the only thing to note is to NOT use the media type `application/json`, which is reserved for getting information about the file or changing properties.

For example, let's download a file named stdout using `curl`:

```
$ curl -X GET BASE/storages/{id}/files/stdout
```

The GET supports the `Range` header, if you want to download only part of the file. For example,

```
$ curl -X GET -H "Range: bytes=10-42" ....
```

You can also get the "tail" of the file, e.g. to get the last 100 bytes

```
$ curl -X GET -H "Range: bytes=-100" ....
```

Similarly, to upload a file localfile to a remote file newfile:

```
$ curl -X PUT --data-binary @localfile BASE/storages/{id}/files/some_dir/newfile
```

Note: Any required parent directories will be created automatically.

Attention: There is a special feature related to the `Content-Type` header here. If the *UNICORE/X* server is setup with metadata support, the value of the `Content-Type` header will be stored in the file's metadata. If you download the file later, the correct `Content-Type` will be used. This will work nicely and automatically for every media type EXCEPT `application/json`! See the *Rest API Examples* for a detailed example.

The second way is to create a UNICORE file transfer by POSTing to the `/imports` or `/exports` path, which requires custom clients, e.g. to support UFTP. For this reason, we do not consider this any further here.

4.3.3.5.4 Creating directories

If you want to create an empty directory, `POST` an empty JSON to `BASE/storages/{id}/files/{filePath}`.

4.3.3.5.5 Copying and renaming data to the same storage

Copy and rename on the same storage is done by POSTing a JSON to specific *action URLs*. The content of the JSON is

```
{"from": "source", "to": "target" }
```

and the URLs are

```
BASE/storages/{id}/actions/copy
BASE/storages/{id}/actions/rename
```

4.3.3.5.6 Copying data to another server

UNICORE supports server-to-server transfers using one of several protocols including the rather efficient UFTP protocol. This is initiated by POSTing to a storage resource's/transfers path `BASE/storages/{id}/transfers`.

The transfer is described using JSON, and can be either `push` or `pull`.

Parameter	Description
file	the file on the storage, relative to storage root
target	the target URL in case of "data push"
source	the source URL in case of "data pull"
extraParameters	any additional parameters (e.g. number of UFTP streams)

Note that to push data from a storage, you'd use the file and target parameters, while for data pull, you need the file and source.

For example, to pull a file via UFTP, you would POST a JSON that looks something like this:

```
{
   file: "localFile.txt",
   source: "UFTP:https://somehost/SITE/rest/core/storages/{id}/files/path_to_file"
   extraParameters: {
     "uftp.compression" : true,
   }
}
```

You can also schedule the transfer, by using

```
scheduledStartTime: <TIME in ISO "yyyy-MM-dd'T'HH:mm:ssZ" format>
```

as one of the *extraParameter* settings.

4.3.3.5.7 Files and metadata

Files on a storage are accessed via the `files` subresource, e.g.

```
BASE/storages/tmp/files/test.txt
```

would access the file *test.txt* on the storage with id *tmp*.

Depending on the media type used, GET and PUT fulfill different functions. Using JSON, the file's properties can be accessed or modified. Using `application/octet-stream` the actual binary file data can be downloaded or uploaded.

The file properties include metadata, which can also be modified (if the server is configured so that metadata is supported).

Searching the metadata index

If the storage supports metadata, the index can be searched using a `GET` request as follows:

```
$ curl -H "Accept: application/json" BASE/storages/{id}/search?q=querystring
```

The query string is appended as the `?q=...` URL query parameter. The server will reply with a JSON listing the files found:

```
{
  "status": "OK",
  "numberOfResults": 2,
  "_links": {
        "search-result-1": {
                "href": "https://..."
        },
        "search-result-2": {
                "href": "https://..."
        },
        ...
  },
```

(continues on next page)

(continued from previous page)

```
  "query": "query-string"
}
```

Triggering the automated metadata extraction

Triggering UNICORE's metadata extraction is done by a `POST` to an action URL for a file or a directory.

```
$ curl -X POST BASE/storages/{id}/files/some_directory/actions/extract --data-binary␣
↪@params.json -H "Content-Type: application/json"
```

This would extract metadata from the files in the *some_directory* directory.

The POSTed JSON can be empty `{}`, or it may contain extra parameters controlling the extraction process. Currently only a single parameter is supported, which controls the recursion depth for the extraction process, e.g.

```
{
  "depth": "2"
}
```

4.3.3.5.8 API Summary

URL	Method	Description
/storages	GET	Get a list of all storages
/storages	POST	Create a new storage
/storages/id	GET	Get a representation of a storage
/storages/id	DELETE	Destroy a storage. Depending on the storage type, this may delete the physical directory.
/storages/id/files/ filePath	GET (as application/json)	Get file list or file details
/storages/id/files/	PUT (as application/json)	Modify file properties (including metadata)
/storages/id/files/	GET (as application/ octet-stream)	Download a file
/storages/id/files/ filePath	PUT (as application/ octet-stream)	Upload a file
/storages/id/files/ dirPath	POST	Create a new directory
/storages/id/files/ filePath	DELETE	Delete a file or directory
/storages/id/actions/copy	POST	Copy file on the same storage resource
/storages/id/actions/ rename	POST	Rename file on the same storage resource
/storages/id/imports	POST	Create an client-server transfer (data upload)
/storages/id/exports	POST	Create an server-client transfer (data download)
/storages/id/transfers	POST	Create a server-server transfer
/storages/id/search?q= query-string	GET	Search the metadata index using the given query string

4.3.3.6 Storage factories

4.3.3.6.1 Synopsis

Storage factory endpoints allow the user to create Storage instances. Depending on the UNICORE server configuration there may be several types of Storages available, which may accept parameters to configure them.

The `storagefactories` resource is used to access the storage factory services available to the user, which then can be used to create new Storage endpoints.

These Storage endpoints are typically used as temporary resources, since they will eventually get cleaned up automatically by the server.

4.3.3.6.2 Creating storages

To create a Storage, you would POST to the appropriate endpoint:

```
$ curl -X POST BASE/storages/default_storage_factory --data-binary @params.json -H
↪"Content-Type: application/json"
```

The POSTed JSON can be empty `{}`, which will create a Storage pointing to some pre-configured directory on the HPC file system.

It may contain extra parameters controlling the type of storage that is created, and also extra parameters depending on the type. You can get more information about the possible types and parameters by inspecting the properties of the `storagefactories` endpoint, and the properties of the actual factories that are configured.

For example, the `storagefactories/default_storage_factory` endpoint supports a 'path' parameter, which controls the path the new Storage should access.

```
{
  "path": "/opt/data/"
}
```

4.3.3.6.3 API Summary

URL	Method	Description
/storagefactories	GET	Get a list of all storage factories
/storagefactories/id	GET	Get the representation of the given factory
/storagefactories/id	POST	Create a new storage endpoint

4.3.3.7 Jobs

4.3.3.7.1 Synopsis

The jobs resource ist used to access the jobs available to the user, to monitor and manage them. It is also possible to submit new jobs.

4.3.3.7.2 Job description format

The job description is described *here*.

4.3.3.7.3 Data management

The job description can contain data staging instructions for instructing the server to download input data before running the job, and for uploading results to some remote location when the job is done.

In addition, UNICORE supports client-controlled data staging, i.e. the client can optionally upload required data to the job's working directory. You can interact with the jobs working directory, which is a normal UNICORE storage. Thus all the storage functions above are applicable here as well.

In case you do want to upload any data, you can set an additional flag in the job description JSON:

```
{
 "haveClientStageIn": "true",
}
```

In this case, the full job submission sequence is:

1. submit job
2. upload data from client to server
3. start job

Note however that **small(!)** files can be embedded into the job description as well.

In the *Rest API Examples* you will find some typical job submission and management examples.

4.3.3.7.4 Starting, aborting or restarting jobs

To abort or restart a job, the REST API uses POST requests to special *action* links, see the API summary below.

4.3.3.7.5 API Summary

URL	Method	Description
/jobs?offset=	GET	Get a list of all jobs
/jobs	POST	Submit a new job. The site will be chosen automatically
/jobs/id	GET	Get a representation of the given job
/jobs/id/details	GET	Get batch system level information about the job (if available)
/jobs/id	DELETE	Destroy the job and its working directory.
/jobs/id/actions/start	POST	Start the job (in case the client did manual staging)
/jobs/id}/actions/abort	POST	Abort the job
/jobs/id/actions/restart	POST	Restart the job

4.3.3.7.6 Job properties

To get information about a job, do a `GET` request to the `/jobs/ID` endpoint.

Property	Values	Description
status	UNKNOWN, STAGINGIN, READY, QUEUED, RUNNING, STAGINGOUT, FAILED, SUCCESSFUL	The status of the job
statusMessage	string	status / error summary message
name	string	Job name as submitted, or "N/A"
log	string	Execution log
exitCode	integer	exit code of the application (if available)
queue	string	batch queue, or "N/A"
submissionTime	date-time	job submission time
terminationTime	date-time	time of automated job cleanup
currentTime	date-time	current server time
owner	X500 name	job owner (who submitted it)
acl	acl	access control list
submissionPreferences	map	user preferences (uid, gid) if applicable

4.3.3.8 Transfers

4.3.3.8.1 Synopsis

The `transfers` resource ist used to access the server-to-server file transfers available to the user, to monitor and manage them.

4.3.3.8.2 API Summary

URL	Method	Description
/transfers	GET	Get a list of all transfers
/transfers/id	GET	Get a representation of a transfer
/transfers/id	DELETE	Abort and destroy the transfer

4.3.3.9 Workflow

4.3.3.9.1 Synopsis

The *UNICORE Workflow system* can be accessed for workflow submission and management.

Basics like authentication, user preferences, security sessions work the exact same way as for a UNICORE/X server.

NOTE that the BASE url for workflows is:

`https://`*`gateway_url/WORKFLOW_SITENAME`*`/rest/workflows`

This base *workflows* URL is used to list the workflows available to the user and submit new ones.

4.3.3.9.2 API Summary

URL	Method	Description
/	GET	Get a list of all workflows accessible to the user
/?offset=	GET	List num workflows, starting at offset, with the given tags
/	POST application/json	Submit a new workflow and start processing it
/id	GET	Get a representation of the workflow with the given ID
/id/actions/abort	POST	Abort the workflow with the given ID
/id/actions/resume	POST	Resume the workflow with the given ID, if in state "Held"
/id/jobs?offset=	GET	Get a list of all jobs submitted for this workflow
/id/files	GET	Get a list of the workflow files for this workflow
/id/files	PUT	Modify the list of the workflow files for this workflow

4.3.3.9.3 Workflow submission

Submitting a workflow is done with a single `POST` with `Content-Type: application/json` to the base URL.

If successful, a new workflow instance will be created, and the URL returned in a HTTP location header.

If the workflow contains errors, the response body will contain a list of errors.

4.3.3.9.4 Workflow description format

The JSON workflow description as understood by the *Workflow engine* is described here: *Workflow description*.

4.3.3.9.5 Workflow properties

For each workflow, a `GET` request will retrieve a representation of the current workflow state, including the current state of the workflow variables.

To get a list of jobs that were submitted for the workflow, do a `GET` request to `BASE/{id}/jobs`.

To get a list of workflow files (registered names and physical locations) for the workflow, do a `GET` request to `BASE/{id}/files`.

4.3.3.9.6 Hold and resume

A workflow in *held* state (`waiting for user input`) can be resumed by a `POST` to the URL `BASE/{id}/actions/resume` with JSON content. The JSON can contain new values for any workflow variables, e.g.

```
{
"Variable1": "10",
"Variable2": "true",
}
```

4.3.4 Utility endpoints

4.3.4.1 Getting the server certificate

A GET request to `BASE/certificate` will retrieve the server's certificate in PEM format

```
$  curl -k BASE/certificate
-----BEGIN CERTIFICATE-----
MIIC3j...
-----END CERTIFICATE-----
```

4.3.4.2 Creating a token

The GET request to endpoint `BASE/token` allows you to create a JWT token signed by the server, that can be used for authentication later.

```
$  curl -k BASE/token
eyJh...
```

The token will (of course) give the same level of authentication that was used when creating it!

The endpoint accepts parameters as query parameters to the GET request

- `lifetime` token lifetime in seconds. If not set, the server's default is used (usually 300 seconds)
- `limited=true` will make the token only valid for the issuing server
- `renewable=true` will allow to get new tokens using the issued token

For example

```
$  curl -k -u demouser:test123 "BASE/token?lifetime=3600&limited=true&renewable=true"
eyJh...
```

4.3.5 Examples and HowTos

Can be found here *Rest API Examples*.

4.3.5.1 Rest API Examples

The following are some examples that use Python and the PyUNICORE library.

PyUNICORE can be installed from PyPI using

```
pip install -U pyunicore
```

The full documentation for PyUNICORE can be found here.

4.3.5.1.1 Basic job submission

Job submission and management

4.3.5.1.2 Storages and data management

Storages and data management

4.3.5.1.3 Workflow submission and management

Workflow submission and management

Job submission and management

```
#!/usr/bin/env python3

import json
import pyunicore.client as uc_client
import pyunicore.credentials as uc_credentials

# Base URL
base = "https://localhost:8080/DEMO-SITE/rest/core"
print ("Accessing REST API at ", base)

#
# setup authentication using username and password
#
credentials = uc_credentials.UsernamePassword("demouser", "test123")

#
# Create a client
#
site_client = uc_client.Client(credentials, base)

#
# Run a test job
#
job_description = {'Executable': "/bin/ls", 'Arguments' :["-lisa", "$HOME"], }

job = site_client.new_job(job_description)
print("Submitted: %s" % job)

# let's wait while the job is still running
job.poll()

# print job properties
```

(continues on next page)

(continued from previous page)

```
print (json.dumps(job.properties, sort_keys=True, indent=4))

# Accessing job outputs

# We can access the wob working directory and the stdout/stderr files

working_dir = job.working_dir
print (json.dumps(working_dir.properties, sort_keys=True, indent=4))

# Let's list all files in the working directory

for f in working_dir.listdir("."):
        print(f)

# Now let's download data from the 'stdout' file
stdout_content = working_dir.stat("stdout").raw().readlines()
for line in stdout_content:
        print(line)
```

Storages and data management

```
    #!/usr/bin/env python3

import json, time
import pyunicore.client as uc_client
import pyunicore.credentials as uc_credentials

# Base URL
base = "https://localhost:8080/DEMO-SITE/rest/core"
print ("Accessing REST API at ", base)
# setup authentication using username and password
credentials = uc_credentials.UsernamePassword("demouser", "test123")
# Create a client
site_client = uc_client.Client(credentials, base)

# List storages
all_storages = site_client.get_storages()
for storage in all_storages:
    print(storage)

# create two storages for testing
storage  = site_client.new_job({}).working_dir
storage2 = site_client.new_job({}).working_dir

# storage properties
print (json.dumps(storage.properties, sort_keys=True, indent=4))

# upload some data
storage.upload("test.txt", destination="test.txt")
```

(continues on next page)

(continued from previous page)

```
# List all files
for f in storage.listdir("."):
        print(f)

# Access a file
test_file = storage.stat("test.txt")
print (json.dumps(test_file.properties, sort_keys=True, indent=4))

# Download data from that file
file_content = test_file.raw().readlines()
for line in file_content:
        print(line)

# server-server copy: "storage2" pulls file from "storage"
source = storage._to_file_url("test.txt")
print("Source file URL: "+source)
transfer = storage2.receive_file(source, "copied-test.txt")
while transfer.is_running():
    time.sleep(1)
print (json.dumps(transfer.properties, sort_keys=True, indent=4))

# Access copied file
copied_file = storage2.stat("copied-test.txt")
print (json.dumps(copied_file.properties, sort_keys=True, indent=4))
```

Workflow submission and management

```
#!/usr/bin/env python3

import json
import pyunicore.client as uc_client
import pyunicore.credentials as uc_credentials

base = "https://localhost:8080/WORKFLOW/rest/workflows"
print ("Accessing Workflow REST API at ", base)

#
# setup authentication using username and password
#
credentials = uc_credentials.UsernamePassword("demouser", "test123")

#
# Create a client for the Workflow service
#
workflow_client = uc_client.WorkflowService(credentials, base)

#
# create the workflow description
```

(continues on next page)

(continued from previous page)

```
#

wf_json = {
        "activities": [
           {
                 "id": "date1",
                 "job": {
                          "Executable": "date",
                          "Job type": "INTERACTIVE",
                 }
          }
        ],
}

#
# create a new workflow instance on the server
#
workflow = workflow_service.new_workflow(workflow_description)

# see the workflow properties
print (json.dumps(workflow.properties, sort_keys=True, indent=4))
```

4.4 Job description format

A UNICORE job describes a *single job* on the target system.

By default, the job will be submitted to the batch system and run on a compute node. However, UNICORE supports *other job types* as well.

UNICORE uses a JSON format that allows you to specify the application or executable you want to run, arguments and environment settings, any files to stage in from remote servers and any result files to stage out. Depending on the client, the JSON may also contain additional instructions that are relevant to that client, so make sure to check the client manuals as well.

4.4.1 Overview

UNICORE's job description consists of several parts (their order does not matter):

- an `Imports` section listing data to be staged in to the job's working directory from remote storage locations (and/or the client's file system, if you use *UCC*)
- pre-processing
- a section describing the main executable
- post-processing
- an `Exports` section listing result files to be staged out to remote storage locations
- a `Resources` section stating any resource requirements like batch queue, job runtime or number of nodes
- a number of additional elements for setting the job name, or defining tags for the job

Here is a table listing the supported elements, these will be described in more detail below.

Tag	Type	Description
ApplicationName	String	Application name
ApplicationVersion	String	Application version
Executable	String	Command line
Arguments	List of strings	Command line arguments
Environment	Map of strings	Environment values
Parameters	Map	Application parameters
Stdout	String	Filename for the standard output (default: "stdout")
Stderr	String	Filename for the standard error (default: "stderr")
Stdin	String	Filename for the standard input (optional)
IgnoreNonZeroExitCode	"true" / "false"	Don't fail the job if app exits with non-zero exit code (default: false)
User precommand	String	Pre-processing
RunUserPrecommandOn-LoginNode	"true"/"false"	Pre-processing is done on login node (default: true)
UserPrecommandIg-noreNonZeroExitCode	"true"/"false"	Don't fail job if pre-command fails (default: false)
User postcommand	String	Post-processing
RunUserPostcommandOn-LoginNode	"true" / "false"	Post-processing is done on login node (default: true)
UserPostcommandIg-noreNonZeroExitCode	"true"/"false"	Don't fail job if post-command fails (default: false)
Resources	Map	The job's resource requests
Project	String	Accounting project
Imports	List of imports	Stage-in / data import
Exports	List of exports	Stage-out / data export
haveClientStageIn	"true" / "false"	Tell the server that the client does / does not want to send any additional files
Job type	'batch', 'on_login_node', 'raw', 'allocate'	Whether to run the job via the batch system ('batch', default) or on a login node ('on_login_node'), or as a batch job with user-specified file containing the batch batch system directives ('raw'), or to only 'allocate' resoures but not start anything
Login node	String	For 'on_login_node' jobs, select a login node by name, as configured server side. Wildcards '*' and '?' can be used)
BSS file	String	For 'raw' jobs, specify the relative or absolute file name of a file containing batch system directives. UNICORE will append the user executable.
Tags	List of strings	Job tags
Notification	String	URL to send job status change notifications to (via HTTP POST)
User email	String	User email to send notifications to (if the batch system supports it)
Name	String	Job name

4.4.2 Job elements

4.4.2.1 Job types

UNICORE supports four types if jobs. They are selected by the `Job type` element. If not given, `batch` is the default.

- `batch` (or `normal`) - this is the default. UNICORE submits the job to the batch system. After being scheduled, the specified executable is launched on the requested number of compute nodes. The job's resource requests (like number of nodes or requested run time) are taken from the job's `Resources` section.
- `on_login_node` (or `interactive`) - the specified executable will be launched on a login node. If you want, you can select the login node with the `Login node` element.
- `raw` - the job goes to the batch system, but the resources are taken from an additional file, which contains BSS directives (e.g.``#SBATCH …`` in the case of Slurm). The name of the file containing BSS directives is given via the `BSS file` element.
- `allocate` - this is basically the same as *batch*, but it only creates an allocation on the batch system, without launching any user tasks. You can submit tasks *into* the allocation later.

4.4.2.2 Specifying the executable or application

To directly call an executable on the remote system:

```
{
   "Executable": "/bin/date",
}
```

You can specify a UNICORE application (defined in the server's IDB) by name and (optional) version:

```
{
   "ApplicationName": "Date",
   "ApplicationVersion": "1.0",
}
```

Note the comma-separation and the curly braces.

4.4.2.3 Arguments and Environment settings

Arguments and environment settings are specified using a list of String values. Here is an example.

```
{

   "Executable": "/bin/ls",

   "Arguments": ["-l", "-t"],

   "Environment": [ "PATH=/bin:$PATH", "FOO=bar" ],

}
```

4.4.2.4 Argument sweeps

To create a sweep over an Argument setting by replacing the value by a sweep specification. This can be either a simple list:

```
"Arguments": [
 { "Values": ["-o 1", "-o 2", "-o 3"] },
],
```

or a range:

```
"Arguments": {
 "-o", { "From": "1", "To": "3", "Step" : "1" },
},
```

where the `From`, `To` and `Step` parameters are floating point or integer numbers.

4.4.2.5 Application parameters

In UNICORE, parameters for applications are often transferred in the form of environment variables. For example, the POVRay application has a large set of parameters to specify image width, height and many more. You can specify these parameters in a very simple way using the `Parameters` keyword:

```
{
  "ApplicationName": "POVRay",

  "Parameters": {
   "WIDTH": "640",
   "HEIGHT": "480",
   "DEBUG": "",
  },

}
```

Note that an *empty* parameter (which does not have a value) needs to be written with an explicit empty string due to the limitations of the JSON syntax.

4.4.2.6 Parameter sweeps

You can sweep over application parameters by replacing the parameter value by a sweep specification. The replacement can be either a simple list:

```
"Parameters": {
 "WIDTH": { "Values": ["240", "480", "960"] },
},
```

or a range:

```
"Parameters": {
 "WIDTH": { "From": "240", "To": "960", "Step": "240" },
},
```

where the `From`, `To` and `Step` parameters are floating point or integer numbers.

4.4.2.7 Pre- and postprocessing

In addition to the main executable (or application), a UNICORE job can contain pre- and/or postprocessing tasks that are run before / after the main executable.

The main elements for this are

- `User precommand` - this will be run after the data stage-in and before the main executable
- `User postcommand` - this will be run after the main executable and before starting to stage-out data

For example

```
{
  "User precommand": "./preprocessing.sh",

  "Executable": "./main.sh",

  "User postcommand": "./post-processing.sh"

}
```

The pre/post commands will be run on a login node by default. Failure of the pre/post commands will cause the job to fail.

The default behaviour can be modified via the following options:

- `RunUserPrecommandOnLoginNode:  'false'` - add pre processing as a prolog to the main job script
- `UserPrecommandIgnoreNonZeroExitCode` - don't fail the job if the pre command exits with a non-zero exit code
- `Login node` - select a preferred login node

and the same for the post command.

4.4.3 Job data management

In general, your job will require data files either from your client machine or from some remote location. Also, result files and other output files need to be accessible, or need to be exported (staged out) when the user task has finished executing.

Most of the job data management will be handled via the job's workspace, which is a unique, per-job directory that UNICORE creates when the job is submitted, and that is linked to the job. The job directory can be accessed at any time during the job's life time.

4.4.3.1 Jobs without client-controlled stage in

Some jobs require additional files from the client machine to be uploaded before the user task can be started.

Uploading LOCAL files is the responsibility of the client! Make sure to read the *client documentation* for more information on this topic.

To tell UNICORE/X that the client does not wish to send any local files, use the flag

```
"haveClientStageIn": "false",
```

Otherwise, the server will wait for an explicit *start* command (see the *REST API* spec for details) before submitting / executing the user task.

4.4.3.2 Importing files into the job workspace

To import (i.e. stage in) files from remote sites to the job's working directory on the remote UNICORE server, there's the `Imports` keyword. Here is an example of `Imports` section which demonstrates some of the possibilities.

```
{
  "Imports": [
    {
      "From": "UFTP:https://gw:8080/DEMO-SITE/rest/core/storages/HOME/files/testfile",
      "To":   "testfile"
    },
    {
      "From": "link:/work/data/testfile",
      "To":   "linked-file"
    },
    {
      "From": "link:/work/data/testfile",
      "To":   "copied-file"
    }
  ]
}
```

An Import can have the following elements.

```
{
  "From": "source-url",
  "To":   "target-path",
  "FailOnError": "true | false",
  "Permissions": "unix-style-rwx-permissions",
  "Credentials": { },
  "ExtraParameters": { },
  "Mode": "overwrite | append | nooverwrite",
}
```

The mandatory `From` element is a URL denoting the source of the file(s). UNICORE knows the following stage-in protocols:

- `https://` : download a file from an HTTP(s) server (UNICORE will try to *guess* whether the HTTP URL refers to a UNICORE file or not)
- `file://` : copy file(s) residing on the remote machine into the job dir
- `link://` : symlink a file/dir residing on the remote machine into the job dir
- `ftp://` : download a file from an FTP server
- `git:` : download the files from the given git repository
- `inline://` : ascii data is given directly, see below

The mandatory `To` element is the target path. As usual in UNICORE, this is relative to the base directory of the storage endpoint, in this case the job working directory. You can import into sub-directories, if these do not exist, they will be created as needed.

The optional flag `FailOnError` lets you you control if the job should continue even if an import operation fails. To do that, set this flag to `false`:

```
{
   "From":        "/work/data/fileName",
   "To":          "fileName",
   "FailOnError": "false",
}
```

The optional `Permissions` element allows you to explicitely set file permissions.

```
{
   "From":        "/work/data/fileName",
   "To":          "myscript.sh",
   "Permissions": "r-xr--r--"
}
```

(An abbreviated version like “r-x” also works).

The optional `Mode` element has three valid options: “overwrite” (default) will simply write the file. “append” will append if existing, and “nooverwrite” will fail if the file already exists.

The optional `Credentials` element can hold e.g. a required username/password and is discussed below.

The optional `ExtraParameters` element is used for protocol-specific extra settings.

4.4.3.3 Using *inline* data to import a file into the job workspace

For short import files, it can be convenient to place the data directly into the job description, which can speed up and simplify the job submission process.

Here is an example:

```
{
  "To":   "myscript.sh",
  "Data": [
    "this is some test data",
    "multi line data",
    "another line"
  ]
}
```

In this case, the `From` URL is not needed. If you give one, it has to start with `inline://`, the rest is not important.

Make sure to properly escape any special characters.

4.4.3.4 Staging in from *git*

You can stage-in a git repository, optionally allowing you to choose a particular commit, and to pass any required credentials.

For example

```
{
  "From": "git:https://github.com/github/testrepo.git",
  "To":   "testrepo",
  "ExtraParameters": {
    "commit" : "26fc7091"
```

(continues on next page)

(continued from previous page)

```
  },
  "Credentials": {
    "Password" : "some_api_token",
    "Username" : "test"
  }
}
```

If the git repo contains any submodules, these will be downloaded as well.

Please note that this operation will not result in a functional git repo, only the files will be downloaded.

4.4.3.5 Sweeping over a stage-in file

You can also sweep over files, i.e. create multiple batch jobs that differ by one imported file. To achieve this, replace the `From` parameter by list of values, for example:

```
{
  "From": [
    "https://gw:8080/DEMO-SITE/rest/core/storages/HOME/files/file1",
    "https://gw:8080/DEMO-SITE/rest/core/storages/HOME/files/file2",
    "https://gw:8080/DEMO-SITE/rest/core/storages/HOME/files/file3",
  ],
  "To": "fileName"
}
```

Note that only a single stage-in can be sweeped over in this way, and that this will not work with files imported from your local client machine.

4.4.4 Exporting result files from the job workspace

To export files from the job's working directory to remote storages, use the `Exports` keyword.

Note: Depending on the client, additional options exist, such as downloading files to your local machine.

Here is an example:

```
{
  "Exports": [
    {
      "From": "stdout",
      "To":   "https://gw:8080/DEMO-SITE/rest/core/storages/HOME/files/results/myjob/
↪stdout"
    },
    {
      "From": "results.dat",
      "To":   "https://gw:8080/DEMO-SITE/rest/core/storages/HOME/files/results/myjob/
↪results.dat"
    },
  ]
}
```

An Export can have the following elements.

```
{
  "From": "file-path",
  "To":   "target-URL",
  "FailOnError": "true | false",
  "Credentials": { },
  "ExtraParameters": { },
}
```

The mandatory `To` element is a URL denoting the target of the export. UNICORE knows the following stage-out protocols:

- `https://` : upload a file to an HTTP(s) server (UNICORE will try to *guess* whether the HTTP URL refers to a UNICORE server or not)
- `file://` : copy file(s) from the job dir to another directory on the remote machine
- `ftp://` : upload a file to an FTP server

4.4.4.1 Specifying credentials for data staging

Some data staging protocols supported by UNICORE require credentials such as username and password.

To pass username and password to the server, the syntax is as follows:

```
{
  "From": "ftp://someserver:25/some/file",
  "To": "input_data",
  "Credentials": {
    "Username": "myname",
    "Password": "mypassword"
  }
}
```

and similarly for exports.

You caan specify Token value for HTTPS data transfers, which will go into an HTTP "Authorization: Token …" header

```
{
  "From": "https://someserver/some/file",
  "To": "input_data",
  "Credentials": {
    "Token": "some_token"
  }
}
```

You may also specify an OAuth Bearer token for HTTPS data transfers, which will go into an HTTP "Authorization: Bearer …" header

```
{
  "From": "https://someserver/some/file",
  "To": "input_data",
  "Credentials": {
    "BearerToken": "some_token"
```

(continues on next page)

(continued from previous page)

```
  }
}
```

You can leave the token value empty, set to “”, if the server already has a valid Bearer token by some other means (e.g. from the incoming job submission call).

4.4.5 Redirecting standard input

If you want to have your application or executable read its standard input from a file, you can use the following

```
"Stdin": "filename",
```

then the standard input will come from the file named *filename* in the job working directory.

4.4.6 Resources

For batch jobs, you will want to control the resources allocated to your job. If you don’t do this, UNICORE will use the default settings configured by the site.

4.4.6.1 Specifying resources

Resources are requested using a `Resources` section:

```
{
  "Resources": {

    "Queue" : "fast",
    "Runtime": "12h",
    "Nodes": "8"

  }
}
```

UNICORE has the following built-in resource names:

Resource name	Description
Runtime	Job runtime (wall time) (in seconds, use “min”, “h”, “d” for other units)
Queue	Batch system queue (partition) to use
Nodes	Number of nodes
TotalCPUs	Total number of CPUs
CPUsPerNode	Number of CPUs per node
GPUsPerNode	Number of GPUs per node
Memory	Memory per node
Reservation	Reservation ID
NodeConstraints	Node constraints
QoS	Batch system QoS
Exclusive	Request exclusive use of the allocated node(s)

Sites may define additional, *custom* resources, which you can use, too.

4.4.6.2 Specifying an accounting project

If the system you're submitting to requires a project name for accounting purposes, you can specify the account (or project) you want to charge the job to using the `Project` element:

```
"Project" : "my_project",
```

(putting the "Project" into the "Resources" element will work, too)

4.4.7 Miscellaneous options

4.4.7.1 Umask

The umask controls the permissions of files created by the job and any processes that are launched from it. UNICORE's default will usually be "077" if not otherwise conigured. If you want to change the initial umask value, you can use the `Umask` keyword, e.g.

```
"Umask": "022",
```

(the value will interpreted as an octal string)

4.4.7.2 Job tags

To set job tags that help you find / filter jobs later, use the `Tags` keyword

```
"Tags": [ "production", "train1", "my_tag" ],
```

4.4.7.3 Specifying a URL for receiving notifications

The UNICORE/X server can send out notifications when the job enters the `RUNNING` and/or `DONE` state.

```
"Notification" : "https://your-service-url",
```

UNICORE/X will send an authenticated `HTTPS POST` message to this URL, with JSON content.

```
{
     "href" : "https://unicore-url/rest/core/jobs/job-uuid",
     "status" : "RUNNING",
     "statusMessage" : ""
}
```

The `status` field will be `RUNNING` when the user application starts executing, and `SUCCESSFUL` / `FAILED` when the job has finished.

```
{
     "href" : "https://unicore-url/rest/core/jobs/job-uuid",
     "status" : "SUCCESSFUL",
     "statusMessage" : "",
     "exitCode" : 0
}
```

Do not expect *realtime* behaviour here, as UNICORE has a certain delay (typically 30 to 60 seconds, depending on the server configuration) until *noticing* job status changes on the batch system.

If you want to verify that the sender of the notification is really UNICORE/X, you will need to check and validate the JWT Bearer token UNICORE/X sends in the Authorization header.

4.4.7.4 Advanced notification settings (UNICORE 9.2.0 and later)

By default, UNICORE will send notifications when the job enters `RUNNING` state or is done, and the status changes to `SUCCESSFUL` or `FAILED`.

For special use cases, you may need to use more detailed notification settings, for example when

- you want notifications on certain low-level (e.g. Slurm level) status changes
- you want notifications on more or other UNICORE-level status changes.

This advanced notification setup looks like this:

```
{
  "NotificationSettings" : {
    "URL": "https://your-service-url",
    "status": [ "STAGINGOUT", "SUCCESSFUL" ],
    "bssStatus": [ "CONFIGURING" ]
  }
}
```

where `status` is a list of UNICORE-level status strings, and `bssStatus` is a list of BSS-level status strings. If `status` is not given explicitly, the default (RUNNING, SUCCESSFUL, FAILED) are used.

The notifications sent by UNICORE contain the `href` job URL, and either a `bssStatus` field, or a `status`, depending on what triggered the notification message.

4.4.7.5 Specifying the job name

The job name can be set simply by

```
"Name": "Test job",
```

4.4.7.6 Specifying the user email for batch system notifications

Some batch systems support sending email upon completion of jobs. To specify your email, use

```
"User email": "foo@bar.org",
```

4.5 Workflow description

4.5.1 Introduction

This chapter provides an overview of the JSON workflow description that is supported by the *Workflow engine*. It will allow you to write workflows *by hand*, i.e. without using tools such as the Java or Python APIs.

After presenting all the constructs individually, several complete examples are given in *Examples*.

4.5.2 Overview and simple constructs

The overall workflow document has the following form

```
{
      "inputs": {},

       "activities": {},

       "subworkflows": {},

       "transitions": [],

       "variables": [],

       "notification": "optional_notification_url",

       "tags": ["tag1", "tag2", "..." ],
}
```

Activities, sub-workflows and transitions make up the workflow logic.

Both activities and sub-workflows are JSON maps (since UNICORE 9.0), where the key is the unique identifier of the element. The 8.x format of using JSON arrays with *id* elements is still supported.

Here is a simple example of two tasks that are to be run in a sequence:

```
{
  "activities": {

    "step1": {
      "job": {
        "Executable": "echo step1",
      }
    },

    "step2": {
      "job": {
        "Executable": "echo step2",
      }
    },
  },

  "transitions": [
```

(continues on next page)

(continued from previous page)

```
    {"from": "step1", "to": "step2" }
  ]
}
```

The remaining elements in the workflow description are:

- `inputs` allows to register external files with the workflow file catalog. See *Data handling*.
- `tags` is an optional list of initial tags, that can later be used to conveniently filter the list of workflows.
- `notification` (optional) denotes an URL to where UNICORE Workflow server will send a `POST` notification (authenticated via a JWT token signed by the Workflow server) when the workflow has finished processing.

Notification messages sent by the Workflow service have the following content:

```
{
  "href" : "workflow_url",

  "group_id": "id of the workflow or sub-workflow",

  "status": "...",

  "statusMessage": "..."
}
```

Both of these are analogous to their conterparts for single jobs in UNICORE.

In the next sections the elements of the workflow description will be discussed in detail.

4.5.3 Activities

Activity elements have the following form:

```
"id": {
   "type": "...",
  ...
}
```

The `id` must be UNIQUE within the workflow. There are different types of activity, which are distinguished by the `type` element.

- `START` denotes an explicit start activity. If no such activity is present, the processing engine detect the proper starting activities.
- `JOB` denotes a executable (job) activity. In this case, the job sub element holds the JSON job definition (if a `job` element is present, you may leave out the `type`).
- `MODIFY_VARIABLE` allows to modify a workflow variable. An option named `variable_name` identifies the variable to be modified, and an option `expression` holds the modification expression in the Groovy programming language syntax (see also the *variables section* later).
- `SPLIT`: this activity can have multiple outgoing transitions. All transitions with matching conditions will be followed. This is comparable to an "*if() … if() … if()*" construct in a programming language.
- `BRANCH`: this activity can have multiple outgoing transitions. The transition with the first matching condition will be followed. This is comparable to an "*if() … elseif() … else()*" construct in a programming language.
- `MERGE` merges multiple flows without synchronising them.

- SYNCHRONIZE merges multiple flows and synchronises them.
- HOLD stops further processing of the current flow until the client explicitly sends a continue message.

4.5.4 Subworkflows

The workflow description allows nested sub workflows, which have the same formal structure as the main workflow (without the tags and inputs). There is an additional type element that is used to distinguish the different control structure types.

```
{

  "id": "unique_id",

  "type": "...",

  "variables": [],

  "activities": {},

  "subworkflows": {},

  "transitions": [],

  "notification" : "optional_notification_url",

}
```

4.5.5 Job activities

Job activities are the basic executable pieces of a workflow. The embedded JSON job definition will be sent to an execution site (UNICORE/X) for processing.

```
{
  "id": "unique_id",

  "type": "job",

  "job": {

    "... standard UNICORE job ...": ""

  },

  "options": {  },
}
```

The execution site is specified by the optional Site name element in the job

```
{
    "id": "unique_id", "type" : "job",

```

(continues on next page)

(continued from previous page)

```
    "job": {

      "Site name": "DEMO-SITE",

    },
}
```

Note: There is currently no form of *brokering* in place, it is up to the user to select an execution site.

The job description is covered in detail in *Job description format*.

The processing of the job can be influenced using the (optional) `options` sub-element. Currently the following options (*key-value*) can be used:

- `IGNORE_FAILURE` if set to `true`, the workflow engine will ignore any failure of the task and continue processing as if the activity had been completed successfully.

  Note: This has nothing to do with the exit code of the actual UNICORE job! Failure means for example data staging failed, or no matching target system for the job could be found.

- `MAX_RESUBMITS` set to an integer value to control the number of times the activity will be retried. By default, the workflow engine will re-try three times (except in those cases where it makes no sense to retry).

For example,

```
{
 "id": "unique_id",

 "job" : {

  ... standard UNICORE job ...

 },

 "options": { "IGNORE_FAILURE": "true",  },
}
```

If you need to pass on user preferences to the site, e.g. for selecting your primary group, or choosing between multiple user IDs, you can specify this in the `job` element like this:

```
...

"job": {

   "User prefences": {
     "uid":   "hpcuser21",
     "group": "hpc",
   }

}
...
```

where the allowed field names are `role`, `uid`, `group` and `supplementaryGroups`.

4.5.6 Data handling

One of the most common tasks is linking the output of one activity to another activity for further processing. The UNICORE workflow system supports this by providing a per-workflow file catalog, where jobs can reference files with special URIs starting with `wf:`.

Jobs can register outputs with the file catalog using stage-out directives, for example,

```
Exports: [
  { "From": "stdout", "To": "wf:step1_stdout" }
]
```

will register the `stdout` file under the name *wf:step1_stdout* (note that the file will not be copied anywhere).

Later jobs can reference files from the catalog using stage-in directives, for example,

```
Imports: [
  { "From": "wf:step1_stdout", "To": "input_file" }
]
```

The workflow engine will take care of resolving the `wf:...` reference to the actual physical location.

Apart from registration of files in jobs, the user can also *manually* register files using the `inputs` section of the main workflow:

```
"inputs": {
  "wf:input_data_1": "https://some_storage/somefile.pdf",
  "wf:input_params": "https://some_storage/parameters.txt"
}
```

For an example of a workflow, have a look at *Simple two-step workflow with data dependency*.

The Workflow REST API allows you to list (and modify) the file catalog via the `BASE/{id}/files` endpoint.

4.5.7 Transitions and conditions

The basic flow of control in a workflow is handled using transition elements. These reference from and to activities or subflows, and may have conditions attached. If no condition is present, the transition is followed unconditionally, otherwise the condition is evaluated and the transition is followed only if the condition matches (i.e. evaluates to true).

The syntax for a Transition is as follows:

```
{

  "from" : "from_id",

  "to" : "to_id",

  "condition": "expression"

}
```

The `from` and `to` elements denote activity or subworkflow id's.

An activity can have outgoing (and incoming) transitions. In general, all outgoing transitions (where the condition is fulfilled) will be followed. The exception is the `Branch` activity, where only the first matching transition will be followed.

The optional condition element is a string-valued expression. The workflow engine offers some pre-defined functions that can be used in these expressions. For example, you can use the exit code of a job, or check for the existence of a file within these expressions.

- `eval(expr)` Evaluates the expression *expr* in Groovy syntax, which must evaluate to a boolean. The expression may contain workflow variables.
- `exitCodeEquals(activityID, value)` Allows to compare the exit code of the UNICORE job associated with the Activity identified by *activityID* to *value*.
- `exitCodeNotEquals(activityID, value)` Allows to check the exit code of the UNICORE job associated with the Activity identified by *activityID*, and check that it is different from *value*.
- `fileExists(activityID, fileName)` Checks that the working directory of the UNICORE job associated with the given Activity contains a file *fileName*.
- `fileLengthGreaterThanZero(activityID, fileName)` Checks that the working directory of the UNICORE job associated with the given Activity contains the named file, which has a non-zero length.
- `before(time)` and `after(time)` check whether the current time is before or after the given time (in *yyyy-MM-dd HH:mm* format).
- `fileContent(activityID, fileName)` Reads the content of the named file in the working directory of the job associated with the given Activity and returns it as a string.

4.5.8 Using workflow variables

Workflow variables need to be declared using an entry in the `variables` array before they can be used.

```
{

  "name": "...",

  "type": "...",

  "initial_value": "..."

}
```

Currently variables of type `STRING`, `INTEGER` , `FLOAT` and `BOOLEAN` are supported.

Variables can be modified using an activity of type `MODIFY_VARIABLE`.

For example, to increment the value of the *COUNTER* variable, the following Activity is used

```
{

 "type": "MODIFY_VARIABLE",

 "id": "incrementCounter",

 "variable_name": "COUNTER",
```

(continues on next page)

(continued from previous page)

```
 "expression": "COUNTER += 1;"

}
```

The `expression` contains an expression in Groovy syntax (which is very close to Java).

The workflow engine will replace variables in job data staging sections and environment definitions, allowing to inject variables into jobs. Examples for this mechanism will be given in the *Examples* section.

4.5.9 Loop constructs

Apart from graphs constructed using `activity` and `transition` elements, the workflow system supports special looping constructs, *for-each*, *while* and *repeat-until*, which allow to build complex workflows.

4.5.9.1 *While* and *repeat-until* loops

These allow to loop a certain part of the workflow while (or until) a condition is met. A *while* loop looks like this

```
{
 "id": "while_example",

 "type" : "WHILE",

 "variables" : [
  {
        "name": "C",
        "type": "INTEGER",
        "initial_value": "1",
  }
 ],

 "body":
  {
   "activities": {
     "task" : {
            "job": { ... }
    },
    "mod": {
          # this modifies the variable used in the 'while'
          # loop's exit condition
          "type": "MODIFY_VARIABLE",
          "variable_name": "C",
          "expression": "C++;",
    }
   },

   "transitions: [
         {"from": "task", "to": "mod"}
   ]
  },
```

(continues on next page)

(continued from previous page)

```
 "condition": "eval(C<5)",

}
```

The necessary ingredients are that the loop's body modifies the loop variable ("C" in the example), and the exit condition eventually terminates the loop.

For a full workflow example, see *While loop example using workflow variables*.

Completely analogously to the *while* loop, a *repeat-until* loop is constructed, the only syntactic difference is that the subworkflow now has a different type element:

```
{
  "id": "repeat_example",

  "type": "REPEAT_UNTIL",

  ...
}
```

Semantically, the *repeat*-loop will always execute the body at least once, since the condition is checked after executing the body, while in the *while* case, the condition will be checked before executing the body.

4.5.9.2 *For-each* loop

The *for-each* loop is a complex and powerful feature of the workflow system, since it allows parallel execution of the loop body, and different ways of building the different iterations. Put briefly, one can loop over variables (as in the *while* and *repeat-until* case), but one can also loop over enumerated values and (most importantly) over file sets.

The basic syntax is:

```
{
 "id": "for_each_example",

 "type": "FOR_EACH",

 "iterator_name": "IT",

 "body": {

 },

# define range to loop over

 "values": [...],

# OR variables

 "variables": [...],

# OR files

```

(continues on next page)

(continued from previous page)

```
 "filesets": [...],

  # with optional chunking
 "chunking":

}
```

The `iterator_name` element allows to control how the *loop iterator variable* is to be called, by default it is named *IT*.

4.5.9.3 The `values` element

Using value, iteration over a fixed set of strings can be defined. The main use for this is parameter sweeps, i.e. executing the same job multiple times with different arguments or environment variables.

```
"values": ["1", "2", "3", ],
```

The following variables are set where `IT` is the loop `iterator_name` defined in the for group as shown above:

- `IT` is set to the current iteration index (1, 2, 3, . . .)
- `IT_VALUE` is set to the current value

4.5.9.4 The `variables` element

The `variables` element allows to define the iteration range using one or more variables, similar to a for-loop in a programming language.

```
"variables: [
 {
   "variable_name": "X",
   "type": "INTEGER",
   "start_value": "0",
   "expression": "Y++",
   "end_condition": "Y<2"
 },
 {
   "variable_name": "Y",
   "type": "INTEGER",
   "start_value": "0",
   "expression": "Y++",
   "end_condition": "Y<2"
 }
],
```

The sub-elements should be self-explanatory.

Note that you can use more than one variable range, allowing you to quickly create things like parameter studies.

The following variables are set where `IT` is the loop `iterator_name` defined in the for group as shown above:

- `IT` is set to the current iteration index (1, 2, 3, . . .)
- `IT_VALUE` is set to the current value

4.5.9.5 The `file_sets` element

This variation of the *for-each* loop, allows to loop over a set of files, optionally chunking together several files in a single iteration.

The basic structure of a file set definition is this:

```
"file_sets": [

 {
  "base": "...",
  "include": [ "..." ],
  "exclude": [ "..." ],
  "recurse": "true|false",
  "indirection": "true|false",
},

]
```

The base element defines a base of the filenames, which will be resolved at runtime, and complemented according to the include and/or exclude elements. The `recurse` attribute allows to control whether the resolution should be done recursively into any subdirectories. The indirection attribute is explained below.

For example, to recursively collect all PDF files (except two files named *unused*.pdf*) in a certain directory on a storage:

```
"file_sets": [

 {
        "base": "https://mysite/rest/core/storages/my_storage/files/pdf/</s:Base>
        "include": [ "*.pdf" ],
        "exclude": [ "unused1.pdf", "unused2.pdf", ],
        "recurse": "true"
  }

]
```

The following variables are set where `IT` is the loop `iterator_name` defined in the for group as shown above:

- `IT` is set to the current iteration index (1, 2, 3, ...)
- `IT_VALUE` is set to the current full file path
- `IT_FILENAME` is set to the current file name (last element of the path)

4.5.9.6 Indirection

Sometimes the list of files that should be looped over is not known at workflow design time, but will be computed at runtime. Or, you wish simply to list the files in a file, and not put them all in your workflow description. The `indirection` attribute on a FileSet allows to do just that. If `indirection` is set to `true`, the workflow engine will load the given file(s) in the fileset at runtime, and read the actual list of files to iterate over from them. As an example, you might have a file filelist.txt containing a list of UNICORE file URLs:

```
https://someserver/file1
https://someserver/fileN
...
```

and the fileset

```
{
   "indirection": "true",
   "base": "https://someserver/rest/core/storages/mystorage/files/</s:Base>
   "include": [ "filelist.txt" ],
}
```

You can have more than one file list.

4.5.9.7 Chunking

Chunking allows to group sets of files into a single iteration, for example for efficiency reasons.

A chunk is either a certain number of files, or a set of files with a certain total size.

```
"chunking": {
 "chunksize": ... ,
 "type": "NORMAL|SIZE",
 "filename_format": "...,
 "expression": "... formula to compute chunksize ...",
}
```

The `chunksize` element is either the number of files in a chunk, or (if type is set to `SIZE`) the total size of a chunk in kbytes.

For example,

- To process 10 files per iteration:

  ```
  "chunking":
  {
    "chunksize": "10",
  }
  ```

- To process 2000 kBytes of data per iteration:

  ```
  "chunking":
  {
    "chunksize": "2000",
    "type": "SIZE"
  }
  ```

The `chunksize` can also be computed at runtime using the expression given in the optional expression element. In the expression, two special variables may be used. The `TOTAL_NUMBER` variable holds the total number of files iterated over, while the `TOTAL_SIZE` variable holds the aggregated size of all files in kbytes. The script must return an integer-valued result. The `type` element is used to choose whether the chunk size is interpreted as number of files or data size.

For example, to choose a larger chunksize if a certain total file size is exceeded:

```
"chunking": {
  "expression": "if(TOTAL_SIZE>50*1024)return 5*1024 else return 2048;"
  "type": "SIZE"
}
```

The optional `filename_format` allows to control how the individual files (which are staged into the job directory) should be named. By default, the index is prepended, i.e. an import statement like

```
"Imports": [{ "From": "${IT_VALUE}", "To" : "infile.txt" }]
```

would result in *1_infile.txt* to *N_infile.txt* in each chunk. In the `filename_format` pattern you can use the variables `{0}`, `{1}` and `{2}`, which denote the index, filename without extension and extension respectively.

```
{0} = 1, 2, 3, ...
{1} = "infile"
{2] = "txt"
```

For example, if you have a set of PDF files, and you want them to be named *file_1.pdf* to *file_N.pdf*, you could use the pattern:

```
"filename_format": "file_{0}.pdf"
```

which would ignore the original filename in the `To` field completely. Or, if you prefer to keep the existing extensions, but append an index to the name, use

```
"filename_format": "{1}{0}.{2}"
```

which would result in filenames like below:

```
inputfile1.txt
inputfile2.txt
...
```

You can also keep the original filenames by setting:

```
"Imports": [{ "From": "${IT_VALUE}", "To" : "${IT_ORIGINAL_FILENAME}"}]
```

The following variables are set where `IT` is the loop `iterator_name` defined in the for group as shown above:

- `IT` is set to the current iteration index (1, 2, 3, . . .)
- `IT_VALUE` is set to the current full file path
- `IT_ORIGINAL_FILENAME_x` is set to the current file name (last element of the path)
- `IT_ORIGINAL_FILENAMES` is set to a ";"-separated list of all the file names (last elements of the paths) in the current chunk

4.5.10 Examples

This section collects a few simple example workflows. They are intended to be submitted using *UNICORE Commandline Client*.

4.5.10.1 Simple *two-step* workflow with data dependency

This example shows how to link output from one task to the input of another task using the internal file catalog.

The first task, *step1*, registers its `stdout` with the file catalog under the name `wf:step1_out`, and the second task, *step2*, pulls that file in for further processing.

```
{
  "activities": {

    "step1": {
      "job": {
        "ApplicationName": "Date",
        "Exports": [
          {"From": "stdout", "To": "wf:step1_out"}
        ]
      }
    },

    "step2": {
      "job": {
        "Executable": "md5sum",
        "Arguments": ["infile" ],
        "Imports": [
          { "From": "wf:step1_out", "To": "infile"}
        ]
      }
    }

  },

  "transitions": [
    {"from": "step1", "to": "step2" }
  ]
}
```

4.5.10.2 Simple *diamond* graph

This example shows how to use transitions for building simple workflow graphs. It consists of four *Date* jobs arranged in a diamond shape, i.e. *date2a* and *date2b* are executed (more or less) simultaneously.

```
{
 "activities": {

   "date1": {
      "job": { "ApplicationName": "Date" }
   },

   "date2a": {
      "job": { "ApplicationName": "Date" }
   },

   "date2b": {
```

(continues on next page)

(continued from previous page)

```
      "job": { "ApplicationName": "Date" }
   },

   "date3": {
      "job": { "ApplicationName": "Date" }
   }

 },

 "transitions": [
   {"from": "date1", "to": "date2a" },
   {"from": "date1", "to": "date2b" },
   {"from": "date2a", "to": "date3" },
   {"from": "date2b", "to": "date3" }
 ]
}
```

4.5.10.3 Conditional execution in an *if-else* construct

Transitions from one activity to another may be conditional, which allows all sorts of *if-else* constructs. Here is a simple example:

```
{

  "activities": {

    "branch": { "type": "BRANCH" },

    "if-job": {
       "job": { "ApplicationName": "Date" }
    },

    "else-job": {
       "job": { "ApplicationName": "Date" }
    }

  },

  "transitions": [
    {"from": "branch", "to": "if-job", "condition": "2+2==4"},
    {"from": "branch", "to": "else-job" }
  ]

}
```

Here we use the `BRANCH` activity which will only follow the first matching transition.

4.5.10.4 *While* loop example using workflow variables

The next example shows some uses of workflow variables in a *while* loop. The loop variable *C* is copied into the job's environment. Another possible use is to use workflow variables in data staging sections, for example to name files.

```
{

  "activities":{},

  "subworkflows": {

   "while-example": {

        "type": "WHILE",

        "variables": [
         {
           "name": "C",
           "type": "INTEGER",
           "initial_value": "0"
         }
        ],

        "condition": "C<5",

        "body": {

           "activities": {

           "job": {
                 "job": {
                        "Executable": "echo",
                        "Arguments": ["$TEST"],
                        "Environment": ["TEST=${C}"],
                        "Exports": [
                          { "From": "stdout", "To": "wf:/out_${C}" }
                        ]
                  }
            },

            "mod": {
                   "type": "MODIFY_VARIABLE",
                   "variable_name": "C",
                   "expression": "C++"
            }

           },

           "transitions": [
                  {"from": "job", "to": "mod" }
           ]
        }
```

(continues on next page)

(continued from previous page)

```
    }

  }

}
```

4.5.10.5 *For-each* loop example

The next example shows how to use the *for-each* loop to loop over a set of files. The jobs will stage-in the current file. Also, the name of the current file is placed into the job environment.

```
{

  "subworkflows": {

   "for-example": {
          "type": "FOR_EACH",
          "iterator_name": "IT",

        "body":
          {
                "activities": {

                  "job": {
                        "id": "job",
                        "job": {
                         "Executable": "echo",
                         "Arguments": ["processing: ", "$NAME"],
                         "Environment": ["NAME=${IT_FILENAME}"],
                         "Imports": [
                           {"From": "${IT_VALUE}", "To": "infile"}
                         ],
                         "Exports": [
                           {"From": "stdout", "To": "wf:/out_${IT}"}
                         ]
                        }
                  }

                },

          },

        "filesets": [
          {
                "base": "https://mygateway.de:7700/MYSITE/rest/core/storages/my_storage/
↪",
                "include": ["*"],
          }
        ]

   }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

4.6 Data-triggered processing

This document describes UNICORE’s data-triggered, rule-oriented processing feature.

4.6.1 What is 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.

This style of processing is storage-oriented, i.e. is defined by the properties of a UNICORE storage endpoint, and by files accessible on that storage endpoint.

4.6.2 Setting up data-triggered processing

Depending on the server configuration, data triggered processing is probably disabled on the standard storages (HOME, ROOT, etc).

If the UNICORE server provides a “storage factory” service, you can create an endpoint with data-triggered processing enabled, and you can select the path

For example using UCC:

```
ucc create-storage path=/path/to/your/directory enableTrigger=true
```

4.6.3 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. It can be modified at any time.

The file is expected to be in JSON format, and has the following elements:

```
{
 "DirectoryScan": {

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

 },
```

(continues on next page)

(continued from previous page)

```
 "Rules": [ ],

 "Enabled": "true | false",

 "Logging": "true | false"

}
```

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 optional Interval element allows you to control the scan interval. A numerical value (seconds), or a time value with unit such as "1h" can be used here.

The optional Enabled element allows you to (temporarily) disable the processing, if you wish.

The optional Logging element allows you to disable the writing of log files. Logging is enabled by default, see below. Log files will be written to a directory .UNICORE_data_processing in the base directory of the storage endpoint.

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

4.6.3.1 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": "example",
      "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.

4.6.3.1.1 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
- `UC_FILES` : (batched actions only) all the newly detected files, relative to the base directory

4.6.3.1.2 Job action

This type of action will be executed for a single new file, and defines a UNICORE job in the usual *job description syntax*.

```
"Action":
{
  "Job": { ... }
}
```

The `Job` element contains a UNICORE job in the usual *job description syntax*.

4.6.3.1.3 Batched job action

This type of action will be executed for a whole set of newly detected files.

```
"Action":
{
  "BatchedJob": { ... }
}
```

The `BatchedJob` element contains a UNICORE job in the usual *job description syntax*.

4.6.3.1.4 Metadata extraction

```
"Action":
{
  "Extract": { ... }
}
```

This action will extract metadata from all the newly detected files. The contents of the `Extract` element are currently unused.

4.6.3.1.5 Sending notifications

```
"Action":
{
  "Notification": "https://url-to-notification-receiver"
}
```

This action will send out a HTTP POST request in JSON format to the specified URL. The JSON will contain information about the new files, as well as the base directory that is being watched.

An example notification could look like this:

```
{
  "href": "https://unicorex-server-url/rest/core",
  "directory": "/path/to/watched/directory",
  "files": [
    "/path-to-new-file1",
    "/path-to-new-file2"
  ]
}
```

(File paths are relative to the base directory!)

4.6.3.2 Logging

If new files are detected, and rules are executed, the server will write a short log file to a directory ".UNICORE_data_processing".

4.6.4 Stopping the processing

Since data-triggered processing is tied to the storage instance, you can stop it by sending an empty REST POST to an URL on the storage, e.g.

```
ucc rest post "{}" 'storage-URL'/actions/stop-processing
```

Destroying the storage instance will also stop the processing (but not delete any files).

```
ucc rest delete "{}" 'storage-URL'
```

4.6.5 Example

As an example, we setup a task that generates checksums for all new files that are detected in the *incoming* directory.

The `.UNICORE_Rules` file could look like this:

```
{
  "DirectoryScan": {
    "IncludeDirs": [
       "incoming"
    ],
},
```

(continues on next page)

(continued from previous page)

```
  "Rules": [
    {
      "Name": "generate-hash",
      "Match": ".*",
      "Action": {
         "Job": {
            "Executable": "sha256sum ${UC_FILE_PATH}",
            "Exports": [
              {
                "From": "stdout",
                "To": "file://${UC_BASE_DIR}/checksums/${UC_FILE_NAME}.sha"
              }
            ]
          }
      }
    }
  ]
}
```

CHAPTER

FIVE

ADMINISTRATOR DOCUMENTATION

- ***Gateway*** An optional server component that provides a reverse https proxy, allowing you to run several backend servers (*UNICORE/X*, *Registery*, . . .) behind a single address.
- ***UNICORE/X*** The central server component of a typical UNICORE installation that provides *REST APIs* for job management and data access services for a single compute cluster (or just a file system).
- ***TSI*** The **T**arget **S**ystem **I**nterface (TSI) server is used to interface to a resource manager such as Slurm and to access files on the cluster.
- ***XUUDB*** An optional service, that is best suited as a per-site service, providing attributes for multiple UNICORE/X-like services at a site. The XUUDB maps a UNICORE user identity (which is formally an X.500 **d**istinguished **n**ame (DN)) to a set of attributes which are typically used to provide local account details (uid, gid(s)) and commonly also to provide authorization information, i.e. the user's role.
- ***Workflow Service*** provides advanced workflow processing capabilities using UNICORE resources. The Workflow service provides graphs of activities including high-level control constructs (*for-each*, *while*, *if-then-else*, etc.), and submits and manages the execution of single UNICORE jobs.
- ***Registry*** The Registry server is a specially configured *UNICORE/X server* which provides the information about available services to clients and other services.
- **UFTPD** The UNICORE File Transfer Server for high performance data transfer.

5.1 Gateway

The UNICORE Gateway is an (optional) server component that provides a reverse https proxy, allowing you to run several backend servers (*UNICORE/X*, *Registry*, . . .) behind a single address. This helps with firewall configuration, requiring only a **single open port** (a similar effect can be achieved using other http servers that can act as a reverse proxy, such as Apache httpd or nginx).

The second functionality of the Gateway is (optional) authentication of incoming requests. Connections to the Gateway are made using SSL, so the Gateway can be configured to check whether the caller presents a certificate issued by a trusted authority. Information about the client is forwarded to services behind the Gateway in UNICORE proprietary format (as a HTTP header).

The Gateway will forward the IP address of the client to the back-end server.

Last not least, the Gateway can be configured as a HTTP load balancer.

Gateway Manual Installation and Operating the Gateway.

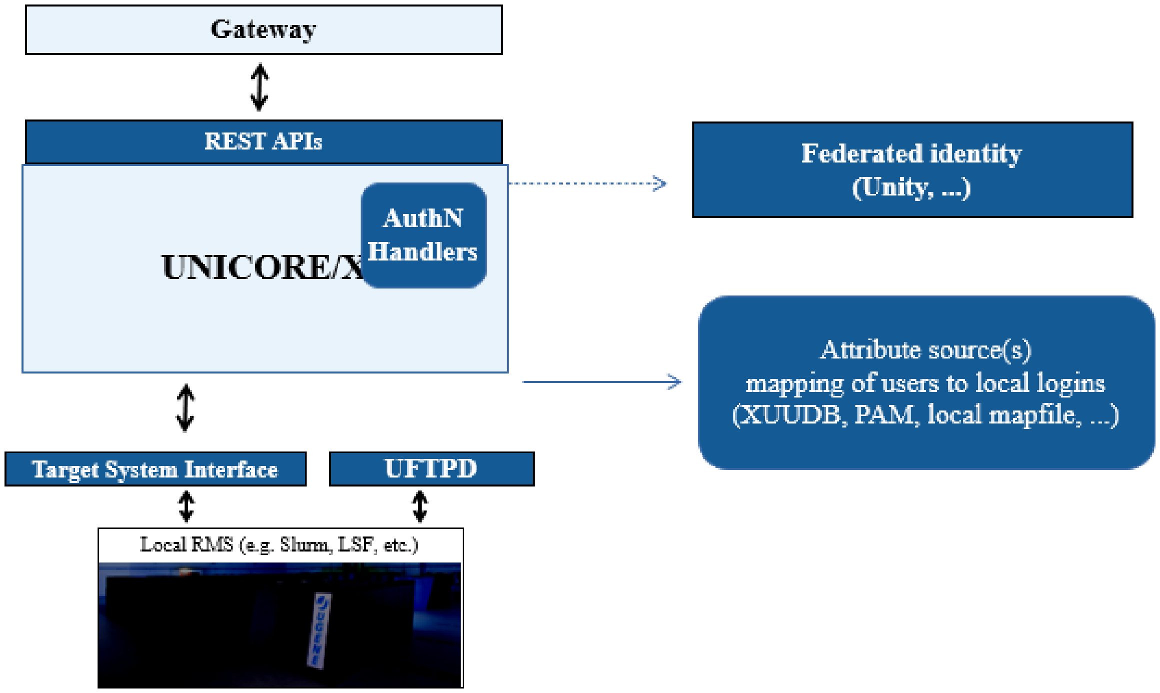

Gateway
REST APIs
UNICORE/X
AuthN Handlers
Federated identity (Unity, ...)
Attribute source(s) mapping of users to local logins (XUUDB, PAM, local mapfile, ...)
Target System Interface
UFTPD
Local RMS (e.g. Slurm, LSF, etc.)

Fig. 5.1: UNICORE Components

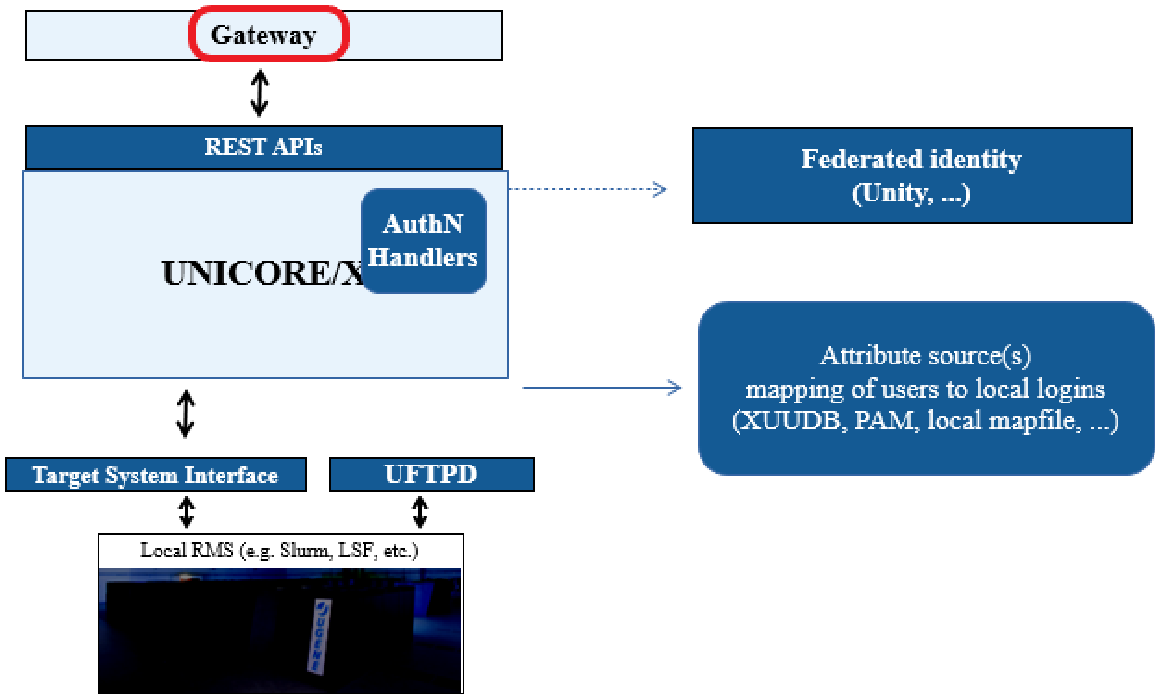

Gateway
REST APIs
UNICORE/X
AuthN Handlers
Federated identity (Unity, ...)
Attribute source(s) mapping of users to local logins (XUUDB, PAM, local mapfile, ...)
Target System Interface
UFTPD
Local RMS (e.g. Slurm, LSF, etc.)

Fig. 5.2: UNICORE Gateway Server

5.1.1 Gateway Manual

The Gateway is the entry point into a UNICORE site, routing HTTPS traffic to servers like *UNICORE/X*. It forwards client traffic to the intended destination, optionally authenticating the client. The Gateway receives the reply and sends it back to the client. In this way, only a single open port in a site's firewall has to be configured.

Attention: LIMITATIONS

The Gateway is not a complete HTTP reverse proxy implementation. For example, it is not possible to run a full, complex web application *behind* the Gateway, especially not if protocols like WebSocket are used.

In effect, traffic to a *virtual* URL, e.g. *https://mygateway:8088/Alpha* is forwarded to the real URL, e.g. *https://host1:7777*.

The mappings of virtual URL to real URL for the available sites are listed in a configuration file `connections.properties`. Additionally, the Gateway supports dynamic registration of sites.

The second functionality of the Gateway is (optional) authentication of incoming requests. Connections to the Gateway are made using SSL, so the Gateway can be configured to check whether the caller presents a certificate issued by a trusted authority. Information about the client is forwarded to services behind the Gateway in UNICORE proprietary format (as a SOAP or HTTP header).

The Gateway will forward the IP address of the client to the back-end server.

Last not least, the Gateway can be configured as a HTTP load balancer.

Important: IMPORTANT NOTE ON PATHS

Depending on the installation method, the paths to various Gateway files are different. If installing using a distribution-specific package the following paths are used:

```
CONF=/etc/unicore/gateway
BIN=/usr/sbin
LOG=/var/log/unicore/gateway
```

If installing using the portable bundle all Gateway files are installed under a single directory. Path prefixes then are as follows, where *INST* is a directory where the Gateway was installed:

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

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

5.1.1.1 Installation

The UNICORE Gateway is distributed in the following formats:

1. As a part of platform independent installation bundle called UNICORE Server bundle. The UNICORE Server bundle is provided as a tar package and includes a command line installer.
2. As a binary, platform-specific package available currently for RedHat (Centos) and Debian platforms on the UNICORE project website at sourceforge. Those packages are not tested on all possible platforms, but should work without any problems with other versions of similar distributions, e.g. SL, Centos, or Fedora.

5.1.1.1.1 Prerequisites

To run, the Gateway requires Java (JRE headless is sufficient) in version 11 or later. We recommend using OpenJDK.

5.1.1.1.2 Installation from the Server bundle

Download the server bundle from the UNICORE project website.

Please review the `README` file available after extracting the bundle. You don't have to change any defaults as the Gateway is installed by default.

You should create and use a system user (e.g. *unicore*) to install and run the gateway. For security reasons, **do not** run the Gateway as the *root* user.

5.1.1.1.3 Installation from a Linux package (rpm or deb)

Use your distribution's package manager to install.

5.1.1.2 Upgrading

The general update procedure is presented below, with possible variations:

1. Stop the old Gateway.
2. Update the server package. This step mostly applies for RPM/DEB managed installations. For Quickstart installation it is enough to replace the `*.jar` files with the new ones.
3. Start the newly installed Gateway.
4. Verify log file and fix any problems reported.

5.1.1.3 Configuration

The Gateway is configured using a set of configuration files, which reside in the `CONF` subdirectory.

5.1.1.3.1 Java and environment settings: `startup.properties`

This file contains settings related to the Java VM, such as the Java command to use, memory settings, library paths, etc.

5.1.1.3.2 Configuring sites: `connections.properties`

This is a simple list connecting the names of sites and their physical addresses. An example is:

```
DEMO-SITE = https://localhost:7777
REGISTRY = https://localhost:7778
```

If this file is modified, the Gateway will re-read it at runtime, so there is no need to restart the Gateway in order to add or remove sites.

Optionally, an administrator can enable a possibility for dynamic site registration at runtime, see *Dynamic registration of Vsites* for details. Then this file should contain only the static entries (or none if all sites register dynamically).

Further options for back-end sites configuration are presented in *Using the Gateway for failover and/or loadbalancing of UNICORE sites*.

5.1.1.3.3 Main server settings: `gateway.properties`

Use the `gateway.hostname` property to configure the network interface and port the Gateway will listen on. You can also select between `https` and `http` protocol, though in almost all cases https will be used.

Example:

```
gateway.hostname = https://192.168.100.123:8080
```

Note: If you set the host to `0.0.0.0`, the Gateway will listen on all network interfaces of the host machine, else it will listen only on the specified one.

If the scheme of the hostname URL is set to `https`, the Gateway uses the configuration data from `security.properties` to configure the HTTPS settings.

Credential and truststore settings

The Gateway credential and truststore is configured using the following properties

Table 5.1: Credential settings

Property name	Type	Default value / mandatory	Description
gateway.credential.path	filesystem path	*mandatory*	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.
gateway.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).
gateway.credential.password	string		Password required to load the credential.
gateway.credential.keyPath	string		Location of the private key if stored separately from the main credential (applicable for 'pem' and 'der' types only),
gateway.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.
gateway.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'.
gateway.credential.reloadOnChange	[true, false]	true	Monitor credential location and trigger dynamical reload if file changes.

Table 5.2: Truststore settings

Property name	Type	Default value / mandatory	Description
gateway.truststore.allowProxy	[ALLOW, DENY]	ALLOW	Controls whether proxy certificates are supported.
gateway.truststore.type	[keystore, openssl, directory]	*mandatory*	The truststore type.
gateway.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)
gateway.truststore.directoryConnectionTimeout	integer number	15	Connection timeout for fetching the remote CA certificates in seconds.
gateway.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.
gateway.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.
gateway.truststore.directoryLocations.*	list of properties with a common prefix		List of CA certificates locations. Can contain URLs, local files and wildcard expressions.(runtime updateable)
gateway.truststore.keystoreFormat	string		The keystore type (jks, pkcs12) in case of truststore of keystore type.
gateway.truststore.keystorePassword	string		The password of the keystore type truststore.
gateway.truststore.keystorePath	string		The keystore path in case of truststore of keystore type.
gateway.truststore.opensslNewStoreFormat	[true, false]	false	In case of openssl truststore, specifies whether the trust store is in openssl 1.0.0+ format (true) or older openssl 0.x format (false)
gateway.truststore.opensslNsMode	[GLOBUS_EUGRIDPMA, EUGRIDPMA_GLOBUS, GLOBUS, EUGRIDPMA, GLOBUS_EUGRIDPMA_REQUIRE, EUGRIDPMA_GLOBUS_REQUIRE, GLOBUS_REQUIRE, EUGRIDPMA_REQUIRE, EUGRIDPMA_AND_GLOBUS, EUGRIDPMA_AND_GLOBUS_REQUIRE, IGNORE]	EUGRIDPMA_GLOBUS	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).
gateway.truststore.opensslPath	filesystem path	/etc/grid-security/certificates	Directory to be used for opeenssl truststore.
gateway.truststore.crlConnectionTimeout	integer number	15	Connection timeout for fetching the remote CRLs in seconds (not used for Openssl truststores).
gateway.truststore.crlDiskCachePath	filesystem path		Directory where CRLs should be cached, after downloading them from remote source. Can be

Scalability settings

To fine-tune the operational parameters of the embedded Jetty server, you can set advanced HTTP server parameters (see *HTTP server settings* for details). Among others you can use the non-blocking IO connector offered by Jetty, which will scale up to higher numbers of concurrent connections than the default connector.

The Gateway acts as a https client for the VSites behind it. The number of concurrent calls is limited, and controlled by two parameters:

```
# maximum total number of concurrent calls to Vsites
gateway.client.maxTotal=100
# total number of concurrent calls per site
gateway.client.maxPerService=20
```

You can also control the limit on the maximum SOAP header size which is allowed by the Gateway. Typically you **don't have to touch this parameter**. However, if your clients do produce very big SOAP headers and the Gateway blocks them, you can increase the limit. Note that such a giant SOAP header usually means that the client is not behaving as intended, e.g. is trying to perform a DoS attack.

```
# maximum size of an accepted SOAP header, in bytes
gateway.soapMaxHeader=102400
```

Note: The Gateway may consume this amount of memory (plus some extra amount for other data) for each opened connection. Therefore, this value multiplied by the number of maximum allowed connections, should be **significantly lower**, then the total memory available for the Gateway.

Dynamic registration of Vsites

Dynamic registration is controlled by three properties in `CONF/gateway.properties` file:

```
gateway.registration.enable=true
gateway.registration.secret=<your secret>
```

If set to `true`, the Gateway will accept dynamic registrations which are made by sending a `HTTP POST` request to the URL `/VSITE_REGISTRATION_REQUEST`. This request must contain a parameter `secret` which matches the value configured in the `gateway.properties` file.

Filters can be set to forbid access of certain hosts, or to require certain strings in the Vsite addresses. For example,

```
gateway.registration.deny=foo.org example.org
```

will deny registration if the remote hostname contains *foo.org* or *example.org*. Conversely,

```
gateway.registration.allow=mydomain.org
```

will only accept registrations if the remote address contains *mydomain.org*. These two (deny and allow) can be combined.

Web interface (*monkey page*)

For testing and simple monitoring purposes, the Gateway displays a website showing detailed site information (the details view can be disabled). Once the Gateway is running, open up a browser and navigate to `https://<gateway_host>:8080` (or whichever URL the gateway is running on). If the Gateway is configured to do SSL authentication, you will need to import a suitable client certificate into your web browser.

A HTML form for testing the dynamic registration is available as well, by clicking the link in the footer of the main Gateway page.

To disable the Vsite details page, set

```
gateway.disableWebpage=true
```

Main options reference

Property name	Type	Default value / mandatory	Description
gateway.acme.enable	[true, false]	false	Enable ACME / Let's Encrypt support. Will add a HTTP listener on the port defined the acme.httpPort property.
gateway.acme.httpPort	integer >= 1	80	Port for the plain HTTP listener.
gateway.acme.tokenDirectory	string	./acme	Directory from which to serve Let's Encrypt / ACME tokens.
gateway.hostname	string	*mandatory*	external gateway bind address
gateway.registration.allow	string		Space separated list of allowed hosts for dynamic registration.
gateway.registration.deny	string		Space separated list of denied hosts for dynamic registration.
gateway.registration.enable	[true, false]	false	Whether dynamic registration of sites is enabled.
gateway.registration.secret	string		Required secret for dynamic registration.
gateway.soapMaxHeader	string		DEPRECATED, no effect
gateway.consignorTokenTimeTolerance	integer >= 0	30	The validity time of the authenticated client information passed to backend sites will start that many seconds before the real authentication. It is used to mask time synchronization problems between machines.
gateway.consignorTokenValidity	integer >= 1	60	What is the validity time of the authenticated client information passed to backend sites. Increase it if there machines clocks are not synhronized.
gateway.signConsignorToken	[true, false]	false	Controls whether information about the authenticated client (the consignor) passed to backend sites should be signed, or not. Signing is slower, but is required when sites may be reached directly, bypassing the Gateway.
gateway.client.chunked	[true, false]	true	Controls whether chunked passing of HTTP requests to backend sites is supported.
gateway.client.connectionTimeout	integer number	30000	Connection timeout, used when connecting to backend sites.
gateway.client.expectContinue	[true, false]	true	Controls whether the HTTP expect-continue mechanism is enabled on connections to backend sites.
gateway.client.gzip	[true, false]	true	Controls whether support for compression is announced to backend sites.
gateway.client.keepAlive	[true, false]	true	Whether to keep alive the connections to backend sites.
gateway.client.maxPerService	integer number	20	Maximum allowed number of connections per backend site.
gateway.client.maxTotal	integer number	100	Maximum total number of connections to backend sites allowed.
gateway.client.socketTimeout	integer number	30000	Connection timeout, used when connecting to backend sites.
gateway.disableWebpage	[true, false]	false	Whether the (so called monkey) status web page should be disabled.
gateway.externalHostname	string	null	External address of the gateway, when it is accessible through a frontend server as Apache HTTP

HTTP server settings

Property name	Type	Default value / mandatory	Description
gateway.httpServer.CORS_allowedHeaders	string	*	CORS: comma separated list of allowed HTTP headers (default: any)
gateway.httpServer.CORS_allowedMethods	string	GET,PUT,POST,DELETE,HEAD	CORS: comma separated list of allowed HTTP verbs.
gateway.httpServer.CORS_allowedOrigins	string	*	CORS: allowed script origins.
gateway.httpServer.CORS_chainPreflight	[true, false]	false	CORS: whether preflight OPTION requests are chained (passed on) to the resource or handled via the CORS filter.
gateway.httpServer.CORS_exposedHeaders	string	Location,Content-Type	CORS: comma separated list of HTTP headers that are allowed to be exposed to the client.
gateway.httpServer.disabledCipherSuites	string	*empty 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
gateway.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.
gateway.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.
gateway.httpServer.enableSNI	[true, false]	false	Enable Server Name Indication (SNI)
gateway.httpServer.fastRandom	[true, false]	false	Use insecure, but fast pseudo random generator to generate session ids instead of secure generator for SSL sockets.
gateway.httpServer.gzip.enable	[true, false]	false	Controls whether to enable compression of HTTP responses.
gateway.httpServer.gzip.minGzipSize	integer number	100000	Specifies the minimal size of message that should be compressed.
gateway.httpServer.highLoadConnections	string		DEPRECATED, no effect
gateway.httpServer.lowResourceMaxIdleTime	string		DEPRECATED, no effect
gateway.httpServer.maxConnections	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.
gateway.httpServer.maxIdleTime	integer >= 1	200000	Time (in ms.) before an idle connection will time out. It should be large enough not to expire connections with slow clients, values below 30s are getting quite risky.
gateway.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.
gateway.httpServer.minThreads	integer >= 1	1	Minimum number of threads to have in the thread pool for processing HTTP connections. Note that

5.1.1.3.4 Require end-user certificates

Using client certificates for end-user authentication are **not required** or recommended. If you still want to require end-users to have a certificate, the Gateway can be configured accordingly. Set the following in `gateway.properties`:

```
gateway.httpServer.requireClientAuthn=true
```

Logging

UNICORE uses Log4j (version 2) as its logging framework, and comes with an example configuration file (`CONF/logging.properties`).

Please refer to the Log4j documentation for more information.

The most important, root log categories used by the Gateway's logging are:

unicore.gateway	General Gateway logging
unicore.security	Certificate details and other security
org.apache.http	Outgoing HTTP to the backend services

5.1.1.4 Using Apache httpd as a frontend

You may wish to use the Apache webserver (httpd) as a frontent for the Gateway (e.g. for security or fault-tolerance reasons).

5.1.1.4.1 Requirements

- Apache httpd
- mod_proxy for Apache httpd

5.1.1.4.2 External references

- https://wiki.eclipse.org/Jetty/Howto/Configure_mod_proxy

5.1.1.5 Using the Gateway for failover and/or loadbalancing of UNICORE sites

The Gateway can be used as a simple failover solution and/or loadbalancer to achieve high availability and/or higher scalability of UNICORE/X sites without additional tools.

A site definition (in `CONF/connections.properties`) can be extended, so that multiple physical servers are used for a single virtual site.

An example for such a so-called multi-site declaration in the `connections.properties` file looks as follows:

```
#declare a multisite with two physical servers

MYSITE=multisite:vsites=https://localhost:7788 https://localhost:7789
```

This will tell the Gateway that the virtual site *MYSITE* is indeed a multi-site with the two given physical sites.

5.1.1.5.1 Configuration

Configuration options for the multi-site can be passed in two ways. On the one hand they can go into the `connections.properties` file, by putting them in the multi-site definition, separated by `;` characters:

```
#declare a multisite with parameters

MYSITE=multisite:param1=value1;param2=value2;param3=value3;...
```

The following general parameters exist:

vsites	List of physical sites
strategy	Class name of the site selection strategy to use (see below)
config	Name of a file containing additional parameters

Using the `config` option, all the parameters can be placed in a separate file for enhanced readability. For example, you could define in `connections.properties`:

```
#declare a multisite with parameters read from a separate file

MYSITE=multisite:config=conf/mysite-cluster.properties
```

and give the details in the file `conf/mysite-cluster.properties`:

```
#example multisite configuration
vsites=https://localhost:7788 https://localhost:7789

#check site health at most every 5 seconds
strategy.healthcheck.interval=5000
```

5.1.1.5.2 Available strategies

A selection strategy is used to decide where a client request will be routed. By default, the strategy is "**Primary with fallback**", i.e. the request will go to the first site if it is available, otherwise it will go to the second site.

Primary with fallback

This strategy is suitable for a high-availability scenario, where a secondary site takes over the work in case the primary one goes down for maintenance or due to a problem. This is the default strategy, so nothing needs to be configured to enable it. If you want to explicitely enable it anyway, set

```
strategy=primaryWithFallback
```

The strategy will select from the first two defined physical sites. The first, primary one will be used if it is available, else the second one. Health check is done on each request, but not more frequently as specified by the `strategy.healthcheck.interval` parameter. By default, this parameter is set to `5000` milliseconds.

Changes to the site health will be logged at `INFO` level, so you can see when the sites go up or down.

Round robin

This strategy is suitable for a load-balancing scenario, where a random site will be chosen from the available ones. To enable it, set

```
strategy=roundRobin
```

Changes to the site health will be logged at `INFO` level, so you can see when the sites go up or down.

It is very important to be aware that this strategy requires that all backend sites used in the pool, share a common persistence. It is because Gateway does not track clients, so particular client requests may land at different sites. This is typically solved by using a non-default, shared database for sites, such as MySQL.

Caution: Currently loadbalancing of target sites is an experimental feature and is not yet fully functional. It will be improved in future UNICORE versions.

Custom strategy

You can implement and use your own failover strategy, in this case, use the name of the Java class as strategy name:

```
strategy=your_class_name
```

5.1.1.6 Gateway failover and migration

The *Using the Gateway for failover and/or loadbalancing of UNICORE sites* covered usage of the Gateway to provide failover of backend services. However, it may be needed to guarantee high-availabilty for the Gateway itself or to move it to other machine in case of the original one's failure.

5.1.1.6.1 Gateway's migration

The Gateway does not store any state information, therefore its migration is easy. It is enough to install the Gateway at the target machine (or even to simply copy it in the case of installation from the core server bundle) and to make sure that the original Gateway's configuration is preserved.

If the new machine uses a different address, it needs to be reflected in the server's configuration file (the listen address). Also, the configuration of sites behind the Gateway must be updated accordingly.

5.1.1.6.2 Failover and loadbalancing of the Gateway

Gateway itself doesn't provide any features related to its own redundancy. However, as it is stateless, the standard redundancy solutions can be used.

The simpliest solution is to use Round Robin DNS, where DNS server routes the Gateway's DNS address to a pool of real IP addresses. While easy to set up this solution has a significant drawback: DNS server doesn't care about machines being down.

5.2 UNICORE/X

UNICORE/X is the central component of a typical UNICORE installation, providing *REST APIs* for job management and data access services for a single compute cluster (or just a file system).

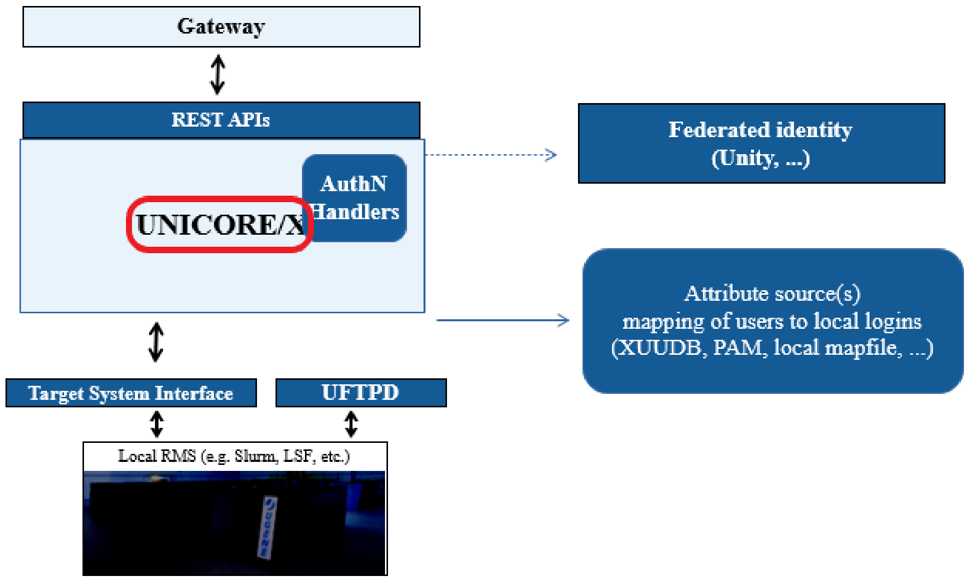

Gateway
REST APIs
AuthN Handlers
UNICORE/X
Federated identity (Unity, ...)
Attribute source(s) mapping of users to local logins (XUUDB, PAM, local mapfile, ...)
Target System Interface
UFTPD
Local RMS (e.g. Slurm, LSF, etc.)

Fig. 5.3: UNICORE/X Server

UNICORE/X Manual Installation and Operating the UNICORE/X server.

UNICORE/X Update Update a UNICORE/X server to this version.

5.2.1 UNICORE/X Manual

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 https://www.unicore.eu.

5.2.1.1 Getting started

5.2.1.1.1 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 https://openjdk.java.net/install/.

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.

5.2.1.1.2 Installation

UNICORE/X can be installed either as a part of the UNICORE Server bundle (*tar.gz* or *zip* archive) or from a Linux package (i.e. *RPM* or *deb*) on the UNICORE project website at sourceforge.

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 *Configuration of UNICORE/X*.

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:

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

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
```

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 *Logging*.

5.2.1.2 Configuration of UNICORE/X

5.2.1.2.1 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.

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, please refer to the *Attribute sources*.
- 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 see *Interfacing UNICORE/X to the TSI*.
- 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 *Features provided by UNICORE/X*.

UNICORE/X is configured using several config files residing in the *CONF* directory, please see *Getting started* for the location of the *CONF* directory.

5.2.1.2.2 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 5.3: 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
main.config	Main server config file. Defines features, storages, AuthN/AuthZ, AIPs/PDPs	no
container.properties	Server address, SSL settings, Web server settings	no
tsi.config	Configuration to access the 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
saml.config	Configure the use of Unity as an attribute source (optional)	no

5.2.1.2.3 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 *Administration* for more on these.

5.2.1.2.4 Config file formats

UNICORE/X uses two different formats for configuration.

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.

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.

Hint: It is advisable to run a tool such as `xmllint` after editing XML files to check for typos.

5.2.1.2.5 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 *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 *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 *can have subkeys*		Properties with this prefix are used to configure container's Jetty HTTP server. See separate documentation for details.
container.messageLogging..*	[true, false] *can have subkeys*	false	Append service name and set to 'true' to enable message logging for that service.

continues on next page

Table 5.4 – continued from previous page

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.onstartupSelftest	[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.pools.executor.idletime	integer number	60000	The timeout in millis for removing idle threads.
container.pools.executor.maxsize	integer number	16	The maximum thread pool size for the internal execution service
container.pools.executor.minsize	integer >= 1	2	The minimum thread pool size for the internal execution service
container.pools.scheduled.idletime	integer number	60000	Timeout in millis for removing idle threads.
container.pools.scheduled.size	integer >= 1	3	Defines the thread pool size for the execution of scheduled services.
container.port	integer [0 – 65535]	7777	Server listen port.
container.runtimeConfigurationUpdate	[true, false]	true	Whether the server refreshes its configuration at runtime whenever the main config file changes.
container.security..*	string *can have subkeys*		Properties with this prefix are used to configure container's security. See separate documentation for details.
container.services.expirycheck.initial.*	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.services.expirycheck.period.*	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.services.instanceLockingTimeout.*	integer number *can have subkeys*	30	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.services.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.

continues on next page

Table 5.4 – continued from previous page

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

5.2.1.2.6 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*.

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.

Registry

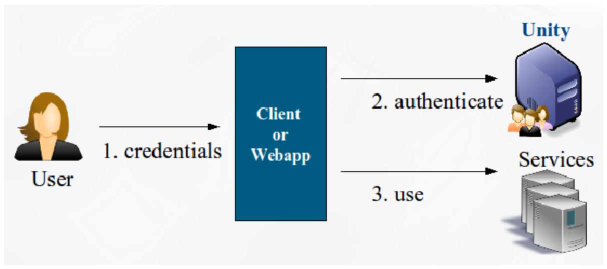
It is possible to configure UNICORE/X to contact one or more external or global *UNICORE Registries* in order to publish information on crucial services there.

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
```

Unity

If you want to support user authentication via Unity, and add an extra level of security by validating the replies from 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 *Credential and truststore settings*.

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*.

5.2.1.2.7 Security

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).
- Authentication options for selecting what kind of credentials users can use to identify themselves to the UNICORE/X server. A number of authentication options exist, from various forms of username/password authentication to OIDC tokens. Even X.509 certificates and SSH keys are supported. If multiple options are configured, the first successful authentication will be used. The description of the configuration options can be found in *Authentication*
- 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 *Attribute sources*.

In very rare cases, you might want to change the

- Access control setup (controlling in detail who can do what on which services). Again, several options exist, which are described in *Authorization back-end (PDP) guide*.

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		Controls which Policy Decision Point (PDP, the authorisation engine) should be used. Default value is determined as follows: if eu.unicore.uas.pdp.local.LocalHerasafPDP is available then it is used. If not then this option becomes mandatory.
container.security.accesscontrol.pdpConfig	filesystem path		Path of the PDP configuration file
container.security.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.combiningPolicy	string	MERGE_LAST_OVERRIDES	What algorithm should be used for combining the attributes from multiple attribute sources (if more then one is defined).
container.security.attributes.disableRuntimeUpdates	[true, false]	false	Whether to not allow runtime updates of the attribute sources.
container.security.attributes.order	string		Attribute sources in invocation order.
container.security.credential..*	string *can have subkeys*		Properties with this prefix are used to configure the credential used by the container. See separate documentation for details.
container.security.defaultVOs.<NUMBER>	list of properties with a common prefix	*empty string*	List of default VOs, which should be assigned for a request without a VO set. The first VO on the list where the user is member will be used.
container.security.dynamicAttributes.*	string *can have subkeys*		Prefix used for configurations of particular dynamic attribute sources.
container.security.dynamicAttributes.combiningPolicy	string	MERGE_LAST_OVERRIDES	What algorithm should be used for combining the attributes from multiple dynamic attribute sources (if more then one is defined).
container.security.dynamicAttributes.disableRuntimeUpdates	[true, false]	false	Whether to not allow runtime updates of the dynamic attribute sources.
container.security.dynamicAttributes.order	string		Dynamic attribute sources in invocation order.
container.security.gateway.certificate	filesystem path		Path to gateway's certificate file in PEM or DER format. Note that DER format is used only for files with '.der' extension. It is used only for gateway's authentication assertions verification (if enabled). Note that this is not needed to set it if waiting for gateway on startup is turned on.
container.security.gateway.enable	[true, false]	true	Whether to gateway-related features are enabled. Note that if it is enabled either the UNICORE/X server must be secured (usually via firewall) to disable non-gateway access.
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.
con-	string		Required secret when autoregistering with

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*	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'.
container.security.credential.reloadOnChange	[true, false]	true	Monitor credential location and trigger dynamical reload if file changes.

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*	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)
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)
container.security.truststore.keystoreFormat	string		The keystore type (jks, pkcs12) in case of truststore of keystore type.
container.security.truststore.keystorePassword	string		The password of the keystore type truststore.
container.security.truststore.keystorePath	string		The keystore path in case of truststore of keystore type.
container.security.truststore.opensslNewStoreFormat	[true, false]	false	In case of openssl truststore, specifies whether the trust store is in openssl 1.0.0+ format (true) or older openssl 0.x format (false)
container.security.truststore.opensslNsMode	[GLOBUS_EUGRIDPMA, EUGRIDPMA_GLOBUS, GLOBUS, EUGRIDPMA, GLOBUS_EUGRIDPMA_REQUIRE, EUGRIDPMA_GLOBUS_REQUIRE, GLOBUS_REQUIRE, EUGRIDPMA_REQUIRE, EUGRIDPMA_AND_GLOBUS, EUGRIDPMA_AND_GLOBUS_REQUIRE, IGNORE]	EUGRIDPMA_GLOBUS	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.opensslPath	filesystem path	/etc/grid-security/certificates	Directory to be used for opeenssl truststore.
container.security.truststore.crlConnectionTimeout	integer number	15	Connection timeout for fetching the remote CRLs in seconds (not used for Openssl truststores).
con-	filesystem path		Directory where CRLs should be cached, af-

5.2.1.2.8 Configuring the execution backend (XNJS and TSI)

Information on the configuration of the XNJS and TSI backend can be found in *Interfacing UNICORE/X to the TSI*.

5.2.1.2.9 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 *Configuration of storages*.

5.2.1.2.10 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.
container.httpServer.CORS_allowedOrigins	string	*	CORS: allowed script origins.
container.httpServer.CORS_chainPreflight	[true, false]	false	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	*empty 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.
container.httpServer.enableSNI	[true, false]	false	Enable Server Name Indication (SNI)
container.httpServer.fastRandom	[true, false]	false	Use insecure, but fast pseudo random generator to generate session ids instead of secure generator for SSL sockets.
container.httpServer.gzip.enable	[true, false]	false	Controls whether to enable compression of HTTP responses.
container.httpServer.gzip.minGzipSize	integer number	100000	Specifies the minimal size of message that should be compressed.
container.httpServer.maxConnections	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.maxIdleTime	integer >= 1	200000	Time (in ms.) before an idle connection will time out. It should be large enough not to expire connections with slow clients, values below 30s are getting quite risky.
container.httpServer.maxThreads	integer 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.
con-	[true, false]	true	Controls whether the SSL socket requires

The HTTP client options are the following:

Property name	Type	Default value / mandatory	Description
container.httpServer.digitalSigningEnabled	[true, false]	true	Controls whether signing of key web service requests should be performed.
container.httpServer.httpAuthnEnabled	[true, false]	false	Whether HTTP basic authentication should be used.
container.httpServer.httpPassword	string	*empty string*	Password for use with HTTP basic authentication (if enabled).
container.httpServer.httpUser	string	*empty string*	Username for use with HTTP basic authentication (if enabled).
container.httpServer.maxWsCallRetries	integer number	3	Controls how many times the client should try to call a failing web service. Note that only the transient failure reasons cause the retry. Note that value of 0 enables unlimited number of retries, while value of 1 means that only one call is tried.
container.httpServer.messageLogging	[true, false]	false	Controls whether messages should be logged (at INFO level).
container.httpServer.securitySessions	[true, false]	true	Controls whether security sessions should be enabled.
container.httpServer.serverHostnameChecking	[NONE, WARN, FAIL]	WARN	Controls whether server's hostname should be checked for matching its certificate subject. This verification prevents man-in-the-middle attacks. If enabled WARN will only print warning in log, FAIL will close the connection.
container.httpServer.sslAuthnEnabled	[true, false]	true	Controls whether SSL authentication of the client should be performed.
container.httpServer.wsCallRetryDelay	integer number	10000	Amount of milliseconds to wait before retry of a failed web service call.
container.httpServer.http.allow-chunking	[true, false]	true	If set to false, then the client will not use HTTP 1.1 data chunking.
container.httpServer.http.connection-close	[true, false]	false	If set to true then the client will send connection close header, so the server will close the socket.
container.httpServer.http.connection.timeout	integer number	20000	Timeout for the connection establishing (ms)
container.httpServer.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.httpServer.http.maxRedirects	integer number	3	Maximum number of allowed HTTP redirects.
container.httpServer.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.httpServer.http.socket.timeout	integer number	0	Socket timeout (ms)
container.httpServer.http.nonProxyHosts	string		Space (single) separated list of hosts, for which the HTTP proxy should not be used.
container.httpServer.http.proxy.password	string		Relevant only when using HTTP proxy: defines password for authentication to the proxy.
container.httpServer.http.proxy.user	string		Relevant only when using HTTP proxy: defines username for authentication to the proxy.
container.httpServer.http.proxyHost	string		If set then the HTTP proxy will be used, with this hostname.
container.httpServer.http.proxyPort	integer number		HTTP proxy port. If not defined then system property is consulted, and as a final fallback

5.2.1.2.11 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.

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
```

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

StorageAccess

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

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
```

Workflow

This feature provides workflow processing involving only this UNICORE/X server, (i.e. NO cross-site workflows)

To disable the whole feature

```
container.feature.WorkflowEngine.enable=false
```

Base

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

Service name	Usage
core	RESTful APIs for jobs and data
Task	Service for async tasks (like metadata extraction)

Admin

This feature provides the Admin service (see *The Admin web service*).

Service name	Usage
admin	RESTful API to the admin service

5.2.1.3 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.

Service name	Usage
registries	RESTful API to the Registry service
Registry	Registry service
ServiceGroupEntry	Registry entries service

5.2.1.4 Administration

5.2.1.4.1 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, please see *Features provided by UNICORE/X* 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).

5.2.1.4.2 Logging

UNICORE uses the Log4j 2 logging framework. The config file is specified with a Java property `log4j.configurationFile`.

Hint: 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.

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)

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

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.

5.2.1.4.3 Administration and monitoring

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

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
```

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.

5.2.1.4.4 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.

5.2.1.5 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.
- a security session is established (if sessions are enabled), and the client can simply send the security session ID on subsequent requests to avoid having to go through the full authentication process again.
- 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.

5.2.1.5.1 Security concepts

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 *Configuration of UNICORE/X*).

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.

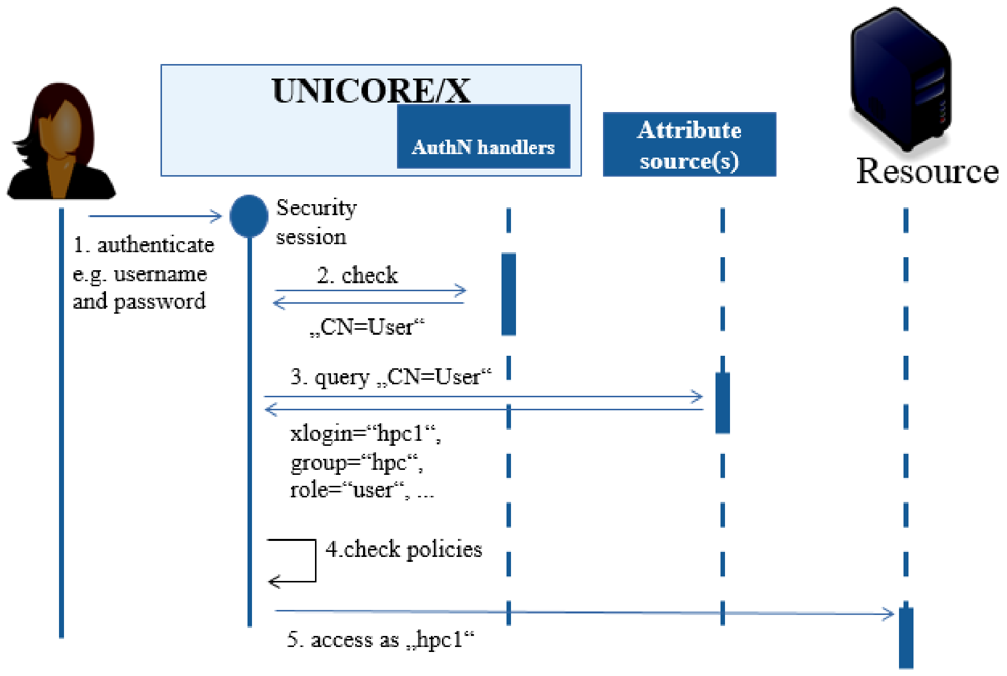


Fig. 5.4: UNICORE Authentication and Authorization

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.

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.

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.

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 *Attribute sources* for more information.

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 *Authorization back-end (PDP) guide* describes how to configure different engines for policy evaluation including a remote one.

Authorisation

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

Security sessions

If enabled (which is the default), the server generates a security session after successful authentication. The session ID is sent back to the client (via HTTP header `X-UNICORE-SecuritySession` in the response), allowing the client to authenticate subsequent requests using this session ID (using the same HTTP header). This will speed-up the client-server communication, especially in cases where external authentication (e.g. via Unity is used). These sessions have a limited lifetime (8 hours by default).

Note: The security session only covers authentication attribute assignment and authorization is always done for each request.

For details on how to configure this feature, see the general properties overview in section *Configuration of UNICORE/X*.

5.2.1.6 Authentication

5.2.1.6.1 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).

5.2.1.6.2 Username-password file

The FILE mechanism uses a local 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.

5.2.1.6.3 Unity authentication using OAuth Bearer token

This mechanism uses the OAuth token sent from the client (HTTP `Authorization: Bearer ...` header) to authenticate to Unity. In Unity terms, this uses the endpoint of type `SAMLSoapIdP` (or `SAMLUnicoreSoapIdP`) 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
```

You can configure an additional validation of the Unity assertions using the configured *trusted assertion issuer* certificate(s):

```
# validate the received assertions?
container.security.rest.authentication.UNITY-OAUTH.validate=true
```

For this to work, UNICORE needs to public key of the Unity server as one of the *trusted assertion issuers*, please refer to the relevant section on trusted assertion issuers in the manual.

5.2.1.6.4 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
```

You can configure an additional validation of the Unity assertions using the configured *trusted assertion issuer* certificate(s):

```
# validate the received assertions?
container.security.rest.authentication.UNITY.validate=true
```

For this to work, UNICORE needs to public key of the Unity server as one of the *trusted assertion issuers*, please refer to the relevant section on trusted assertion issuers in the manual.

5.2.1.6.5 OAuth token authentication with an OIDC server

This mechanism checks the OAuth token issued by an OIDC server such as Keycloak directly with the issuing server.

```
container.security.rest.authentication.OAUTH.class=eu.unicore.services.rest.security.
↪OAuthAuthenticator
container.security.rest.authentication.OAUTH.address=https://your.server/auth/realms/
↪your_realm/protocol/openid-connect/userinfo
```

UNICORE will use the user's OAuth token to make a call to the `userinfo` endpoint, effectively checking if that token is (still) valid.

You can alternatively use the `introspect` endpoint, where UNICORE acts as an OAuth client with client ID and secret to check the token's validity and get user info. In this case you need to set `validate=true` and provide client ID and secret

```
container.security.rest.authentication.OAUTH.address=https://your.server/auth/realms/
↪your_realm/protocol/openid-connect/token/introspect
container.security.rest.authentication.OAUTH.validate=true
container.security.rest.authentication.OAUTH.clientID=your-client-id
container.security.rest.authentication.OAUTH.clientSecret=your-client-secret
```

The parameter `dnTemplate` is used to define the DN that will be assigned to authenticated users, where the `%param` parameters will be replaced by the corresponding parameters from the (validated OIDC token) The default template is `UID=%email`.

```
container.security.rest.authentication.OAUTH.dnTemplate=UID=%email
```

Successful OAuth authentication can assign the role and set the username based on the reply from the OIDC server. To enable this, you need to define templates for setting the role and/or the user ID. For example, to assign role "user" to each successfully authenticated request, and to use the "login_name" attribute in the token as the UNIX uid,

```
container.security.rest.authentication.OAUTH.roleTemplate=user
container.security.rest.authentication.OAUTH.uidTemplate=%login_name
```

This can be overriden later by the configured *attribute sources*.

5.2.1.6.6 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
```

5.2.1.6.7 PAM

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

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

container.security.rest.authentication.PAM.class=eu.unicore.services.rest.security.
↪PAMAuthenticator
container.security.rest.authentication.PAM.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*".

A successful PAM authentication will also assign a "user" role, and will set the username as the UNIX login, which can be overriden later by the configured *attribute sources*.

5.2.1.6.8 Customizing JWT Delegation

UNICORE has a delegation mechanism 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.

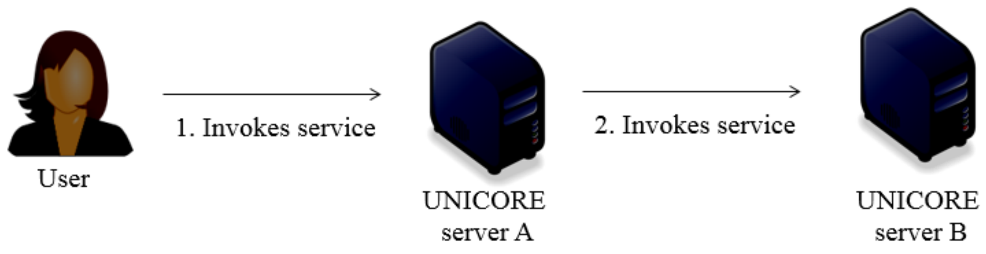


Fig. 5.5: UNICORE Delegation

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

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

The public keys that servers use to verify the JWT signatures are by default distributed via the shared service Registry.

This works out of the box, and does not require any configuration.

However, if required, you can load additional public keys for trusted services from local PEM files using the following:

```
container.security.rest.jwt.trustedLocalIssuer.1=<path_to_local_PEM_file>
container.security.rest.jwt.trustedLocalIssuer.2=...
```

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.

Note that a server with HMAC secret defined will still trust certificate-based JWT tokens, but will always use HMAC to sign its own delegation tokens.

5.2.1.7 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 **A**ttribute **I**nformation **P**oint, 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.

5.2.1.7.1 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.

5.2.1.7.2 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.

The **full list** of options related to attribute sources is available *here*.

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=...
...
```

(continues on next page)

(continued from previous page)

```
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).

5.2.1.7.3 Available attribute sources

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 **G**rid **C**omponent **ID** (GCID). This makes it easy to assign users different attributes when accessing different UNICORE/X servers.

Full XUUDB documentation is available from *XUUDB Manual*.

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>
```

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 *Virtual Organisations (VO) Support*.

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.

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
```

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

```

(continues on next page)

(continued from previous page)

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

5.2.1.8 Virtual Organisations (VO) Support

VO (**V**irtual **O**rganisation) 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.

5.2.1.8.1 Overview

Features

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

- Unity can provide all user attributes to be 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 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.

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. The 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 options* for the syntax.

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 *Guide to XACML security policies* for details.

Supported VO (SAML) 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!**

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.

5.2.1.8.2 Configuration

This sections describes the default configuration file format which is used to configure the SAML attribute source and provides detailed and comprehensive information on all configuration options. In most cases, the defaults are fine - you can refer to the *HOWTO* 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 `saml.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*.

To enable the VO subsystem certain settings are also required in the main server's configuration file. You have to enable the SAML 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 XUUDB

container.security.attributes.SAML.class=eu.unicore.uas.security.saml.SAMLAttributeSource
container.security.attributes.SAML.configurationFile=conf/saml.config

# ... xuudb configuration omitted ...
```

Before proceeding to fill the SAML/VO configuration it is suggested to prepare a truststore, which should contain **ONLY** the certificates of the trusted SAML 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 the standard UNICORE logging configuration file. See *Logging configuration* section for possible settings related to the SAML subsystem.

Main SAML (VO) configuration file

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

Property name	Type	Default value / mandatory	Description
saml.attributeQuery.password	string		If certificate-based authentication to the SAML server is disabled, you might be able to use username/password. This sets the password.
saml.attributeQuery.username	string		If certificate-based authentication to the SAML server is disabled, you might be able to use username/password. This sets the username.
saml.attributeQueryURL	string	localhost	Full address (URL) of SAML Attribute Query service.
saml.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.
saml.enableGenericAttributes	[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.
saml.group	string		Group which is accepted by this attribute source. UNICORE/X will honor only attributes with exactly this scope or global (i.e. without scope set)
saml.localServerURI	string		Can contain a local server SAML identifier to be used in SAML requests. If unset, then the server's X.500 DN is used.
saml.truststore..*	string *can have subkeys*		Properties starting with this prefix are used to configure validation of SAML assertion issuers certificates. Trust anchors should contain only the trusted SAML servers certificates. All options are the same as those for other UNICORE truststores.
saml.unicoreAttribute..*	string *can have subkeys*		Properties starting with this prefix are used to configure mappings of SAML attributes to UNICORE internal ones.
saml.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 SAML truststore.

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 *UNICORE incarnation and authorization attributes*).

Property name	Range of values	Description
saml.unicoreAttribute.NAME	URI	Value must be a SAML attribute name which will be used as a UNICORE internal incarnation attribute NAME.
saml.unicoreAttribute.NAME.default	URI	Value must be a SAML attribute name which will be used as a default for UNICORE internal incarnation attribute NAME.
saml.unicoreAttribute.NAME.disabled	ANY, IGNORED	When this attribute is present regardless of its value the NAME attribute won't be mapped.

Example mapping for Unity attributes

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

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

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

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

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

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

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

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

Logging configuration

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

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

```
logger.saml.name=unicore.security.saml
logger.saml.level=debug
```

5.2.1.8.3 VO (SAML) configuration HOWTOs

SAML and UNICORE - basic case

This section shows all the steps which are required to setup a UNICORE/X server and Unity to work as SAML attribute source. In this scenario we will use Unity to centrally store mappings of user DNs to UNIX logins (Xlogins) and roles of of those 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 the SAML attribute source 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 XUUDB
container.security.attributes.combiningPolicy=MERGE_LAST_OVERRIDES
# ...  xuudb configuration omitted

container.security.attributes.SAML.class=eu.unicore.uas.security.saml.
↪SAMLAttributeSource
```

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

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

saml.verifySignatures=true
saml.truststore.type=directory
saml.truststore.directoryLocations.1=/opt/unicore/certs/unity/*.pem

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

saml.cacheTtl=20

```

(continues on next page)

(continued from previous page)

```
saml.attributeQueryURL=https://unity.example.org/unicore-soapidp/saml2unicoreidp-
↪soap/AssertionQueryService
#saml.attributeQuery.username=UX-VENUS
#saml.attributeQuery.password=the!njs!!

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

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

(Very) 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 the `saml.config` file:

```
saml.enableGenericAttributes=true
```

Then you can modify the XACML policy to require certain VO attributes.

Important fact to note here 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>
```

(continues on next page)

(continued from previous page)

```
                  <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>
```

5.2.1.9 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, PostgreSQL, 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 or PostgreSQL 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.

5.2.1.9.1 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.maxSize.*	integer number *can have subkeys*	10	Maximum number of elements in the cache (default: 10).
persistence.class.*	string *can have subkeys*	eu.unicore.persist.impl.H2Persist	The persistence implementation class, which controls with DB backend is used.
persistence.cluster.config.*	string *can have subkeys*		Clustering configuration file.
persistence.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_mode.*	[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.tabletype.*	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.pgsql.useSSL.*	[true, false] *can have subkeys*	true	(PostgreSQL) Connect using SSL.
persistence.pool_timeout.*	integer number *can have subkeys*	3600	Connection pool timeout when trying to get a connection.
persistence.port.*	integer number *can have subkeys*		The database port. If not set, the default port for the chosen DB backend is used.
persistence.user.*	string *can have subkeys*	sa	The database username.

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
```

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>
```

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=3306
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.

The PostgreSQL engine

To configure PostgreSQL for UNICORE persistence data, you need to set

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

You will need access to a PostgreSQL server, and configure access to a database. We recommend a dedicated PostgreSQL user. Here is an example for how to setup access.

- create a dedicated PostgreSQL user *unicore* (and set a password)

  ```
  $ sudo -u postgres createuser -P unicore
  ```

- create a database for holding UNICORE data

  ```
  $ sudo -u postgres createdb -O unicore unicore_data
  ```

- Check that the PostgreSQL server allows for password authentication for the UNICORE user. Ensure that in `pg_hba.conf` you have lines similar to these:

  ```
  host    all          all          127.0.0.1/32          md5
  host    all          all          ::1/128               md5
  ```

- If UNICORE server(s) and PostgreSQL servers are on different hosts, adapt hostnames accordingly.
- You can check that the connection works by invoking the following on the UNICORE/X server:

  ```
  $ psql -h localhost unicore_data unicore
  ```

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

```
persistence.class=de.fzj.unicore.persist.impl.PGSQLPersist
persistence.database=unicore_data
persistence.user=unicore
persistence.password=some_password
persistence.host=<your_postgresql_host>
persistence.port=5432
```

5.2.1.10 Interfacing UNICORE/X to the TSI

The link from UNICORE/X to the UNICORE TSI, the component that deals with the actual job execution and file system access is configured using a properties file named `tsi.config`. It is included 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.
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.defaultUmask	integer number	0027	Default umask to be used for jobs.
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.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.strictUserInputChecking	[true, false]	false	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 and should usually be left at their default values.

5.2.1.10.1 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(s)
- In the UNICORE TSI configuration, edit the `tsi.properties` file

- On the UNICORE/X server, edit `tsi.config` and `simpleidb` files
- Start the newly installed TSI (as *root* in a multiuser setting)
- Restart UNICORE/X

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 SLURM or Torque.

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 UNICORE 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 that user's id.

As the TSI is a crucial 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 one, denoted `<your_tsi>` in the following.
- Check the TSI configuration, especially command locations, path settings, etc.

Required TSI Configuration

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

```
# UNICORE/X machine
tsi.unicorex_machine=<UNICORE/X host>

# UNICORE/X listener port (check unicorex/conf/tsi.config  variable ``CLASSICTSI.
↪replyport``
tsi.unicorex_port=7654

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

UNICORE/X configuration

Edit `unicorex/conf/main.config` and check that the `tsi.config` file is included:

```
# read TSI-related settings
$include.TSI conf/tsi.config
```

Edit `unicorex/conf/tsi.config`. 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. UNICORE/X 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
```

Communication parameters

Some additional parameters exist for tuning the UNICORE/X-TSI communication.

Table 5.5: UNICORE/X-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

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 UNICORE/X 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.

Hint: When changing TSIs, it's a good idea to remove the UNICORE/X state and any files before restarting. See *The UNICORE persistence layer* for details.

Enabling SSL for the UNICORE/X 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 tsi.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*!

Using an SSH tunnel for the UNICORE/X to TSI communication

In the special case that the callback port on the UNICORE/X server 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
```

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.KILL	string	(too complex to show here)	Unix command template for aborting a process and its child processes.
CLASSICTSI.LN	string	/bin/ln -s	Unix 'ln' command.
CLASSICTSI.MKDIR	string	/bin/mkdir -p	Unix directory creation command.
CLASSICTSI.MV	string	/bin/mv	Unix 'mv' command.
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.interactive_execution.disable	[true, false]	false	Disable execution of user commands on the TSI node.
CLASSICTSI.jobLimit	integer number	-1	Limit number of running jobs (useful with NOBATCH TSI, -1 = no limit)
CLASSICTSI.limitTSIConnections	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.
CLASSICTSI.priveduser	string	unicore	Account used for getting statuses of all batch jobs (cannot be 'root').
CLASSICTSI.replyport	integer >= 1	7654	Reply port on UNICORE/X server.
CLASSICTSI.reservationAdminUser	string	unicore	Account used for making reservations (cannot be 'root'). If null, the current user's login will be used.
CLASSICTSI.reservationEnabled	[true, false]	false	Whether to enable the reservation interface.
CLASSICTSI.socket.connect.timeout	integer >= 0	10	Connection timeout (seconds) on when establishing (or checking) the TSI connection. Set to '0' for no timeout.
CLASSICTSI.socket.no_check_matching_ips	[true, false]	false	Disable checking if IP address(es) of command/data socket callbacks are as expected.

continues on next page

Table 5.6 – continued from previous page

Property name	Type	Default value / mandatory	Description
CLASSICTSI.socket.timeout	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.grace	integer >= 1	2	How many times the XNJS will re-check job status in case of a 'lost' job.
CLASSICTSI.statusupdate.interval	integer >= 1	10000	Interval (ms) for updating job statuses on the batch system.

5.2.1.10.2 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. UNICORE/X 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 the tsi.config 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.

5.2.1.11 The IDB

The UNICORE IDB (**I**ncarnation **D**ata**B**ase) 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.

5.2.1.11.1 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.

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`.

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*, . . .).

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

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
```

5.2.1.11.2 IDB syntax description

Ihe IDB is written in JSON format.

Note: The older XML format was deprecated in UNICORE 8 and removed in UNICORE 9.

The IDB contains Partitions, Applications, Submit/Execute script templates and Info elements, all of which will be described below. Additionally, the administrator can customize the script template that is used to perform special actions, such as loading modules, or changing the shell (please read *Script templates* for more information).

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

```
{
  "Partitions" : {},

  "Info" : {},

  "Applications" : [],

  "ExecuteScriptTemplate" : "...",

  "SubmitScriptTemplate"  : "...",
}
```

Partitions

Each Partition corresponds essentially to a batch queue. Each partition may have its own runtime 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": "4:4",
    "TotalCPUs": "4-256",
    "Runtime": "1-72000:3600",
  },
 },

 "dev" : {
  "Description": "Development queue",
  "OperatingSystem": "LINUX",
  "CPUArchitecture": "x86_64",
```

(continues on next page)

(continued from previous page)

```
  "Resources": {
    "Nodes": "1-4:1",
    "CPUsPerNode": "4:4",
    "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",
```

(continues on next page)

(continued from previous page)

```
 "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
```

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"
```

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", "#COMMAND", "#RESOURCES", "#SCRIPT" ],

"ExecuteScriptTemplate" : [ "#!/bin/sh", "#COMMAND" "#RESOURCES", "#SCRIPT" ],

}
```

You can give these as an array of strings (lines), or as a single string with embedded `\n` line breaks.

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)
- "`#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",
  "#COMMAND",
  "#RESOURCES",
  "LD_LIBRARY_PATH= $LD_LIBRARY_PATH:/opt/openmpi-2.1/lib; export LD_LIBRARY_PATH",
```

(continues on next page)

(continued from previous page)

```
  "PATH=$PATH:/opt/openmpi-2.1/bin; export PATH",
  "#SCRIPT"
],
```

Attention: 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",
"#COMMAND",
"nmodule load java-11 > /dev/null 2>&1",
"#SCRIPT" ]
```

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.

Summary

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>

5.2.1.11.3 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 5.7: 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
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",
 ],
```

(continues on next page)

(continued from previous page)

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

 Prologue: "module load python3",

 Resources: {
   Nodes: 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`.

Arguments

Command line arguments are specified using `Arguments` tags:

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

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

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.

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",
  },
}
```

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",

```

(continues on next page)

(continued from previous page)

```
   PostCommand: "/opt/licenses/release_license"
}
```

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: "",
}
```

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,

}
```

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 flag

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

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

  IgnoreNonZeroExitCode: true,

}
```

5.2.1.11.4 Application argument metadata

For client components it can be useful to have a description of an application in terms of its arguments. This allows clients 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 can be used to limit the range of valid values, depending on the `Type` of the argument. The processing of this attribute is client-dependent.

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.

5.2.1.12 Data staging

When executing user jobs, UNICORE/X 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 and scp.

Some of these have additional configuration options, which are given in this section.

5.2.1.12.1 SCP support

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

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
```

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.

5.2.1.12.2 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 `tsi.config`.

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

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

5.2.1.12.3 Configuration reference

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

Property name	Type	Default value / mandatory	Description
XNJS.staging.addWaitingLoop	[true, false]	false	Whether to add a waiting loop for files to appear on shared filesystems.
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 >= 1	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.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'.

5.2.1.13 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, 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.host	string	null	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	null	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.
coreServices.uftp.command.socketTimeout	integer [0 – 300]	10	The timeout (in seconds) for communicating with the command port.
coreServices.uftp.command.sslDisable	[true, false]	false	Allows to disable SSL on the command port (useful for testing).
coreServices.uftp.disableSessionMode	[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).

5.2.1.13.1 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
```

(continues on next page)

(continued from previous page)

```
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.

5.2.1.14 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 *Metadata service*
- Data-triggered processing is described in *Data-triggered processing*

5.2.1.14.1 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.allowTrigger	[true, false]	true	(if creating via factory) If user is allowed to enable the triggering feature.
coreServices.sms.storage.N.allowUserDefinedPath	[true, false]	true	(if creating via factory) Whether the allow the user to set the storage base directory.
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 eu.unicore.uas.impl.sms.SMSBaseImpl		Storage implementation class 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	027	Default (initial) umask for files in the storage. Must be an octal number.
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.
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.infoProviderClass	Class extending eu.unicore.uas.impl.sms.StorageInfoProvider	eu.unicore.uas.impl.sms.DefaultStorageInfoProvider	(Very) advanced setting, providing 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.
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.workdir	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!

5.2.1.14.2 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.allowTrigger	[true, false]	true	(if creating via factory) If user is allowed to enable the triggering feature.
coreServices.targetsystem.storage.N.allowUserDefinedPath	[true, false]	true	(if creating via factory) Whether the allow the user to set the storage base directory.
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 eu.unicore.uas.impl.sms.SMSBaseImpl		Storage implementation class used (and mandatory) in case of the CUSTOM type.
coreServices.targetsystem.storage.N.cleanup	[true, false]	false	Whether files of the storage should be removed when the storage is destroyed. This is mostly useful for storage factories.(runtime updateable)
coreServices.targetsystem.storage.N.defaultUmask	integer number	027	Default (initial) umask for files in the storage. Must be an octal number.
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.enableTrigger	[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.infoProviderClass	Class extending eu.unicore.uas.impl.sms.StorageInfoProvider	eu.unicore.uas.impl.sms.DefaultStorageInfoProvider	(Very) advanced setting, providing 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)
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
```

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
```

5.2.1.14.3 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.N.allowTrigger	[true, false]	true	(if creating via factory) If user is allowed to enable the triggering feature.
coreServices.sms.factory.N.allowUserDefinedPath	[true, false]	true	(if creating via factory) Whether the allow the user to set the storage base directory.
coreServices.sms.factory.N.checkExistence	[true, false]	true	Whether the existence of the base directory should be checked when creating the storage.
coreServices.sms.factory.N.class	Class extending eu.unicore.uas.impl.sms.SMSBaseImpl		Storage implementation class used (and mandatory) in case of the CUSTOM type.
coreServices.sms.factory.N.cleanup	[true, false]	false	Whether files of the storage should be removed when the storage is destroyed. This is mostly useful for storage factories.(runtime updateable)
coreServices.sms.factory.N.defaultUmask	integer number	027	Default (initial) umask for files in the storage. Must be an octal number.
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.infoProviderClass	Class extending eu.unicore.uas.impl.sms.StorageInfoProvider	eu.unicore.uas.impl.sms.DefaultStorageInfoProvider	(Very) advanced setting, providing 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.
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.workdir	string		(DEPRECATED, use 'path' instead)

For example,

```
coreServices.sms.factory.TYPE1.description=GPFS file system
coreServices.sms.factory.TYPE1.fixedpath=GPFS file system
coreServices.sms.factory.TYPE1.path=/mnt/gpfs/unicore/unicorex-1/storage-factory

# 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
```

5.2.1.14.4 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
```

5.2.1.15 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

5.2.1.15.1 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 *Configuration of storages* 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/
```

5.2.1.15.2 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
```

5.2.1.16 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.

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

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

Since shared storages (HOME, ROOT, etc) 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
```

(continues on next page)

(continued from previous page)

```
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:

```
├── dir1
│   ├── ...
│   └── .UNICORE_Rules
├── dir2
│   ├── ...
│   └── .UNICORE_Rules
├── dir3
│   ├── ...
│   └── .UNICORE_Rules
└── .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.

5.2.1.17 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 **P**olicy **D**ecision **P**oint (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*. You can also review this guide if you want to have a deeper understanding of the authorization process.

5.2.1.17.1 Basic configuration

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

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.

5.2.1.17.2 Available PDP modules

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
```

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. 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.

5.2.1.18 XACML 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 **P**olicy **D**ecision **P**oints (PDP). The *Authorization back-end (PDP) guide* 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.

5.2.1.18.1 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
```

5.2.1.18.2 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" />
```

(continues on next page)

(continued from previous page)

```
                </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.

5.2.1.18.3 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>
```

5.2.1.18.4 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) please read the XACML specification. To get more information on XACML used 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.

5.2.2 UNICORE/X Update

As a first step and precaution, you should make backups of your existing config files and put them in a safe place.

In the following, *LIB* refers to the directory containing the jar files for the component, and *CONF* to the config directory of the existing installation.

- It is assumed that you have unpacked the **tar.gz** file somewhere, e.g. to `/tmp/`. In the following, this location will be denoted as "*$NEW*":

```
$ export NEW=/tmp/unicore-servers-9.0.0
```

- Stop the server. If not yet done, make a backup of the config files.
- Update the jar files:

```
$ rm -rf LIB/*
$ cp $NEW/lib/*.jar LIB
```

- Start the server.
- Check the logs for any **ERROR** or **WARN** messages and if necessary correct them.

5.3 TSI

The UNICORE **T**arget **S**ystem **I**nterface (TSI) is used by the *UNICORE/X server* to perform tasks on the target resource, such as submitting and monitoring jobs, handling data, managing directories etc. It is a daemon running on the frontend(s) of the target resource (e.g. a cluster login node) which provides a remote interface to the operating system, the batch system and the file system of the target resource.

The TSI must be started by *root* on the cluster login node(s), and will run with elevated privileges. It requires an open port (default: **4433**) where it receives connections from the UNICORE/X server(s). The TSI will make outgoing connections (callbacks) to the UNICORE/X server(s). Please set up your firewall(s) accordingly. Operation through an SSH tunnel is possible as well, see the *TSI Manual* for details.

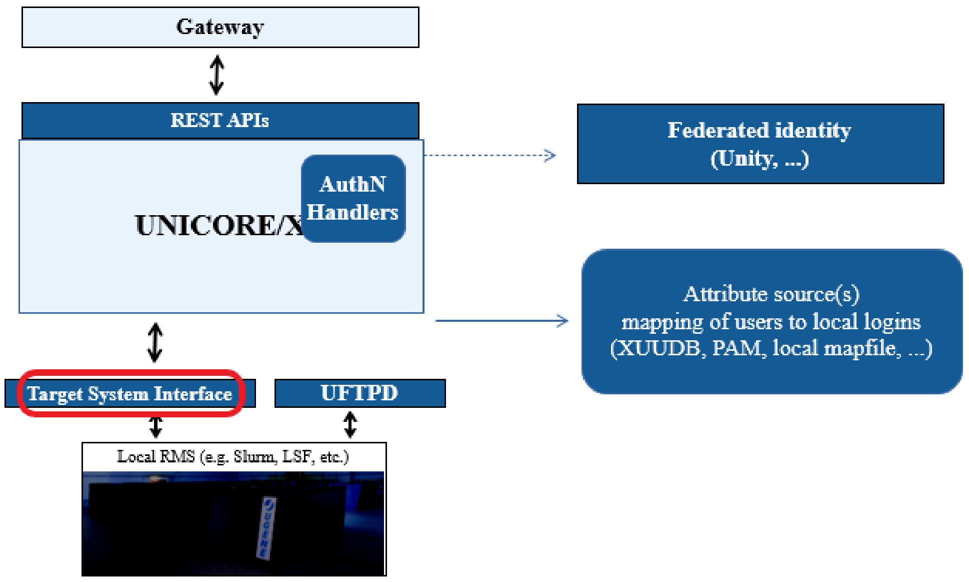

Gateway
REST APIs
UNICORE/X
AuthN Handlers
Federated identity (Unity, ...)
Attribute source(s) mapping of users to local logins (XUUDB, PAM, local mapfile, ...)
Target System Interface
UFTPD
Local RMS (e.g. Slurm, LSF, etc.)

Fig. 5.6: UNICORE TSI

TSI Manual TSI Manual with detailed instructions and examples for using the TSI.

TSI API The API to the TSI as used by *UNICORE/X*.

Building the TSI Building TSI distribution packages.

5.3.1 TSI Manual

The TSI performs the work on behalf of UNICORE users and so must be able to execute processes under different uids and gids. Therefore, in production it must be run with sufficient privileges to allow this (during development and testing it can be run as a normal user).

You can configure the TSI and *UNICORE/X* to communicate via SSL. In this case, you need a server certificate for the TSI. For details, see section *Enabling SSL for the UNICORE/X - TSI communication*.

The TSI is one point where UNICORE's seamless model meets local variations and so will usually need to be adapted to the target system. This is described in section *Adapting the TSI to your system*.

Note: In production environments, the TSI will run with elevated privileges. Make sure to read and understand section *Securing and hardening the system* on security and hardening the system.

5.3.1.1 Prerequisites

The TSI requires Python Version 3.6 or later. It works only on Unix-style operating systems (e.g. Linux or Mac OS/X), Windows is not directly supported.

The TSI uses the `setpriv` tool to run as a non-root user (*unicore*) with the capability to switch uid/gid to the requested values, in order to perform tasks on behalf of the requesting user. If this is not available on the system, the TSI will run as *root* (but never perform any actions as *root*).

Batch system status checks (e.g. via `squeue` for Slurm) will be executed under a system account (usually *unicore*) which is configured in the UNICORE/X server configuration. Note that this system account cannot be *root*, as the TSI will never execute anything as *root*.

The system account **MUST** be able to list batch job status from all users! If necessary, configure your batch system accordingly. For details on this procedure we refer to the documentation of your batch system.

If you want to run user scripts in the proper user slice, you can enable PAM, which requires an appropriate PAM module file, by default this is called `unicore-tsi`.

5.3.1.2 Installation

The TSI is available either as a generic distribution (part of the UNICORE core server bundle package, or as a separate tgz archive) or as a batch system specific package (such as an RPM, deb or tgz for Torque or Slurm) on the UNICORE repositories at sourceforge.

5.3.1.2.1 Batch system specific distribution

Use the installation tools of your operating system to install the package. The following table shows the location of the TSI files.

Name in this manual	Location	Description
CONF	/etc/unicore/tsi	Configuration files
BIN	/usr/share/unicore/tsi/bin	Start/stop scripts
LIB	/usr/share/unicore/tsi/lib	Python files
LOGS	/var/run/unicore/tsi/logs	Log files

5.3.1.2.2 Generic distribution

The generic TSI distribution contains several TSI variations for many popular batch systems.

Before being able to use the TSI, you must install one of the TSI variants and configure it for your local environment:

- Execute the installation script `Install.sh` and follow the instructions to copy all required files into a new TSI installation directory.
- Adapt the configuration as described below.

In the following, `TSI_INSTALL` refers to the directory where you installed the TSI. This has the following subdirectories:

Name in this manual	Location	Description
TSI_INSTALL		Base directory chosen during execution of `Install.sh`
CONF	TSI_INSTALL/conf	Configuration files
BIN	TSI_INSTALL/bin	Start/stop scripts
LIB	TSI_INSTALL/lib	Python files
LOGS	TSI_INSTALL/logs	Log files

5.3.1.2.3 File permissions

The permissions on the TSI files should be set to **read only for the owner**. The default installation procedure will initially take care of this. As the TSI is executed with elevated privileges, you should never leave any TSI files (or directories) writable after any update.

5.3.1.3 Configuring the TSI

The TSI is configured by editing the *CONF*/`tsi.properties` and *CONF*/`startup.properties` files. Please review these two files carefully.

Changes outside the config files should not be necessary, except for new portings and any local adaptations, as detailed in the next section. If changes are made, they should be passed on to the UNICORE developers, so that they can be incorporated into future releases of the scripts. To do that, send mail to unicore-support or use the issue tracker at sourceforge.

5.3.1.3.1 Verifying

Before starting the TSI, you should make sure that the batch system integration is working correctly. See the section on *Adapting the TSI to your system* below!

5.3.1.3.2 TSI networking configuration

In tsi.properties, the TSI host interface and port are defined, as well as the allowed UNICORE/X host(s).

```
# TSI host interface, use "0.0.0.0" to bind to all interfaces
tsi.my_addr=localhost

# The port on which the TSI will listen for UNICORE/X requests
tsi.my_port=14433

# Comma-separated list of UNICORE/X machine(s) from where
# connections are allowed
tsi.unicorex_machine=my-unicorex-a.server.org, my-unicorex-b.server.org

# Optionally, define a fixed callback port to UNICORE/X
# (If not set, the TSI will use the port requested by UNICORE/X)
tsi.unicorex_port=7654
```

NOTE: if using SSL (see section *Enabling SSL for the UNICORE/X - TSI communication*), the `tsi.unicorex_machine` is ignored.

You can optionally configure a range of local ports for the TSI to use. If this is set, the TSI will use free ports from that range only. Per UNICORE/X connection, two local ports are required, so make sure to not set this range too small (should be at least 20 ports).

```
tsi.local_portrange=50000:50100
```

5.3.1.3.3 UNICORE/X configuration

UNICORE/X configuration is described fully in the relevant *UNICORE/X manual*. Here we just give the most important steps to get the TSI up and running.

The relevant UNICORE/X config file is usually called `tsi.config`.

Hostnames and ports

UNICORE/X needs to know the TSI hostname and port:

```
CLASSICTSI.machine=frontend.mycluster.org
CLASSICTSI.port=4433
```

SSL support

If you wish to setup SSL for the UNICORE/X-to-TSI communication, please refer to section *Enabling SSL for the UNICORE/X - TSI communication*.

5.3.1.3.4 ACL support

The TSI (together with UNICORE/X) provides a possibility to manipulate file **A**ccess **C**ontrol **L**ists (ACLs). To use ACLs, the appropriate support must be available from the underlying file system. Currently, only the so called POSIX ACLs are supported (*so called* as in fact the relevant documents POSIX 1003.1e/1003.2c were never finalized), using the popular `setfacl` and `getfacl` commands. Most current file systems provide support for the POSIX ACLs.

Note: Note, that the current version is relying on extensions of the ACL commands which are present in the Linux implementation. In case of other implementation (e.g. BSD) the ACL module should be extended, otherwise the default ACLs (which are used for directories) support will not work.

To enable POSIX ACL support you typically must ensure that:

- the required file systems are mounted with ACL support turned on,
- the `getfacl` and `setfacl` commands are available on your machine.

Configuration of ACLs is performed in the `tsi.properties` file. First of all, you can define a location of `setfacl` and `getfacl` programs with `tsi.setfacl` and `tsi.getfacl` properties. By providing absolute paths you can use non-standard locations, typically it is enough to leave the default, non-absolute values which will use programs as available under the standard shell search path. Note that if you will comment any of those properties, the POSIX ACL subsystem will be turned off.

Configuration of ACL support is per directory, using properties of the format: `tsi.acl.PATH`, where *PATH* is an absolute directory path for which the setting is being made. You can provide as many settings as required, the most specific one will be used. The valid values are `POSIX` and `NONE` respectively for POSIX ACLs and for turning off the ACL support.

Consider an example:

```
tsi.acl./=NONE
tsi.acl./home=POSIX
tsi.acl./mnt/apps=POSIX
tsi.acl./mnt/apps/external=NONE
```

The above configuration turns off ACL for all directories, except for everything under `/home` and everything under `/mnt/apps` with the exception of `/mnt/apps/external`.

Warning: Do not use symbolic links or `..` or `.` in properties configuring directories - use only absolute, normalized paths. Currently spaces in paths are also unsupported.

Note: The ACL support settings are typically cached on the UNICORE/X side (for a few minutes). Therefore, after changing the TSI configuration (and after resetting the TSI) you have to wait a bit until the new configuration is applied also in UNICORE/X.

ACL limitations

There is no ubiquitous standard for file ACLs. *POSIX draft* ACLs are by far the most popular however there are several other implementations. Here is a short list that should help to figure out the situation:

- POSIX ACLs are supported on Linux and BSD systems.
- The following file systems support POSIX ACLs: Lustre, ext{2,3,4}, JFS, ReiserFS and XFS.
- Solaris ACLs are very similar to POSIX ACLs and it should be possible to use TSI to manipulate them at least partially (remove all ACL operation won't work for sure and note that usage of Solaris ACLs was never tested). Full support may be provided on request.
- NFS version 4 provides a completely different, and currently unsupported implementation of ACLs.
- NFS version 3 uses ACLs with the same syntax as Solaris OS.
- There are also other implementations, present on AIX or Mac OS systems or in AFS FS.

Note that in future more ACL types may be supported and will be configured in the same manner, just using a different property value.

5.3.1.3.5 Enabling SSL for the UNICORE/X - TSI communication

SSL support should be enabled for the UNICORE/X - TSI communication to increase security. This is a **MUST** when UNICORE/X and TSI run on the same host, and/or user login is possible on the UNICORE/X host, to prevent attackers gaining control over the TSI.

You need:

- a private key and certificate for the TSI,
- the CA certificate of the TSI certificate,
- the DN (subject distinguished name) of the UNICORE/X servers that shall be allowed to connect to the TSI,
- the CA certificate of the UNICORE/X certificate.

The certificate of the TSI signer CA must be added to the UNICORE/X truststore.

The following configuration options must be set in `tsi.properties`:

`tsi.keystore`	file containing the private TSI key in PEM format
`tsi.keypass`	password for decrypting the key
`tsi.certificate`	file containing the TSI certificate in PEM format
`tsi.truststore`	file containing the certificate of the accepted CA(s) in PEM format
`tsi.allowed_dn.NNN`	allowed DNs of UNICORE/X servers in RFC format

SSL is activated if the keystore file is specified in `tsi.properties`.

The truststore file contains the CA cert(s):

```
-----BEGIN CERTIFICATE-----

  ... PEM data omitted ...

-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
```

(continues on next page)

(continued from previous page)

```
   ... PEM data omitted ...

-----END CERTIFICATE-----
```

The `tsi.allowed_dn.NNN` properties are used to specify which certificates are allowed, for example,

```
tsi.allowed_dn.1=CN=UNICORE/X 1, O=UNICORE, C=EU
tsi.allowed_dn.2=CN=UNICORE/X 2, O=UNICORE, C=EU
```

Attention: If you do not specify any access control entries, all certificates issued by trusted CAs are allowed to connect to the TSI. Be very careful to prevent illicit access to the TSI!

When UNICORE/X connects, its certificate is checked:

- the UNICORE/X cert has to be valid (i.e. issued by a trusted CA and not expired),
- the subject of the UNICORE/X cert is checked against the configured ACL (list of allowed DNs).

On the UNICORE/X side, set the following property (usually in the `xnjs.properties` file):

```
# enable SSL using the UNICORE/X key and trusted certificates
CLASSICTSI.ssl.disable=false
```

5.3.1.4 Adapting the TSI to your system

5.3.1.4.1 Environment and paths

The environment and path settings for the main TSI process and all its child processes (TSI workers) are controlled in the `startup.properties` file.

Important: Please revise the path and environment settings in the main `startup.properties` config file.

These should include the path to all executables required by the TSI, notably the batch system commands, and if applicable, the ACL commands.

As the TSI process runs as root, and switches to the required user/group IDs before each request, setting up the required environment per user has to be done carefully. Per-user settings are usually done on the UNICORE/X level using *IDB templates*, please refer to the *UNICORE/X documentation*.

5.3.1.4.2 Assigning groups to the current user

The current user will all her groups assigned. On some systems the default Python function used for resolving a user's groups does not see all the groups. If this is the case, set in `tsi.properties`:

```
tsi.use_id_to_resolve_gids=true
```

This will use a different implementation via the system command `id -G <username>`.

5.3.1.4.3 Batch system integration: BSS.py

The file BSS.py contains the functions specific to the used batch system, specifically it prepares the job script, deals with job status reporting and job control.

Even if you run a well-supported batch system such as Torque or Slurm, you should make sure that the job status reporting works properly.

Also, any site-specific resource settings (e.g. settings related to GPUs, network topology etc) are dealt within this file.

5.3.1.4.4 Reporting free disk space

UNICORE will often invoke the `df` command which is implemented in the IO.py file in order to get information about free disk space. On some distributed file systems, executing this command can take quite some time, and it may be advisable to modify the `df` function to optimize this behaviour.

5.3.1.4.5 Reporting computing time budget

If supported by your site installation, users might have a computing time budget allocated to them. The BSS.py module contains a function `get_budget` that is used to retrieve this budget as a number e.g. representing core-hours. By default, this function returns `-1` to indicate that computing time is not budgeted.

5.3.1.4.6 Filtering cluster working nodes

Starting from version 6.5.1 the TSI can filter nodes based on the properties defined for nodes in BSS configuration. It can limit working nodes only to those having shared file system. It can be defined in the `tsi.properties` file by setting the property `tsi.nodes_filter`.

Attention: Note that this feature is not working for all batch systems. Currently, it is supported in Torque and SLURM.

5.3.1.4.7 Resource reservation

The reservation module Reservation.py is responsible for interacting with the reservation system of your batch system.

> **Attention:** Note that this feature is not available for all batch systems. Currently, it is included in Torque and SLURM.

5.3.1.5 Execution model

The main TSI process will respond to UNICORE/X requests and start up TSI workers to do the work for the UNICORE/X server. The TSI workers connect back to the UNICORE/X server.

It is possible to use the same TSI from multiple UNICORE/X servers.

Since the main TSI process runs with elevated privileges, it must authenticate the source of commands as legitimate. To do this, the TSI is initialised with the address(es) of the machine(s) that runs the UNICORE/X. The TSI will only accept requests from the defined UNICORE/X machine(s). The callback port can be pre-defined in `tsi.properties` as well. If it is undefined, the TSI will attempt to read it from the UNICORE/X connect message.

Note that it is possible to enable SSL on the TSI listen port, see below. In SSL mode, there is no check of the UNICORE/X address.

If the UNICORE/X process shuts down, any TSI workers that are connected to UNICORE/X will also shut down. However, the main TSI process will continue executing and will spawn new TSI workers processes when the UNICORE/X server is restarted. Therefore, it is not necessary to restart the TSI daemon when restarting UNICORE/X.

If a TSI worker stops execution, UNICORE/X will request a new one to replace it.

If the main TSI process stops execution, then all TSI processes will also be killed. The TSI must then be restarted, this does not happen automatically.

5.3.1.6 PAM, systemd and user slices

By default, user tasks (such as user scripts on the TSI node) will run in the same slice as the TSI itself.

You can enable PAM, which will open a user session before running the user's tasks, so the tasks will be run in the correct user slice, and thus the system's resource management will properly apply also to tasks started via UNICORE.

To do this, set in `tsi.properties`

```
tsi.open_user_sessions=1
```

By default, a PAM module `unicore-tsi` is expected (`/etc/pam.d/unicore-tsi`). For example, this could contain:

```
#%PAM-1.0
auth        sufficient    pam_rootok.so
session     required      pam_limits.so
session     required      pam_unix.so
session     required      pam_systemd.so
```

5.3.1.7 Directories used by the TSI

The TSI must have access to the *filespace* directory specified in the UNICORE/X configuration (usually the property `XNJS.filespace` in `xnjs.properties`) to hold job directories. These directories are written with the TSI's uid set to the Unix user for which the work is being performed. If you use a shared directory for all users, this directory must be world writable. The required Unix access mode is `1777`.

5.3.1.8 Running the TSI

For the Linux packages, the TSI is pre-configured for systemd, and if you want to run it as a a system service, you can use `systemctl`:

```
$ sudo systemctl add-wants multi-user.target unicore-tsi-variant
```

(where *variant* stands for the concrete TSI implementation, such as `nobatch` or `slurm`)

5.3.1.8.1 Starting

If installed from an Linux package, the TSI can be started via *systemd*:

```
$ sudo systemctl start unicore-tsi-variant
```

The TSI can also be started using the script `BIN/start.sh`.

5.3.1.8.2 Stopping the TSI

If installed from an Linux package, the TSI can be stopped via *systemd*:

```
$ sudo systemctl stop unicore-tsi-variant
```

The TSI can also be stopped using the script `BIN/stop.sh` (cf. section *Scripts*). This will stop the main TSI process and the tree of all spawned processes including the TSI workers.

TSI worker processes (but not the main process) will stop executing when the UNICORE/X server it connects to stops executing.

It is possible to stop a TSI worker process, but this could result in the failure of a job (the UNICORE/X server will recover and create new TSI processes).

5.3.1.8.3 TSI logging

By default, the TSI logs to the system journal (syslog), and you can read the logs via `journalctl`, for example,

```
$ sudo journalctl -u unicore-tsi-variant
```

To print logging output to stdout instead, set

```
tsi.use_syslog=false``
```

in the *CONF*/tsi.properties file.

Since stdout is redirected to a file (see the STARTLOG definition in CONF/startup.properties) the logging output will be in that file.

For more verbose logging, set

```
tsi.debug=true
```

in *CONF*/tsi.properties.

5.3.1.9 Porting the TSI to other batch systems

Most variations are found in the batch subsystem commands, porting to a new BSS usually requires changes to the following files:

- BSS.py
- Reservation.py (reservation functions if applicable)

It is recommended to start from a up-to-date and well-documented TSI, e.g. the Torque or Slurm variation. If you have further questions regarding porting to a new batch system, please use the unicore-support or unicore-devel mailing lists.

5.3.1.10 Securing and hardening the system

In a normal multi-user production setting, the TSI runs with elevated privileges, and thus it is critical to prevent illicit access to the TSI, which would allow accessing or destroying arbitrary user data, as well as impersonating users and generally wreaking havoc.

Once the connection to the UNICORE/X is established, the TSI is controlled via a simple text-based API. An attacker allowed to connect to the TSI can very easily execute commands as any valid (non-root) user.

In non-SSL mode, the TSI checks the IP address of the connecting process, and compare it with the expected one which is configured in the `tsi.properties` file.

In SSL mode, the TSI checks the certificate of the connecting process, by validating it against its truststore which is configured in the `tsi.properties` file.

We recommed the following measures to make operating the TSI secure:

- Prevent all access to the TSI's config and executable files. This is usually done by setting appropriate file permissions, and usually already taken care of during installation (please see the section *File permissions*).
- Make sure only UNICORE/X can connect to the TSI. This is most reliably done by configuring SSL for the UNICORE/X to TSI communication (please see the section *Enabling SSL for the UNICORE/X - TSI communication*).
- If SSL cannot be used, the UNICORE/X should run on a separate machine.
- On the UNICORE/X machine, user login should be impossible. This will prevent bypassing the IP check (in non-SSL mode) and/or accessing the UNICORE/X private key (in SSL mode).
- If you for some reason HAVE to run UNICORE/X and TSI on the same machine, and user login or execution of user commands is possible on that machine, you **MUST use SSL**, and take special care to protect the UNICORE/X config files and keystore using appropriate file permissions. Not using SSL in this situation is a serious risk! An attacker connecting to the TSI can impersonate any user and access any user's data (except for the *root* user).

- An additional safeguard is to establish monitoring for UNICORE/X, and kill the TSI in case the UNICORE/X process terminates.

Important: Summarizing, it is critical to protect config files and executable files. We strongly recommend to configure SSL. Using SSL is a **MUST** in deployments where users can login to the UNICORE/X machine.

5.3.2 TSI API

This document describes the API to the TSI as used by *UNICORE/X* (more concretely, the *XNJS* subsystem of UNICORE/X). The parts of the TSI that interact with the target system have been isolated and are documented here with their function calls.

The functions are implemented in the TSI as calls to Python methods. Input data from the UNICORE/X server is passed as arguments to the method. Output is returned to the UNICORE/X server by calling some global methods documented below or by directly accessing the TSI's command and data channels. TSIs are shipped with default implementations of all the functions and can be tailored by changing the supplied code or by implementing new versions of the functions that need to change for the system.

Note that this document is not a complete definition of the API, it is a general overview. The full API specification can be derived by reading the TSI code supplied with a UNICORE release.

5.3.2.1 Initialisation

For connecting to the UNICORE/X server, a callback mechanism is used. First, the UNICORE/X server will contact the main TSI process to request the creation of a new TSI worker process. The main TSI will call back the UNICORE/X server and create the necessary communications. It will receive any initialisation information send by the UNICORE/X server. After successful creation of the TSI worker process, the UNICORE/X server can communicate with the worker and ask it to execute commands. The UNICORE/X-to-TSI connection uses two sockets, a data and a command socket.

After initialisation is complete, the `process()` function (in the TSI.py module) is entered, which reads messages from the UNICORE/X server and dispatches processing to the various TSI functions.

5.3.2.2 Messages to the UNICORE/X server

The TSI provides methods to pass messages to the UNICORE/X server. In particular the UNICORE/X server expects every method to reply at the end of its execution. The messaging methods are implemented in Connector.py:

- `write_message(string)` Sends a message to the UNICORE/X server
- `ok(string)` Sends a message to the UNICORE/X server to say that execution of the command was successful. This message always starts with a line `TSI_OK`
- `failed(string)` Sends a message to the UNICORE/X server to say that execution of the command failed. The string is sent to the UNICORE/X server as part of the failure message. This message always starts with a line `TSI_FAILED`

Messages have to end with a special tag `ENDOFMESSAGE`, since the command sockets are left open for receiving the next command.

5.3.2.3 User identity and environment setting

In production mode the TSI will be started as a privileged user capable of changing the TSI worker process' uid and gid to the values requested by the UNICORE/X server. This change is made before the TSI executes any external actions. The idenity is passed as a line in the message string sent by the UNICORE/X server, which starts with `#TSI_IDENTITY`.

The TSI performs three types of work: the execution and monitoring of jobs prepared by the user, transfer and manipulation of files on storages and the management of Uspaces (job working directory). Only the first type of work, execution of jobs, needs a complete user environment. The other two types of TSI work use a restricted set of standard commands (`mkdir`, `cp`, `rm`, etc) and should not require access to specific environments set up by users. Furthermore, job execution is not done directly by the TSI but is passed on to the local Batch Subsystem which ensures that a full user environment is set before a job is executed. Therefore, the TSI only needs to set a limited user environment for any child processes that it creates. The TSI sets the following environment in any child process:

- `$USER` This is set to the user name supplied by the UNICORE/X server.
- `$LOGNAME` This is set to the user name supplied by the UNICORE/X server.
- `$HOME` This is set to the home directory of the user as given by the target system's password file.
- `$PATH` This is inherited from the parent TSI process (see the `tsi.properties` file).

Localisations of the TSI will also need to set any other environment necessary to access the BSS.

For testing, the TSI may be started as a non-privileged user, when no changing of uid and gid is possible.

5.3.2.4 Method dispatch

To determine which method to call, the TSI checks the message from the UNICORE/X server for the occurrence of special tags (followed by a new line). For example, the occurrence of a `#TSI_SUBMIT` tag will lead to execution of the `BSS.submit()` function. Before entering any method, user/group ID switching is performed, as explained in the previous section.

5.3.2.5 Job execution and job control functions

5.3.2.5.1 Job submission (#TSI_SUBMIT)

The `submit(string)` function submits a user script to the BSS.

Input

As input, the script to be executed is expected. The string from the UNICORE/X server is processed to replace all instances of `$USER` by the user's name and `$HOME` by the user's home directory. No further processing needs to be done on the script.

The UNICORE/X server will embed information in the script that the TSI may need to use. This information will be embedded as comments so no further processing is needed. Each piece of information will be on a separate line with the format:

```
#TSI_<name> <value>
```

If the value is the string `NONE`, then the particular information should not be supplied to the BSS during submission. The information is:

- `#TSI_JOBNAME` This is the name that should be given to the job. If this is `NONE` (or is determined to be invalid), the TSI will use a default jobname.

- `#TSI_PROJECT` The user's project (for accounting).
- `#TSI_STDOUT#` and `#TSI_STDERR` The names for standard output and error files.
- `#TSI_OUTCOME_DIR` The directory where to write the stdout and stderr files to. In general this is the same as `#TSI_USPACE_DIR#`.
- `#TSI_USPACE_DIR` The initial working directory of the script (i.e. the Uspace directory).
- `#TSI_TIME` The run time (wall clock) limit requested by this job in seconds.
- `#TSI_MEMORY#` The memory requirement of the job. The UNICORE/X server supplies this as a *megabytes per node* value.
- `#TSI_TOTAL_PROCESSORS` The number of processors required by the job.
- `#TSI_PROCESSORS` The number of processors per node required by the job.
- `#TSI_NODES` The number of nodes required by this job.
- `#TSI_QUEUE` The BSS queue to which this job should be submitted.
- `#TSI_UMASK` The default umask for the job.
- `#TSI_EMAIL` The email address to which the BSS should send any status change emails.
- `#TSI_RESERVATION_REFERENCE` If the job should be run in a reservation, this parameter contains the reservation ID.
- `#TSI_ARRAY` If multiple instances of the same job are to be submitted, this contains the list of array IDs, e.g. "1-100", or "2,4,6".
- `#TSI_ARRAY_LIMIT` If multiple instances of the same job are to be submitted, this optionally limits the number of concurrently running instances. E.g. *5* will limit the number of instances to *5*.
- `#TSI_BSS_NODES_FILTER <filterstring>` Administrators can define a string in the IDB which is to be used as nodes filter, if the BSS supports this.

In addition to these, additional site-specific resources (e.g. GPUs) can be defined on the UNICORE/X server, which are passed via `#TSI_SSR_<resource_name> <resource_value>` lines.

Output

- *Normal*: the output is the BSS identifier of the job unless the execution was interactive. In this case the execution is complete when the TSI returns from this call and the output is that from `ok()`.
- *Error*: `failed()` called with the reason for failure

Raw job submission

If the instruction `#TSI_JOB_MODE raw` is encountered in the submit script, the TSI will ignore any further instruction relevant for batch system submission. Instead a second instruction `#TSI_JOB_FILE <filename>` determines a file that will be read and used as BSS specific information.

Resource allocation job

If the instruction `#TSI_JOB_MODE allocate` is encountered in the submit script, the TSI will use the requested resources as in a normal batch job submission. The TSI will create a script that only allocates resources from the BSS, but does not launch anything. The allocation identifier will be written to a file `BSS_ALLOCATION_ID` in the working directory.

Once this job has finished, the allocation ID can be read from the `BSS_ALLOCATION_ID`, and can be used in subsequent jobs.

5.3.2.5.2 Script execution (#TSI_EXECUTESCRIPT)

The function `TSI.execute_script()` executes the script directly from the TSI process, without submitting the script to the batch subsystem. This function is used by the UNICORE/X server to create and manipulate the Uspace, to perform file management functions, etc. The UNICORE/X server also uses this to execute user defined code, for example when user precommands or postcommands are defined in execution environments.

Input

The script to be executed. The string from the UNICORE/X server is processed to replace all instances of `$USER` by the user's name and `$HOME` by the user's home directory. No further processing needs to be done on the script. If a `#TSI_DISCARD_OUTPUT` string is present, no output will be gathered.

Output

- *Normal*: The script has been executed. Concatenated stderr and stdout from the execution of the script is sent to the UNICORE/X server following the `ok()` call.
- *Error*: `failed()` called with the reason for failure.

5.3.2.5.3 Job control

- `#TSI_ABORTJOB` The `BSS.abort_job()` function sends a command to the BSS to abort the named BSS job. Any stdout and stderr produced by the job before the abort takes effect must be saved.
- `#TSI_CANCELJOB` The `BSS.cancel_job()` function sends a command to the BSS to cancel the named BSS job. Cancelling means both finishing execution on the BSS (as for abort) and removing any stdout and stderr.
- `#TSI_HOLDJOB` The `BSS.hold_job()` function sends a command to the BSS to hold execution of the named BSS job. Holding means suspending execution of a job that has started or not starting execution of a queued job. Note that suspending execution can result in the resources allocated to the job being held by the job even though it is not executing and so some sites may not allow this. This is dealt with by the relaxed post condition below. Some sites can hold a job's execution and release the resources held by the job (leaving the job on the BSS so that it can resume execution). This is called freezing. The UNICORE/X server can send a request for a freeze (`#TSI_FREEZE`) which the TSI may execute, if there is no freeze command initialised the TSI may execute a hold in its place An acceptable implementation is for `hold_job` to return without executing a command.
- `#TSI_RESUMEJOB` The `BSS.resume_job()` function sends a command to the BSS to resume execution of the named BSS job. Not that suspending execution can result in the resources allocated to the job being held by the job even though it is not executing and so some sites may not allow this. An acceptable implementation is for `resume_job` to return without executing a command (if `hold_job` did the same).

Input

All job control functions require the BSS job ID as parameter in the form `#TSI_BSSID <identifier>`.

Output

- *Normal*: the job control function was invoked. No extra output.
- *Error*: `failed()` called with the reason for failure.

5.3.2.5.4 Detailed job info (#TSI_GETJOBDETAILS)

`#TSI_GETJOBDETAILS` the `BSS.get_job_details()` function sends a command to the BSS requesting detailed information about the job. The format and content is BSS specific, and is sent to UNICORE/X without any further processing.

Input

All job control functions require the BSS job ID as parameter in the form `#TSI_BSSID <identifier>`.

Output

- *Normal*: detailed job information sent via `ok()`.
- *Error*: `failed()` called with the reason for failure.

5.3.2.5.5 Status listing (#TSI_QSTAT)

This `BSS.get_status_listing()` function returns the status of all the jobs on the BSS that have been submitted through any TSI providing access to the BSS.

This method is called with the TSI's identity set to the special user ID configured in the UNICORE/X server (`CLASSICTSI.priveduser` property). This is because the UNICORE/X server expects the returned listing to contain every UNICORE job from every UNICORE user but some BSS only allow a view of the status of all jobs to privileged users.

Input

None.

Output

- *Normal*: The first line is `QSTAT`. There follows an arbitrary number of lines, each line containing the status of a job on the BSS with the following format: `id status <queuename>`, where `id` is the BSS identifier of the job and `status` is one of: `QUEUED`, `RUNNING`, `SUSPENDED` or `COMPLETED`. Optionally, the queue name can be listed as well. The output must include all jobs still on the BSS that were submitted by a TSI executing on the target system (including all those submitted by TSIs other than the one executing this command). The output may include lines for jobs on the BSS submitted by other means.
- *Error*: `failed()` called with the reason for failure.

5.3.2.5.6 Getting the user's remaining compute budget (#TSI_GET_COMPUTE_BUDGET)

This `BSS.get_budget()` function returns the remaining compute budget for the user (in core hours) or `-1` if not known or not applicable.

Input

None.

Output

- *Normal*: Budget info (see format below) is sent via `ok()`.
- *Error*: `failed()` called with the reason for failure.

Format

The output is a multiline string which each line of the form

```
<PROJECT> <ABSOLUTE_BUDGET> <PERCENTAGE> <UNITS>
```

where,

PROJECT	the project / budget account name
ABSOLUTE_BUDGET	the absolute value (integer) of compute time remaining
PERCENTAGE	the relative amount (integer, 0-100) of compute time remaining
UNITS	the units used (should be one of: `core-h`, `node-h`, `cpu-h`)

5.3.2.6 I/O functions

5.3.2.6.1 Reading a file (#TSI_GETFILECHUNK)

The `IO.get_file_chunk()` function is called by the UNICORE/X server to fetch the contents of a file.

Input

- `#TSI_FILE <file name>` The full path name of the file to be sent to the UNICORE/X server
- `#TSI_START <start byte>` Where to start reading the file
- `#TSI_LENGTH <chunk length>` How many bytes to return

The file name is modified by the TSI to substitute all occurrences of the string `$USER` by the name of the user and all occurrences of the string `$HOME` by the home directory of the user.

Output

- *Normal*: The UNICORE/X server has a copy of the request part of the file (sent via the data socket).
- *Error*: `failed()` is called with the reason for failure.

5.3.2.6.2 Writing files (#TSI_PUTFILECHUNK)

The `put_file_chunk()` function is called by the UNICORE/X server to write the contents of one file to a directory accessible by the TSI.

Input

- The `#TSI_FILESACTION` parameter contains the action to take if the file exists (or does not):
  - `0` = don't care,
  - `1` = only write if the file does not exist,
  - `2` = only write if the file exists,
  - `3` = append to file.
- The `#TSI_FILE` parameter contains the filname and permissions.
- The `#TSI_LENGTH` parameter contains the number of bytes to read from the data channel and write to disk.

The TSI replies with `TSI_OK`, and the data to write is then read from the data channel.

Output

- *Normal*: The TSI has written the file data.
- *Error*: `failed()` called with the reason for failure.

5.3.2.6.3 File ACL operations (#TSI_FILE_ACL)

The `process_acl` function allows to set or get the access control list on a given file or directory. Please refer to the file ACL.py to learn about this part of the API.

5.3.2.6.4 Listing directories and getting file information (#TSI_LS)

This function allows to list directories or get information about a single file.

Input

- The `#TSI_FILE` parameter contains the file/directory name.
- The `#TSI_LS_MODE` parameter contains the kind of listing:
  - `A` = info on a single file,
  - `R` = recursive directory listing,
  - `N` = normal directory listing.

Output

- *Normal*: The TSI writes the listing to the command socket, see the IO.py file for a detailed description of the format
- *Error*: TSI replies with `TSI_FAILED` and the reason for failure.

5.3.2.6.5 Getting free disk space (#TSI_DF)

This function allows to get the free disk space for a given path.

Input

The `#TSI_FILE` parameter contains the file/directory name.

Output

- *Normal*: The TSI writes the disk space info to the command socket, see the IO.py file for a detailed description of the format.
- *Error*: TSI replies with `TSI_FAILED` and the reason for failure.

5.3.2.7 Resource reservation functions

The TSI offers functionality to create and manage reservations. These are implemented in the file Reservation.py, different versions for different scheduling systems exist.

5.3.2.7.1 Creating a reservation (#TSI_MAKE_RESERVATION)

This is used to create a reservation.

Input

- `#TSI_RESERVATION_OWNER <xlogin>`: The user ID (xlogin) of the reservation owner,
- `#TSI_STARTTIME <time>`: The requested start time in ISO8601 format (*yyyy-MM-dd*`T` *HH:mm:ss*`Z`),
- The requested resources are passed in in the same way as for job submission.

Output

- *Normal*: The command replies with a single reservation ID string.
- *Error*: `failed()` called with the reason for failure.

Querying a reservation (#TSI_QUERY_RESERVATION)

This is used to query the status of a reservation.

Input

- `#TSI_RESERVATION_REFERENCE <reservation_ID>`: The reservation reference that shall be queried.

Output

- *Normal*: The command produces two lines. The first line contains the status (UNKNOWN, INVALID, WAITING, READY, ACTIVE, FINISHED or OTHER) and an optional start time (*ISO 8601*). The second line contains a human-readable description.
- *Error*: `failed()` called with the reason for failure.

5.3.2.7.2 Cancelling a reservation (#TSI_CANCEL_RESERVATION)

This is used to cancel a reservation.

Input

- #TSI_RESERVATION_REFERENCE <reservation_ID>: The reservation reference that is to be cancelled.

Output

- *Normal*: `ok()` called with no special output.
- *Error*: `failed()` called with the reason for failure.

5.3.2.8 Miscellaneous functions

5.3.2.8.1 Getting TSI version information (#TSI_PING)

The `TSI.ping()` function returns the TSI version.

Input

None.

Output

- TSI version string as defined in the TSI.py file

5.3.2.8.2 Getting user information (#TSI_GET_USER_INFO)

The `TSI.get_user_info()` function returns the user's HOME directory, and a list of public keys, which is read froma list of configurable files in the user's HOME directory (defaulting to `.ssh/accepted_keys`).

Input

None.

Output

- User info (format below) is sent via +message()+

Format

The output is a multiline string

```
home: <user_home_directory>
Accepted key 1: <public_key_1>
Accepted key 2: <public_key_2>
 ...
status: <status message>
```

5.3.3 Building the TSI

Clone the git repository:

```
$ git clone https://github.com/UNICORE-EU/tsi
$ cd tsi
```

Use the supplied `Makefile` to run tests and/or build packages for the various supported batch systems.

You will need Java, Maven and Ant to build *RPM/DEB* packages.

5.3.3.1 Packaging

Run

```
$ make <bss>-<type>
```

where **<bss>** is one of: `nobatch`, `slurm`, `torque`, `lsf` and **<type>** is one of: `tgz`, `deb`, `rpm`.

5.3.3.2 Generic binary TGZ

To create a *generic* binary archive that can be used to install any version of the TSI via the `Install.sh` script, run

```
$ make clean tgz
```

5.4 XUUDB

The UNICORE XUUDB is a service acting as an *attribute source* as part of a UNICORE installation. It is an optional service, that is best suited as a per-site service, providing attributes for multiple *UNICORE/X*-like services at a site.

The XUUDB maps a UNICORE user identity (which is formally an X.500 **d**istinguished **n**ame (**DN**)) to a set of attributes. The attributes are typically used to provide local account details (uid, gid(s)) and commonly also to provide authorization information, i.e. the user's role.

XUUDB Manual User Manual with detailed instructions and examples for using the XUUDB.

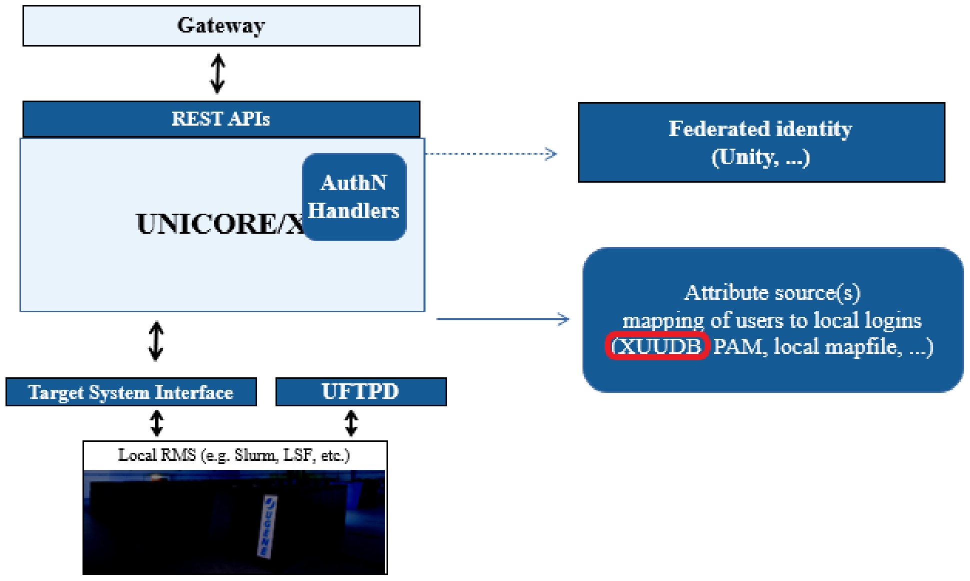

Gateway
REST APIs
UNICORE/X
AuthN Handlers
Federated identity (Unity, ...)
Attribute source(s) mapping of users to local logins (XUUDB PAM, local mapfile, ...)
Target System Interface
UFTPD
Local RMS (e.g. Slurm, LSF, etc.)

Fig. 5.7: UNICORE XUUDB

5.4.1 XUUDB Manual

The XUUDB server is an Attribute Source implementation which can be used by UNICORE servers. It is used to map a UNICORE identity (an X500 distinguished name) to authorisation and incarnation attributes. The XUUDB is also capable of performing dynamic mappings of incarnation attributes, using a rule engine.

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

5.4.1.1 Overview

The UNICORE XUUDB is used to map a UNICORE user identity (an X.500 **d**istinguished **n**ame (**DN**)) to a set of attributes. The attributes are typically used to provide local account details (uid, gid(s)) and sometimes also to provide authorization information, i.e. the user's role.

The UNICORE XUUDB is best suited as a site-service. Theoretically, it can be used for multiple sites, however as it offers limited authorization capabilities and doesn't allow for grouping users, it is better to use the more flexible Unity server in such a case. In case of the simple one host-service XUUDB sometimes can be replaced by a simple file storing the mappings. Please refer to the *UNICORE/X documentation* for more information.

The XUUDB offers two web services, one for querying, and one for administration of the users' database. There are several clients which can use the XUUDB server:

- Admin client (see *The admin client*) can be used to control the XUUDB database contents.
- UNICORE servers include the XUUDB client code (it is named XUUDB Attribute Information Point) and can consume and process the XUUDB information.

Both admin and client access to the XUUDB can be protected by a client-authenticated SSL.

The XUUDB can map users using two different mechanisms:

- *classic* or *static* mechanism, where administrator enters mappings for each DN manually,

- *dynamic* mechanism, where administrator only define rules stating what attributes should be assigned to UNICORE users fulfilling rule's condition.

5.4.1.1.1 The classic mapping

The classic or static mechanism when UNICORE is used as a gateway to HPC site, with a well defined set of users. It is also useful in federated scenarios when a dedicated, external infrastructure is build to maintain a global list of users.

Using it it is possible to set a list of UNIX logins (aka XLogins or uids), a list of UNIX groups (aka projects or gids) and the role attribute used for authorization. The first uid and the first gid is assumed to be the default one but Grid users are allowed to choose any of the available.

In case of the default authorization policy the *user* role is required to get a normal access to the site, the *admin* role grants super-user privileges, and the *banned* role bans the user.

The XUUDB stores and compares only distinguished names (DNs), not full certificates.

Multiple UNICORE sites can share the XUUDB, even if the attributes are different per UNICORE site. Sites are grouped by the so-called GCID (grid component ID).

5.4.1.1.2 The dynamic mapping

The dynamic mechanism is used to map users who *were already authorized*, therefore it doesn't make sense (and is not possible) to assign the authorization attributes as *role*. The dynamic mechanism is useful in deployments where a site doesn't know the precise list of its users (which are maintained externally), or simply doesn't want to define local accounts for each UNICORE user. In other words, the site relies on a trusted 3rd party to maintain a list of genuine and authorized users, and automatically assigns a local account to each user.

As it will be shown later on dynamic mappings can be also used in other scenarios, also being complementary to static mappings.

Dynamic mappings configuration is described in the section *Dynamic mappings configuration*.

Important: IMPORTANT NOTE ON PATHS

XUUDB is distributed either as an platform independent and portable bundle (as part of the UNICORE quickstart package) or as an installable, platform dependent package such as RPM.

Depending on the installation package used paths are different. If installing using distribution-specific package the following path prefixes are used:

```
CONF=/etc/unicore/xuudb
BIN=/usr/sbin
ADMIN=/usr/sbin/unicore-xuudb-admin
LOG=/var/log/unicore/xuudb
```

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

```
CONF=INST/conf
BIN=INST/bin
ADMIN=BIN/admin.sh
LOG=INST/log
```

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

5.4.1.2 Installation

The UNICORE XUUDB is distributed in the following formats:

- As a part of a platform independent installation bundle called UNICORE core server bundle.
- As a platform-specific binary package. available for RedHat (Centos) and Debian platforms. Those packages are not tested on all possible platforms, but should work without any problems with other versions of similar distributions.

In both cases an installation of XUUDB installs both XUUDB Server and XUUDB admin client.

After installing the server you will have to configure it. This is described in the section *The XUUDB server*.

5.4.1.2.1 Installation from Core Server Bundle

Download the core server bundle from the UNICORE project website at https://sourceforge.net/projects/unicore/files/Servers/Core.

5.4.1.2.2 Installation from RPM or DEB packages

An up-to-date list of the available repositories is given on the following page https://sourceforge.net/p/unicore/wiki/Linux_Repositories/.

The preferred way is to use Yum or APT to install (and subsequently update) XUUDB.

After the repository is configured, the following command will install the XUUDB:

```
$ yum install unicore-xuudb
```

or in case of Debian systems:

```
$  apt-get install unicore-xuudb
```

5.4.1.3 The XUUDB server

5.4.1.3.1 Security

Usually, client-authenticated SSL is used to protect the XUUDB. For this you will need certificates for the XUUDB server and all Grid components that want to talk to the XUUDB. In general, the UNICORE servers (like *UNICORE/X*) and the XUUDB-admin client need to connect to the XUUDB-server. To make SSL connections possible, you have to put the following certificates as trusted certs into the XUUDB's server truststore:

- CA certificate(s) of the UNICORE/X server(s) that query the XUUDB
- CA certificate(s) of the XUUDB-admin user certificate(s)

and XUUDB's CA certificate in the truststores of its clients.

XUUDB server may be run using a plain HTTP port. Then there is no access control at all, so this mode is useful only in environments where XUUDB port is fully protected otherwise against unauthorised access.

5.4.1.3.2 Administrative access

The XUUDB provides two kinds of web service interfaces, one for querying the XUUDB (i.e. mapping UNICORE users to local UNIX users), and a second one for administration of the XUUDB (i.e. adding and editing entries). All access to the XUUDB (including the administration utility!) is through these web services. To prevent arbitrary users from modifying the XUUDB, the administrative interface has to be protected.

To protect the administrative interface, an ACL file is used, which is a plain text file containing the distinguished names of the administrators. At least, it has to contain the DN of the certificate used by the administration utility.

As the static XUUDB data is rather sensitive (at least if privacy of the users is a concern) and dynamic mappings often require some local modifications (e.g. assigning an account from a pool) it is often desirable to protect also the query operations. The XUUDB server offers such an option (see *Base server settings*).

The ACL file can be changed at runtime to easily add or remove administrators.

To change the location of the ACL file, edit the server configuration and set a configuration parameter (see *Base server settings*).

The ACL entries are expected in the RFC 2253 format. To get the name of a certificate in the correct format using openssl, you can use the following OpenSSL command:

```
$ openssl x509 -in demouser.pem -noout -subject -nameopt RFC2253
```

5.4.1.3.3 Configuration

By default, the configuration is defined in the file `CONF/xuudb_server.conf`. To use a different configuration file, edit the start script, or use `--start <config_file>` as command line arguments when starting.

The server's configuration file allows for setting the general XUUDB settings, database backend settings, advanced HTTP server settings and finally (for secure HTTPS URLs) the server's truststore and credential. The available properties are described in the following sections.

For production deployments you should review the listen address and setup correctly truststore and credential. Defaults for the embedded database configuration and HTTP server settings are usually fine. In case if you plan to use dynamic mappings, also the dynamic mapping rules need to be provided.

Base server settings

Property name	Type	Default value / mandatory	Description
xuudb.aclFile	filesystem path	*mandatory*	File with DNs of servers/clients authorised to access protected XUUDB services.
xuudb.address	string	http://localhost:34463	HTTPS or HTTP URL where the server should listen.
xuudb.db..*	string *can have subkeys*		Properties with this prefix are used to configure database backend, used by XUUDB. See separate documentation for details.
xu-udb.dynamicAttributesConfig	filesystem path	conf/dynamicAttributesCfg.xml	File with configuration of the dynamic part of the XUUDB.
xuudb.protectAll	[true, false]	false	If true then access to both query and modify operations are protected by ACL. If false then only modification operations are protected.

Database settings

The XUUDB can be configured to use different database backends. The embedded H2 database and external MySQL and PostgreSQL are supported. H2 database (the default) requires no additional configuration actions. In any case XUUDB will automatically create the required database tables.

For MySQL you have to properly set up the server and create a database. After installing and starting the MySQL server login to its using MySQL client as administrator and using a commands similar to the below ones, create a database and assign full access to a xuudb user.

```
create database xuudb;
grant all on xuudb.* to 'xuudbuser'@'127.0.0.1' identified by 'pass';
```

For PostgreSQL, the commands to create the DB and the required user would be (similar to):

```
sudo -u postgres createuser -P unicore
sudo -u postgres createdb -O unicore xuudb
```

Check that the PostgreSQL server allows for password authentication for the UNICORE user. Ensure that in pg_hba.conf you have lines similar to these:

```
host   all         all         127.0.0.1/32          md5
host   all         all         ::1/128               md5
```

Of course you are free to choose different names for the user, password and database. If XUUDB server is installed on other host then the proper address must be set instead of localhost.

Use the following properties to configure database connection from the XUUDB server. In case of external database pay attention to enter proper values.

Table 5.8: XUUDB Server database configuration

Property name	Type	Default value / mandatory	Description
xuudb.db.charset	string	utf8	(MySQL) Charset to use for XUUDB tables.
xuudb.db.dialect	[h2, mysql, pgsql]	h2	Database SQL dialect. Must match the selected driver, however sometimes more then one driver can be available for a dialect.
xuudb.db.driver	Class extending java.sql.Driver	org.h2.Driver	Database driver class name. This property is optional - if not set, then a default driver for the chosen database type is used.
xuudb.db.jdbcUrl	string	jdbc:h2:data/xuudb2	Database JDBC URL.
xuudb.db.password	string	*empty string*	Database password.
xuudb.db.username	string	sa	Database username.

Configuring advanced HTTP server settings

UNICORE servers are using an embedded Jetty HTTP server. In most cases the default configuration should be perfectly fine. However, for some sites (e.g. experiencing an extremely high load) HTTP server settings can be fine-tuned with the following parameters:

Property name	Type	Default value / mandatory	Description
xuudb.httpServer.CORS_allowedHeaders	string	*	CORS: comma separated list of allowed HTTP headers (default: any)
xuudb.httpServer.CORS_allowedMethods	string	GET,PUT,POST,DELETE,HEAD	CORS: comma separated list of allowed HTTP verbs.
xuudb.httpServer.CORS_allowedOrigins	string	*	CORS: allowed script origins.
xuudb.httpServer.CORS_chainPreflight	[true, false]	false	CORS: whether preflight OPTION requests are chained (passed on) to the resource or handled via the CORS filter.
xuudb.httpServer.CORS_exposedHeaders	string	Location,Content-Type	CORS: comma separated list of HTTP headers that are allowed to be exposed to the client.
xuudb.httpServer.disabledCipherSuites	string	*empty 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
xuudb.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.
xuudb.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.
xuudb.httpServer.enableSNI	[true, false]	false	Enable Server Name Indication (SNI)
xuudb.httpServer.fastRandom	[true, false]	false	Use insecure, but fast pseudo random generator to generate session ids instead of secure generator for SSL sockets.
xuudb.httpServer.gzip.enable	[true, false]	false	Controls whether to enable compression of HTTP responses.
xuudb.httpServer.gzip.minGzipSize	integer number	100000	Specifies the minimal size of message that should be compressed.
xuudb.httpServer.maxConnections	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.
xuudb.httpServer.maxIdleTime	integer >= 1	200000	Time (in ms.) before an idle connection will time out. It should be large enough not to expire connections with slow clients, values below 30s are getting quite risky.
xuudb.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.
xuudb.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.
xuudb.httpServer.requireClientAuthn	[true, false]	true	Controls whether the SSL socket requires client-side authentication.
xuudb.httpServer.wantClientAuthn	[true, false]	true	Controls whether the SSL socket accepts (but does not require) client-side authentication.
xuudb.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]'

Example

In this example we will turn on compression of all responses bigger then 50kB (assuming that the client supports decompression). Additionally, we are limiting the number of concurrent clients that can be served to more or less 50, while keeping 10 threads ready all the time to server new clients quicker.

```
jetty.gzip.enable=true
jetty.gzip.minGzipSize=51200
jetty.maxThreads=50
jetty.minThreads=10
```

5.4.1.3.4 Configuring PKI trust settings

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

Certificates validation is primarily configured using a set of initially trusted certificates of so called **C**ertificate **A**uthorities (**CA**s). Those trusted certificates are also known as *trust anchors* and their collection is called a *trust store*.

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

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

- *Keystore trust store* - the only format supported in older UNICORE versions. Trusted certificates are stored in a single binary file in JKS or PKCS12 format. The file can be only manipulated using a special tool like JDK keytool or openssl (in case of PKCS12 format). This format is great if trust store should be in a single file or when compatibility with other Java solutions or older UNICORE releases is desired.
- *OpenSSL trust store* - allows to use a directory with CA certificates stored in PEM format, under precisely defined names: the CA certificates, CRLs, signing policy files and namespaces files are named `<hash>.0`, `<hash>.r0`, `<hash>.signing_policy` and `<hash>.namespaces`. Hash is the old hash of the trusted CA certificate subject name (in Openssl version > 1.0.0 use `-subject_hash_old switch` to generate it). If multiple certificates have the same hash then the default zero number must be increased. This format is the same as used by other then UNICORE popular middlewares as Globus and gLite. It is suggested when a common trust store with such middlewares is needed.
- *Directory trust store* - the most flexible and convenient option, suggested for all remaining cases. It allows to use a list of wildcard expressions, concrete paths of files or even URLs to remote files as a set of trusted CAs and in the same way for the CRLs. With this trust store administrator can simply configure all files (or all with a specified extension) in a directory to be used as a trusted certificates.

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

Property name	Type	Default value / mandatory	Description
xuudb.truststore.allowProxy	[ALLOW, DENY]	ALLOW	Controls whether proxy certificates are supported.
xuudb.truststore.type	[keystore, openssl, directory]	*mandatory*	The truststore type.
xuudb.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)
xuudb.truststore.directoryConnectionTimeout	integer number	15	Connection timeout for fetching the remote CA certificates in seconds.
xuudb.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.
xuudb.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.
xuudb.truststore.directoryLocations.*	list of properties with a common prefix		List of CA certificates locations. Can contain URLs, local files and wildcard expressions.(runtime updateable)
xuudb.truststore.keystoreFormat	string		The keystore type (jks, pkcs12) in case of truststore of keystore type.
xuudb.truststore.keystorePassword	string		The password of the keystore type truststore.
xuudb.truststore.keystorePath	string		The keystore path in case of truststore of keystore type.
xuudb.truststore.opensslNewStoreFormat	[true, false]	false	In case of openssl truststore, specifies whether the trust store is in openssl 1.0.0+ format (true) or older openssl 0.x format (false)
xuudb.truststore.opensslNsMode	[GLOBUS_EUGRIDPMA, EU-GRIDPMA_GLOBUS, GLOBUS, EU-GRIDPMA, GLOBUS_EUGRIDPMA_REQUIRE, EU-GRIDPMA_GLOBUS_REQUIRE, GLOBUS_REQUIRE, EU-GRIDPMA_REQUIRE, EU-GRIDPMA_AND_GLOBUS, EU-GRIDPMA_AND_GLOBUS_REQUIRE, IGNORE]	EU-GRIDPMA_GLOBUS	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).
xuudb.truststore.opensslPath	filesystem path	/etc/grid-security/certificates	Directory to be used for opeenssl truststore.
xuudb.truststore.crlConnectionTimeout	integer number	15	Connection timeout for fetching the remote CRLs in seconds (not used for Openssl truststores).
xuudb.truststore.crlDiskCachePath	filesystem path		Directory where CRLs should be cached, after downloading them from remote source. Can be left undefined if no disk cache should be used. Note that directory

Examples

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

Directory trust store, with a minimal set of options:

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

Directory trust store, with a complete set of options:

```
truststore.type=directory
truststore.allowProxy=DENY
truststore.updateInterval=1234
truststore.directoryLocations.1=/trust/dir/*.pem
truststore.directoryLocations.2=http://caserver/ca.pem
truststore.directoryEncoding=PEM
truststore.directoryConnectionTimeout=100
truststore.directoryDiskCachePath=/tmp
truststore.crlLocations.1=/trust/dir/*.crl
truststore.crlLocations.2=http://caserver/crl.pem
truststore.crlUpdateInterval=400
truststore.crlMode=REQUIRE
truststore.crlConnectionTimeout=200
truststore.crlDiskCachePath=/tmp
```

Openssl trust store:

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

Java keystore used as a trust store:

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

Configuring the credential

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

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

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

Property name	Type	Default value / mandatory	Description
xu-udb.credential.path	filesystem path	*mandatory*	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.
xu-udb.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).
xu-udb.credential.password	string		Password required to load the credential.
xu-udb.credential.keyPath	string		Location of the private key if stored separately from the main credential (applicable for 'pem' and 'der' types only),
xu-udb.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.
xu-udb.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'.
xu-udb.credential.reloadOnChange	[true, false]	true	Monitor credential location and trigger dynamical reload if file changes.

Examples

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

Credential as a pair of DER files:

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

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

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

5.4.1.3.5 Dynamic mappings configuration

Dynamic mappings are configured with a set of rules. When designing rules it is good to remember that all users, which will be evaluated, were already successfully authorized.

Each rule has a condition which selects users and a list of mappings which should be applied for the selected users. Example conditions (in English):

- all members of a */vo.wonderworld.gov*
- all (authorized) users
- all users having extra attribute *matlabAllowed* with any value AND being member of a subgroup of */vo.wonderworld.gov/dynamic/*

Example mappings (in English):

- add user a supplementary group *matlab*
- assign uid from a pool of existing uids
- assign a fixed gid *grid*
- invoke an external program and use its standard output as users gid

Precisely speaking, a mapping must have defined:

- what attribute it maps: *uid*, (primary)*gid* or *supplementaryGids*
- using what method: *fixed*, *pool* or *script*

Additionally, one can define an optional parameter stating if the mapping should overwrite an attribute value which was previously set (either by an earlier rule or assigned using a different attribute source).

As it was mentioned, there are three kinds of mappings. Let's shortly introduce them one by one.

Fixed mappings

Fixed mappings are the most basic option. The mapping is formed by a simple assignment of a fixed value. It can be used to:

- assign a common (shared!) uid to selected users (rarely used)
- assign a fixed gid to selected users (very useful to assign a gid to all Grid users, or all members of a VO)
- assign some supplementary gids to selected users (useful to provide additional local permissions to users having a special role/attributes/etc.)

The example in the pool mappings section contains also a fixed mapping.

Script mappings

Script mappings are the *Do It Yourself* mechanism. You can provide a command line which will be parsed and invoked. The application must return (on its standard output) a string with a mapping result (depending on what is mapped - gid, uid or a space separated list of supplementary gids). Of course the script can be informed who is actually being mapped, by using parameters enclosed in `${}`. The list of available parameters is given below:

- `userDN` user's DN
- `issuerDN` user's certificate issuer's DN
- `role` user's role
- `vo` user's selected VO
- `extraAttributes` map with extra attributes, names are the keys
- `xlogin` user's uid (if already established)
- `gid` user's gid (if already established)
- `supplementaryGids` user's supplementary gids (if already established)
- `xloginSet` whether uid was set
- `gidSet` whether gid was set
- `dryRun` whether the current invocation is only a simulation, and shouldn't affect any persisted system settings

The example below contains also a script mapping.

Pool mappings

Finally, the *pool mappings* are both flexible and relatively easy to use — it is the most advanced mapping type. Using the pool mapping you have to prepare a set of reserved identifiers (uids or gids depending on what is mapped). The related system accounts can be precreated or can be created on-demand. The pool mapping is configured with an additional, very important parameter: pool key. Pool key is a name of one of the user's attributes: `userDN`, `issuerDN` (DN of CA which issued user's certificate), `role`, `vo` or any other generic user's attribute.

To explain how the pool works let's assume that key is set to `userDN`. Then the pool will map a user as follows: first it is checked if there is an existing mapping bound to the user's DN. If it is found then it is simply returned. If not (the user is trying to use the site for the first time) a new identifier is selected from the pool, and stored under the key being the user's DN. Then the new identifier is returned.

Therefore, all users having the same value of the pool key will get the same mapping and vice versa. If DN is the key then all users will have a distinct mapping (useful for uids or for gids, if every user should get a unique one). If, for instance, a VO is the key then all VO members will have the same mapping (useful for gid, or for uid if all VO members should have the same user account).

The following example should help to understand those concepts and is also providing a basic syntax reference:

```
<?xml version="1.0" encoding="UTF-8"?>
<dynamicAttributes xmlns="http://unicore.eu/xuudb/dynamicAttributesRules"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <rules>
    <!-- all members of the vo /vo.wonderworld.gov should have a common uid 'shared_user'-
↪->
    <rule>
```

(continues on next page)

(continued from previous page)

```
      <condition>vo.matches("/vo.wonderworld.gov/.*")</condition>
      <mapping type="fixed" maps="uid">shared_user</mapping>
    </rule>

    <!-- all users with a role 'admin' should get a primary gid from the 'admins-pool' pool.
    For pools the 'maps' parameter is optional - it is better to specify it in the pool␣
↪definition,
    below. -->
    <rule>
      <condition>role="admin"</condition>
      <mapping type="pool">admins-pool</mapping>
    </rule>

    <!-- all users from the /biology VO get an uid from the pool and a fixed primary gid
↪'biol' -->
    <rule>
      <condition>vo.matches("/biology/.*")</condition>
      <mapping type="pool">biology-uids-pool</mapping>
      <mapping type="fixed" maps="gid">biol</mapping>
    </rule>

    <!-- complicated condition: all users who have a generic attribute 'matlabAllowed'␣
↪set AND the value
        of this attribute is 'true' get a supplementary group 'matlab' -->
    <rule>
      <condition>attributes["matlabAllowed"] != null and attributes["matlabAllowed"].
↪contains("true")</condition>
      <mapping type="fixed" maps="supplementaryGids">matlab</mapping>
    </rule>

    <!-- all (authorized) users, who do not have an uid set (overwriteExisting=false)
        should have an uid assigned by a script /usr/local/bin/create-mapping.pl. The␣
↪script will be called
        with two arguments: user's DN and VO.
    <rule>
      <condition>true</condition>
      <mapping type="script" maps="uid" overwriteExisting="false">/usr/local/bin/create-
↪mapping.pl "${userDN}" "${vo}" </mapping>
    </rule>
  </rules>

  <!-- Here come pools -->
  <pools>
    <!-- pool 'admins-pool' maps gids. The list of gids provides groups which were
    pre-created in the system. The gids will be stored per-user dn, so every admin will␣
↪get another group.
    Finally the list of gids uses special expressions where number ranges are provided.
    -->
    <pool id="admins-pool" type="gid" key="dn" precreated="true">
      <id>admin_grp[1-100]</id>
      <id>admin_grp[200-1000]</id>
    </pool>
```

(continues on next page)

(continued from previous page)

```
    <!-- This pool identifiers are loaded from an external file -->
    <pool id="biology-uids-pool" type="uid" key="dn" precreated="true">
      <file>src/test/resources/externalUidsPool</file>
    </pool>
  </pools>
</dynamicAttributes>
```

Usage of pools brings several issues regarding old mappings removal and notifications about pools getting empty. In the first case it suggested not to remove the users for the time a VO or Grid is supported: it is a simplest approach, and nowadays operating systems can support thousands of uids without any problem (Linux can have 32bit uid numbers).

In case a site wants to recycle mappings, XUUDB offers the following mechanism:

- Inactive mappings can be automatically (after a configurable time threshold) or manually (using the admin client) *frozen*. An identifier belonging to a frozen mapping is still assumed to be occupied, but the mapped user won't have it assigned (in the unprobable case that she returns to the site). Freezing is introduced to give a time for tidying up local resources assigned to the identifier. Such cleaning must be done manually and should include removal of all owned files and killing any processes. Of course, this depends whether the identifier was a gid or uid. Also please note that in case of clusters, all nodes should be cleaned up.
- After the clean up is done, the frozen mapping can be removed, again manually using the admin client or automatically, after staying in the frozen state for a specified amount of time. Note that it is impossible to remove an alive mapping.

If administrator is able to provide scripts which performs cleanup, then it is possible to invoke them upon pool mapping freezing and automate the whole process. In a similar way other handlers may be configured and XUUDB will invoke them to notify about mappings removal, assignment of a new mapping (useful when accounts are not pre-created but should be created on demand) and also when a pool is getting empty.

The following example shows all the possible handlers and lists arguments which are passed to them. As it can be seen all pool options including handlers, can be configured globally or per-pool.

```
<?xml version="1.0" encoding="UTF-8"?>
<dynamicAttributes xmlns="http://unicore.eu/xuudb/dynamicAttributesRules"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- how often (in s) pools should be checked for old or inactive mappings -->
  <poolMonitoringDelay>300</poolMonitoringDelay>
  <defaultConfiguration>
        <!-- in seconds: automatic freezeing (time measured from last mapping use)... -->
        <automaticFreezeAfter>3600000</automaticFreezeAfter>
        <!-- ... and final removal (time measurd from mapping freeze) -->
        <automaticDeleteAfter>36000</automaticDeleteAfter>
        <!-- when less then this free mappings are left generate a warning -->
        <emptyWarningAbsolute>20</emptyWarningAbsolute>
        <!-- when less then this percent of free mappings is left generate a warning -->
        <emptyWarningPercent>5</emptyWarningPercent>

        <!-- timeout for running ALL external programms -->
        <handlerInvocationTimeLimit>10000</handlerInvocationTimeLimit>

        <!-- Various handlers. Arguments are pool.getId(), pool.getType().toString(),
                                      bean.getEntry(), oldSec+"" -->

        <!-- Handler invoked before freezing an account.
             Arguments: <poolId> <poolType> <identifier> <inactiveForInSeconds>
```

(continues on next page)

(continued from previous page)

```
            If handler returns a non-zero exit status then the freezing is skipped
           (unless invoked by admin-client).
        -->
        <handlerAboutToFreeze>/opt/handlers/releaseAccountResources.sh</
↪handlerAboutToFreeze>

        <!-- Handler invoked before deleting a frozen identifier.
             Arguments: <poolId> <poolType> <identifier> <frozenForInSeconds>
             If handler returns a non-zero exit status then the deletion is skipped
             (unless invoked by admin-client).
             -->
        <handlerAboutToDelete>/opt/handlers/notifyAccountRecycled.sh</
↪handlerAboutToDelete>

        <!-- Handler invoked when an identified from the uids pool is going to be used␣
↪for the first time
        (or for the first time after deleting it), if the pool is set as not pre-created.
             Arguments: <poolId> <uid> <key>
        -->
        <handlerCreateSystemUid>/opt/handlers/adduser.sh</handlerCreateSystemUid>

        <!-- Handler invoked when an identified from the gids pool is going to be used␣
↪for the first time
        (or for the first time after deleting it), if the pool is set as not pre-created.
             Arguments: <poolId> <gid> <key>
        -->
        <handlerCreateSystemGid>/opt/handlers/addgroup.sh</handlerCreateSystemGid>

        <!-- Handler invoked when a pool warning threshold is exceeded.
             Arguments: <poolId> <poolType> <remainingFreeIds>
        -->
        <handlerPoolGettingEmpty>/opt/handlers/notifyNearlyEmpty.sh</
↪handlerPoolGettingEmpty>

        <!-- Handler invoked when a pool gets empty.
             Arguments: <poolId> <poolType>
        -->
      <handlerPoolEmpty>/opt/handlers/notifyEmpty.sh</handlerPoolEmpty>
  </defaultConfiguration>

  <rules>
        <!-- some rules ........ -->
  </rules>

  <pools>
        <!-- Pool can overwrite any of the global configuration options -->
    <pool id="admins-pool" type="gid" key="dn" precreated="true">
      <configuration>
         <!-- disable automatic freezing for this pool -->
                <automaticFreezeAfter>-1</automaticFreezeAfter>
      </configuration>
      <id>admin_grp[1-100]</id>
```

(continues on next page)

(continued from previous page)

```
      <id>admin_grp[200-1000]</id>
    </pool>
  </pools>
</dynamicAttributes>
```

5.4.1.3.6 Starting the XUUDB server

Start the server with

```
$ BIN/start.sh
```

In case if XUUDB was installed with binary package use:

```
$ /etc/init.d/unicore-xuudb start
```

5.4.1.3.7 Stopping the server

Stop the server with

```
$ BIN/stop.sh
```

This sends a TERM signal to the XUUDB process. Please do not use `kill -9` to stop XUUDB, to avoid corrupting the database.

In case if XUUDB was installed with binary package use:

```
$ /etc/init.d/unicore-xuudb stop
```

5.4.1.3.8 Logging

UNICORE uses the Log4j logging framework. It is configured using a config file. By default, this file is found in components configuration directory and is named `logging.properties`. The config file is specified with a Java property `log4j.configuration` (which is set in startup script).

Several libraries used by UNICORE also use the Java utils logging facility (the output is two-lines per log entry). For convenience its configuration is also controlled in the same `logging.properties` file and is directed to the same destination as the main Log4j output.

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 *LOGS* directory.

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

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

# A1 is set to be a rolling file appender with default params
```

(continues on next page)

(continued from previous page)

```
log4j.appender.A1=org.apache.log4j.DailyRollingFileAppender
log4j.appender.A1.File=logs/uas.log

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

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

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

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

- PatternLayout
- RollingFileAppender
- DailyRollingFileAppender

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

Logger categories, names and levels

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

Therefore, the logging output produced can be controlled in a fine-grained manner. Log levels in Log4j are (in increasing level of severity):

- `TRACE` on this level huge pieces of unprocessed information are dumped
- `DEBUG` on this level UNICORE logs (hopefully) admin-friendly, verbose information, useful for hunting problems
- `INFO` standard information, not much output
- `WARN` warnings are logged when something went wrong (so it should be investigated), but recovery was possible
- `ERROR` something went wrong and operation probably failed
- `FATAL` something went really wrong - this is used very rarely for critical situations like server failure

For example, to debug a security problem in the UNICORE security layer, you can set:

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

If you are just interested in details of credentials handling, but not everything related to security you can use the following:

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

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

Note: You could turn on the general unicore `DEBUG` logging for a while (so the full category is printed). Then interesting events can be seen and subsequently the logging configuration can be fine tuned to only show them.

Several logging categories common in XUUDB:

Log category	Description
unicore	All of UNICORE
unicore.security	Security layer
unicore.client	Client calls (to other servers)
unicore.xuudb	XUUDB related
unicore.xuudb.server	XUUDB server
unicore.xuudb.server.db	XUUDB server database layer
unicore.xuudb.client	XUUDB admin client

5.4.1.4 The admin client

The admin client is used to edit the XUUDB, using a web service interface. It is configured in the file `CONF/xuudb_client.conf`. Client is invoked using the following pattern:

```
$ ADMIN <command> <options>
```

You can get detailed usage info by calling the admin script without any options. As it was noted above the actual utility path is dependent on how XUUDB was installed: it is either `/usr/sbin/unicore-xuudb-admin` or `INST/bin/admin.sh`.

Note: To switch on the confirmation message asked by the `add` command, edit the `admin.sh` script, so that the `xuudb.batch` property is set to `false`.

The client configuration requires the URL of the XUUDB server in the property `xuudb.address` and in case of secure HTTPS connections also a configuration truststore and credential. The settings are exactly the same as in case of the XUUDB server, so refer to its documentation *Configuring PKI trust settings*.

5.4.1.4.1 Configuring advanced HTTP client settings

UNICORE client stack can be configured with several advanced options. In most cases you can skip this section as defaults are fine.

The following table lists all available options. A special note for the `http.*` properties: those are passed to the Apache Commons HTTP Client library. Therefore, it is possible to configure all relevant options of the client. The options are listed under this address: https://hc.apache.org/httpclient-legacy/preference-api.html. Also see the *example* below.

Property name	Type	Default value / mandatory	Description
xuudb.client.digitalSigningEnabled	[true, false]	true	Controls whether signing of key web service requests should be performed.
xuudb.client.httpAuthnEnabled	[true, false]	false	Whether HTTP basic authentication should be used.
xuudb.client.httpPassword	string	*empty string*	Password for use with HTTP basic authentication (if enabled).
xuudb.client.httpUser	string	*empty string*	Username for use with HTTP basic authentication (if enabled).
xuudb.client.maxWsCallRetries	integer number	3	Controls how many times the client should try to call a failing web service. Note that only the transient failure reasons cause the retry. Note that value of 0 enables unlimited number of retries, while value of 1 means that only one call is tried.
xuudb.client.messageLogging	[true, false]	false	Controls whether messages should be logged (at INFO level).
xuudb.client.securitySessions	[true, false]	true	Controls whether security sessions should be enabled.
xuudb.client.serverHostnameChecking	[NONE, WARN, FAIL]	WARN	Controls whether server's hostname should be checked for matching its certificate subject. This verification prevents man-in-the-middle attacks. If enabled WARN will only print warning in log, FAIL will close the connection.
xuudb.client.sslAuthnEnabled	[true, false]	true	Controls whether SSL authentication of the client should be performed.
xuudb.client.sslEnabled	[true, false]	true	Controls whether the SSL/TLS connection mode is enabled.
xuudb.client.wsCallRetryDelay	integer number	10000	Amount of milliseconds to wait before retry of a failed web service call.
xuudb.client.http.allow-chunking	[true, false]	true	If set to false, then the client will not use HTTP 1.1 data chunking.
xuudb.client.http.connection-close	[true, false]	false	If set to true then the client will send connection close header, so the server will close the socket.
xuudb.client.http.connection.timeout	integer number	20000	Timeout for the connection establishing (ms)
xuudb.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.
xuudb.client.http.maxRedirects	integer number	3	Maximum number of allowed HTTP redirects.
xuudb.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.
xuudb.client.http.socket.timeout	integer number	0	Socket timeout (ms)
xuudb.client.http.nonProxyHosts	string		Space (single) separated list of hosts, for which the HTTP proxy should not be used.
xuudb.client.http.proxy.password	string		Relevant only when using HTTP proxy: defines password for authentication to the proxy.
xuudb.client.http.proxy.user	string		Relevant only when using HTTP proxy: defines username for authentication to the proxy.
xuudb.client.http.proxyHost	string		If set then the HTTP proxy will be used, with this hostname.
xuudb.client.http.proxyPort	integer number		HTTP proxy port. If not defined then system property is consulted, and as a final fallback 80 is used.
xuudb.client.http.proxyType	string	HTTP	HTTP proxy type: HTTP or SOCKS.

Example

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

Here we are setting an extremely short connection and socket timeouts for the clients calls, using the Apache HTTP client parameters. Additionally server hostname to certificate subject name checking is set to cause connections failures, preventing man in the middle attacks.

```
client.http.connection.timeout=2000
client.http.socket.timeout=2000
client.serverHostnameChecking=FAIL
```

5.4.1.4.2 Commands

The help with examples is provided for each command. You can use `helpAll` to print a reference documentation for all commands. Selected commands are also described below.

```
[classic]
  add
  adddn
  check-cert (chc)
  check-dn (chdn)
  export
  import
  list
  remove
  update

[dynamic]
  findMapping (fm)
  findReverseMapping (fr)
  freezeMappings
  getDynamicAttributes (getDyn)
  listMappings (lm)
  listPools (lp)
  removeMappings
  removePool (rmp)
  simulate (sim)

[other]
  help
  helpAll
```

Common options:

gcID The so-called *grid component ID* is used to group entries, and must match the setting in the UNICORE/X configuration file `uas.config`. For example, if you have two systems with different user name mappings, you can handle both with a single XUUDB, since you can store two user name mappings for each certificate, by choosing a different gcID for both systems. When updating xuudb entries, the special gcid * can be used as wildcard for updating user entries on all systems.

pemfile A file containing a public key in PEM format .

DN The distinguished name of a user.

xlogin xlogins (from UNIX login) are used for incarnation. Grid user's request which results in invocation of operations on a target system (usually through BSS) must be mapped to a local UNIX user. This attribute specifies the XLogins which are valid for the user. The first one is also used as a default one, if user does not request a particular one. Multiple logins can be specified using a `:`.

project Defines a primary group UNIX group for a user. If it is undefined then a default group for the XLogin is used.

role The usual roles in UNICORE are `user` for a normal user, and `admin` for an administrator. Custom roles can be added, and can be assigned permissions in the UNICORE/X security policy file.

5.4.1.4.3 Adding entries using `add` or `adddn`

Example using the DN:

```
$ ADMIN adddn DEMO-SITE "CN=John Doe, O=Test Inc" userlogin user
```

Example using a pem file:

```
$ ADMIN add DEMO-SITE /path/to/usercert.pem userlogin user
```

Note: When extracting the DN from a certificate file using OpenSSL, make sure to use the `RFC2253` option, for example,

```
$ openssl x509 -in demouser.pem -noout -subject -nameopt RFC2253
```

5.4.1.4.4 Checking the content

Apart from `list`, you can use the `check-cert` and `check-dn` commands to see what the XUUDB contains for a certain certificate or DN.

5.4.1.4.5 Removing entries

To remove all entries from xuudb (you will have to confirm this)

```
$ ADMIN remove ALL
```

To remove some entries, you have to specify attributes.

To remove a user with cert *cert.pem* at gcid *MYSITE*:

```
$ ADMIN remove gcid=MYSITE pemfile=/path/cert.pem
```

To remove all users from gcid *OLDMACHINE*:

```
$ ADMIN remove gcid=OLDMACHINE
```

To remove a user with xlogin *jdoe* from all gcids:

```
$ ADMIN remove xlogin=jdoe
```

5.4.1.4.6 Exporting/importing

The `export` command creates a csv file, which will contain the complete XUUDB database:

```
$ ADMIN export uudb.csv
```

If the file already exists, the export tool will complain. To override this, please specify the `overwrite` option, e.g.

```
$ ADMIN export uudb.csv overwrite
```

The `import` command takes the a csv file (as generated by `export`) and imports all entries. Already existing entries will not be changed. To do updates, execute `admin.sh remove ALL` before, or specify `clearDB` as a second argument:

```
$ ADMIN import uudb.csv
```

5.4.1.4.7 Updating entries

The `update` command can be used to modify existing entries, for example to replace the certificate or the login. For example,

```
$ ADMIN update DEMO-SITE certs/demouser.pem xlogin=jb007
```

would update the entry identified by the gcID *DEMO-SITE* and the given pem file, and assign a new xlogin. If you want to update a user's entry on all the sites, you would use:

```
$ ADMIN update \* certs/demouser.pem xlogin=jb007
```

Note: The wildcard * is a special character for the shell and needs to be escaped with a backslash.

5.5 Workflow Service

The Workflow service is a server component that supports submission and execution of application workflows consisting of UNICORE jobs and control constructs:

- *RESTful APIs*
- *JSON workflow description*
- Full range of UNICORE user authentication options and AAI integration

Workflow Service Manual User Manual with detailed instructions and examples for using the Workflow service.

5.5.1 Workflow Service Manual

The UNICORE Workflow service provides advanced workflow processing capabilities using UNICORE resources. The Workflow service provides graphs of activities including high-level control constructs (*for-each*, *while*, *if-then-else*, etc), and submits and manages the execution of single UNICORE jobs.

The Workflow service offers a *REST API* for workflow submission and management and uses an easy-to-understand *workflow description* syntax in JSON format.

Thanks to a flexible internal workflow model and execution engine, the Workflow service can be in principle extended with custom workflow parsers and custom activities.

The Workflow service supports the full range of authentication options provided by UNICORE and uses JWT tokens for delegated authentication when submitting jobs to the execution sites.

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

5.5.1.1 Installing and setting up the UNICORE Workflow engine

This chapter covers basic installation of the Workflow engine and the integration of the workflow services into an existing UNICORE system.

As a general note, the Workflow engine is based on a *UNICORE/X* instance. General UNICORE configuration concepts (such as user authentication, gateway integration, shared registry, attribute sources) fully apply, and you should refer to the *UNICORE/X manual* for such details.

5.5.1.1.1 Prerequisites

- Java 11 or later
- An existing UNICORE installation with *Gateway*, Shared *Registry* and one ore more *UNICORE/X* execution systems.
- A server certificate (for production use)

5.5.1.1.2 Updating from previous versions

If you update from 7.x, please note that it is a major update, and we suggest installing from scratch based on the template config files. The required changes are very similar to the UNICORE/X 7.x to 8.x update.

5.5.1.1.3 Installation

The workflow system is available either as part of the UNICORE Server bundle (tar.gz or zip), or as separate Linux packages (deb or rpm) on the UNICORE project website at sourceforge.

The basic installation procedure is completely analogous to the installation of a *UNICORE/X* server.

- If you downloaded the UNICORE server bundle, please untar the *tar.gz*, review the `configure.properties` file and edit the parameters to integrate the workflow services into your existing UNICORE environment. Then call `./configure.py` to apply your settings to the configuration files. Finally use `./install.py` to install the workflow server files to the selected installation directory.
- If using the Linux packages, simply install using the package manager of your system and review the config files.

5.5.1.1.4 Setup

After installation, there are some manual steps needed to integrate the new server into your UNICORE installation.

- *Gateway*: edit `gateway/conf/connections.properties` and add the connection data for the Workflow server. For example,

  ```
  WORKFLOW = https://localhost:7700
  ```

- *XUUDB*: if you chose to use an XUUDB for the Workflow server, you might have to add entries to the XUUDB to allow users access to the workflow engine. Optionally, you can edit the GCID used by the workflow/servorch servers, so that existing entries in the XUUDB will match.
- *Registry*: if the registry is setup to use access control (which is the default), you need to allow the Workflow server to register in the Registry. The exact procedure depends on how you configured your Registry, please refer to the section *Access control* in the *Registry manual*. If you're using default certificates and the XUUDB, the required entries can be added as follows:

  ```
  $ cd xuudb
  $ bin/admin.sh add REGISTRY <workflow>/conf/workflow.pem nobody server
  ```

5.5.1.1.5 Verifying the installation

Using the *UNICORE commandline client*, you can check whether the new server is available and accessible:

```
$ ucc system-info -l
```

should include output such as

```
Checking for <Workflow submission> endpoint ...
... OK, found 1 endpoint(s)
 * https://localhost:8080/WORKFLOW/rest/workflows
  * server v8.0.0 CN=Demo Workflow,O=UNICORE,C=EU
  * authenticated as: 'CN=Demo User, O=UNICORE, C=EU' role='user'
```

The "*authenticated as:*" line should list you as *user*.

Some more info about the server can be obtained via

```
$ ucc rest get https://localhost:8080/WORKFLOW/rest/workflows
```

Running a test job

Using *UCC* again, you can submit workflows

```
$ ucc workflow-submit /path/to/samples/date1.json
```

and get the ID of your new workflow back, e.g.

```
https://localhost:8080/WORKFLOW/rest/workflows/86686f72-b732-42e8-b14d-a8bd514e7edf
```

5.5.1.1.6 API documentation

Since version 8.0, the Workflow engine exclusively uses a *RESTful API* for all operations including *job submission*.

You can find an API reference and usage examples in *Workflow description*.

5.5.1.2 Configuration of the Workflow server

This chapter covers configuration options for the Workflow server. Since the Workflow server is running in the same underlying environment (**U**NICORE **S**ervices **E**nvironment, USE), a lot of the basic configuration options are documented in the *UNICORE/X manual*.

Note: The configuration files in the distribution are commented, and contain example settings for all the options listed here.

Depending on how you installed the server, the files are located on:

- `/etc/unicore/workflow` (Linux package)
- `<basedir>/workflow/conf` (standalone installer)

5.5.1.2.1 Workflow processing

Some details of the workflow engine's behaviour can be configured. All these settings are made in `uas.config`.

Limits

To avoid too many tasks submitted (possibly erroneously) from a workflow, various limits can be set:

- `workflow.maxActivitiesPerGroup` limits the total number of tasks submitted for a single group (i.e. (sub-)workflow). By default, this limit is `1000`, ie. a maximum number of 1000 jobs can be created by a single group. Note, that it is not possible to limit the total number of jobs for any workflow, it can only be applied to individual parts of the workflow (such as loops).
- `workflow.forEachMaxConcurrentActivities` limits the maximum number of tasks in a for-each group that can be active at the same time (default: `20`).

Resubmission

The workflow engine will (in some cases) resubmit failed tasks to the service orchestrator. To completely switch off the resubmission,

```
workflow.resubmitDisable=true
```

To change the maximum number of resubmissions from the default `3`,

```
workflow.resubmitLimit=3
```

Cleanup behaviour

This controls the behaviour when a workflow is removed (automatically or by the user). By default, the workflow engine will remove all child jobs, but will keep the storage where the files are. This can be controlled using two properties:

- `workflow.cleanupStorage` remove storage when workflow is destroyed (default: `false`)
- `workflow.cleanupJobs` remove jobs when workflow is destroyed (default: `true`)

5.5.1.2.2 XNJS settings

The workflow engine uses the XNJS library for processing workflows. Some settings for modifying the behaviour are available, and are usually found in the workflow server's `container.properties` file.

An important characteristic is the number of threads used by the workflow engine for processing. Note, this does not control the number of concurrent activities, etc., since all XNJS processing is asynchronous. The default number (4) is usually fine.

This properties is set via

```
XNJS.numberofworkers=4
```

5.5.1.2.3 Property reference

A complete reference of the properties for configuring the Workflow server is given in the following table:

Property name	Type	Default value / mandatory	Description
workflow.additionalSettings	filesystem path		Optional file containing additional settings (e.g. XNJS.* settings) only used for the workflow service.
workflow.cleanupJobs	[true, false]	true	Whether to remove child jobs when the workflow is destroyed.
workflow.cleanupStorage	[true, false]	false	Whether to cleanup the workflow storage when the workflow is destroyed.
workflow.fastPollingInterval	integer >= 1	20	Interval in seconds for (slow) polling of job states.
workflow.forEachConcurrentActivities	integer >= 1	100	Default maximum number of concurrent for-each iterations (user can increase this).
workflow.forEachMaxConcurrentActivities	integer >= 1	200	Hard limit on the number of concurrent for-each iterations.
workflow.internalMode	[true, false]	false	Internal mode: Workflow service only uses services deployed in the same UNICORE/X instance.
workflow.maxActivitiesPerGroup	integer >= 1	1000	Maximum number of workflow activities per activity group.
workflow.pollingInterval	integer >= 1	600	Interval in seconds for (slow) polling of job states.
workflow.resubmitDisable	[true, false]	false	Whether to disable automatic re-submission of failed jobs.
workflow.resubmitLimit	integer >= 1	3	Maximum number of re-submissions of failed jobs.
workflow.xnjsConfiguration	string	n/a	(deprecated)

5.5.1.3 Updating an existing UNICORE Workflow service

This chapter covers the steps required to update an existing workflow installation (version 8.x).

5.5.1.3.1 Stop the server

Stop the workflow server.

5.5.1.3.2 Backup

You should make a backup of your existing data and, if necessary, your config files.

5.5.1.3.3 Update jar files

The Java libraries have to be replaced with the new versions.

5.5.1.3.4 Update config files

Compare the config files from the new version to your existing one. Check the changelog for new features that might require updates to config files.

5.5.1.3.5 Restart the servers

Restart the workflow server, and check the logs for any suspicious error messages!

5.6 Registry

The Registry server provides information about available services to clients and other services. It is a specially configured *UNICORE/X* server, so please make sure to read the general *UNICORE/X manual* as well.

Multiple UNICORE/X sites can share a Registry, greatly simplifying the use of UNICORE services. Since such a registry is vital to the functioning of a UNICORE-based federation, you can have more than one.

5.6.1 Installation

5.6.1.1 Prerequisites

To run the Registry, you need a Java runtime (headless is enough), in version 11 or later.

UNICORE servers have been most extensively tested on Linux systems, but run on MacOS/X as well.

Please note that

- to integrate into secure production environments, you will need access to a certificate authority and generate server certificates for all your UNICORE servers.
- to make your UNICORE servers accessible outside of your firewalls, you should setup and configure a *UNICORE Gateway*.

5.6.1.2 A note on paths

The Registry can be installed either from the UNICORE Server bundle (tar.gz or zip archive) or from a Linux package on the UNICORE project website at sourceforge (i.e. RPM or deb).

Attention: Using the Linux packages, you can install only a single Registry instance per machine (without manual changes).

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

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

5.6.2 Registry configuration

A Registry is running in a *normal* *UNICORE/X* container, however, you should use a dedicated UNICORE/X instance for the Registry, making sure no other services are running.

Thus, most of the UNICORE/X documentation regarding access control, keystores, etc also applies to the Registry. Please, make sure to read the *UNICORE/X documentation* as well.

5.6.2.1 Registry configuration (`CONF/uas.config`)

Apart from hostname, port, and other properties, the `uas.config` file must contain the following entry:

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

This setting configures the container to operate as a shared Registry.

5.6.2.2 Starting and stopping

The Registry is started and stopped like any other *UNICORE/X* container using the scripts in the `bin` folder.

5.6.2.3 Access control

It is absolutely **VITAL** that the Registry only contains **trusted entries**. Therefore the default access control policies (`CONF/xacml2Policies/*.xml`) only allow to add entries only for callers with the role *server*.

You will need to map the certificates / DNs of all servers wishing to publish into the registry as having the role *server*. Please check the *UNICORE/X documentation* on how to do that, using an XUUDB or other attribute source.

5.6.2.4 User / server authentication

While users can read registry content without needing to be authenticated, servers **MUST** be authenticated and mapped to role *server* to be able to write to the Registry.

To accept servers, the REST interface must be configured for X509 authentication.

As an example the following configuration will achieve this:

```
#
# Authentication for the REST interface
#
container.security.rest.authentication.order=X509
container.security.rest.authentication.X509.class=eu.unicore.services.rest.security.
↪X509Authenticator
```

For further details we refer also to the *UNICORE/X documentation* on authentication and REST services.

5.6.2.5 Gateway configuration

If running the Registry behind a *Gateway*, you'll need to add an entry to the Gateway's site list file (`connections.properties`) that points to your Registry server. Another option is to use dynamic registration. In the following, we assume the Registry is named *REGISTRY*.

5.6.2.6 UNICORE/X configuration

To publish the services in a shared registry, configure the address of the registry in `uas.config`:

```
# switch on use of external registry
container.externalregistry.use=true

# URL
container.externalregistry.url=https://...

# optionally you can have more registries
container.externalregistry.url.2=https://...
```

The entries in the global Registry are updated at a specified interval. To control this interval, edit a property in `CONF/container.properties`:

```
# default termination time for registry entries in seconds
container.wsrf.sg.defaulttermtime=1800
```

5.6.2.7 Client configuration

Clients will require the URL of a Registry. For example, in the *UCC* preferences file (supply the correct values for your setup):

```
registry=https://gwhost:port/REGISTRY/rest/registries/default_registry
```

CHAPTER

SIX

GETTING SUPPORT

For getting more information or support, please see the *Links* and *Support* page.

6.1 Links

GitHub	UNICORE on GitHub
SOURCEFORGE	UNICORE on SourceForge
YouTube	UNICORE on YouTube
	UNICORE on Twitter

6.2 Support

If you encounter any issues with the software, please consider opening a ticket on the general issue tracker https://sourceforge.net/p/unicore/issues.

In case you have direct questions related to the UNICORE software, please use the UNICORE support mailing lists at https://sourceforge.net/p/unicore/mailman.

Please note also the following places for getting more information:

UNICORE **Project Website** https://www.unicore.eu

Support list unicore-support@lists.sf.net

Developer's list unicore-devel@lists.sf.net (*needs registration*)

Source code https://github.com/UNICORE-EU

CHAPTER

SEVEN

LICENSE

UNICORE software is available as Open Source under the *BSD License* while the software repository is hosted on SourceForge and the source code is available on GitHub.

7.1 License

```
BSD 2-Clause License

Copyright (c) 1997-2023
Forschungszentrum Jülich GmbH, Fujitsu Labs Europe, ICM Warsaw,
Intel Corporation, CINECA, University of Manchester, T-Systems,
and other contributors to UNICORE: https://www.unicore.eu

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```