UNICORE COMMANDLINE CLIENT: USER MANUAL

UNICORE Team

Document Version:	1.0.0
Component Version:	7.11.0
Date:	17 09 2018

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

Contents

1 Overview 1

2 Installation and configuration 1

2.1 Prerequisites 1

2.2 Download 1

2.3 Installation and configuration 2

2.4 Preferences file 2

2.5 Logging 3

2.6 Installing UCC extensions 3

2.7 Testing the installation 3

3 Getting started with UCC 3

3.1 Getting help 4

3.2 Connecting 4

3.3 List available sites 4

3.4 Running your first job 4

3.5 Listing your jobs 5

4 Common options to UCC 5

4.1 User attributes and VOs 6

4.2 Configuration file 6

4.3 Credential and truststore options 7

4.4 Trust store examples 12

4.5 Using Unity 13

4.6 Support for oidc-agent 13

4.7 Using MyProxy 14

4.8 Client options 15

4.9 Other options 17

5 SAML PUSH support **18**
5.1 Introduction . . . 18
5.2 Basic usage . . . 18
5.3 Attribute filtering . . . 18
5.4 Rules for multiple filters . . . 19

6 Running jobs **20**
6.1 Introduction . . . 20
6.2 Options overview . . . 21
6.3 Resource brokering . . . 22
6.4 Processing jobs asynchronously . . . 22
6.5 Executing a command . . . 24

7 Job description format **25**
7.1 Site name . . . 25
7.2 Specifying the application or executable . . . 26
7.3 Arguments and Environment settings . . . 26
7.4 Application parameters . . . 27
7.5 Job data management . . . 28
7.6 Resources . . . 31
7.7 Execution environments . . . 32
7.8 Miscellaneous options . . . 33

8 Data management functions **34**
8.1 Specifying remote locations . . . 34
8.2 Data movement . . . 35
8.3 General commands . . . 37
8.4 Finding data . . . 38
8.5 Deprecated commands . . . 39
8.6 Using the StorageFactory service . . . 39

9 Metadata management functions **39**
9.1 Basics . . . 40
9.2 Available commands . . . 40

10 Workflows **42**
10.1 Introduction . . . 42
10.2 Command overview . . . 42
10.3 Basic use . . . 42
10.4 Managing workflow data . . . 43

11 Batch processing **46**
11.1 Options . . . 46
11.2 Performance tuning options . . . 46
11.3 Resource selection in batch mode . . . 47

12 OGSA-BES functions **48**
12.1 OGSA-BES Setup . . . 48
12.2 Running and monitoring OGSA-BES jobs . . . 49
12.3 Get OGSA-BES job outputs . . . 50
12.4 Enabling username/password authentication . . . 50

13 The UCC shell **50**
13.1 Exiting the shell . . . 51
13.2 Changing property settings . . . 51

14 Sharing resources **51**
14.1 Editing ACLs . . . 52
14.2 Permission levels . . . 52

15 UCC for site administrators **52**
15.1 Security considerations . . . 53
15.2 Admin commands . . . 53
15.3 Listing jobs, sites, ... 56
15.4 Low-level operations . . . 57

16 Scripting **57**
16.1 Script context . . . 57
16.2 Examples . . . 58

17 Frequently asked questions **60**

17.1 Configuration 60

17.2 Usage 61

Overview

The UNICORE Commandline client (UCC) is a full-featured client for the UNICORE middleware. UCC has client commands for all the UNICORE basic services, the OGSA-BES interface and the UNICORE workflow system.

It offers the following functions

- Job submission and management for both UNICORE native and OGSA-BES interfaces
- 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
- Full UNICORE workflow system support, including the possibility to run single jobs through the resource brokering system
- 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, the ability to generate SAML trust delegations, low-level WSRF 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 http://www.unicore.eu.

Installation and configuration

Prerequisites

To run UCC, you need a Java runtime version 8 or later (OpenJDK, Oracle or IBM). You might need to install the "unlimited cryptography extensions" for the Oracle or IBM Java.

Download

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

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 your keystore location and other parameters on the commandline, it is easiest to place this information in a file, so that you do not have to key in this information repeatedly.

Preferences file

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

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

```
credential.path=<your keystore>
credential.password=<your password>
truststore.type=keystore
truststore.keystorePath=<your keystore>
truststore.keystorePassword=<your password>
client.serverHostnameChecking=NONE
registry=<your registry>
```

Please refer to Section 4 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/services/Registry?res= ↩
   default_registry
```

Note
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> ....
```

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.

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.

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
```

Getting started with UCC

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

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 here for a list of common options.

Connecting

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

```
ucc connect [options]
```

List available sites

Then, list the sites available to you using

```
ucc list-sites [options]
```

Running your first job

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

```
ucc run [options] -v [UCC_HOME]/samples/date.u
```

Note
Look for UCC samples in the /usr/share/doc/unicore/ucc/samples directory,

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.

Listing your jobs

The command

```
ucc list-jobs [options]
```

will print a list of jobs (actually their addresses) with their respective status (RUNNING, SUCCESSFUL, etc)

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 batch --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 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 *userhome/.ucc/preferences* is checked.
`-k,--authenticationMethod <auth>`	Authentication method to use (default: X509)
`-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

User attributes and VOs

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. Also, when using UCC with infrastructures that run a SAML-enabled VO server such as UVOS, there are additional options for specifying the VO server, your VO etc. Please see [?] section for details about the VO support.

Here is a list of options related to user attributes and VOs.

Table 2: Security and VO options for the UCC

Option (short and long form)	**Description**
`-U, --user`	User ID to use remotely (if you have multiple)
`-Z, --preference`	Select from your remote attributes (e.g. xlogin)
`-J, --VO`	VO server URL
`-G, --voGroup`	VO group
`-A, --attributeAssertion`	File containing a VO attribute assertion
`-I, --includeAttributes`	Filter VO attributes
`-Q, --excludeAttributes`	Filter VO attributes

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.

For example, to set your keystore, truststore and registry, the file would contain the following settings

```
credential.path=<your keystore>
credential.password=XXXXXXX
truststore.type=keystore
truststore.keystorePath=<your keystore>
truststore.keystorePassword=XXXXXX
registry=https://localhost:8080/DEMO-SITE/services/Registry?res= ↩
    default_registry
```

Note
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 the credential and/or truststore passwords are not given in the properties file, they will be queried interactively.

Credential and truststore options

In general you need a keystore containing your identity in order to use UNICORE, as well as a truststore file (or directory) containing trusted certificates. (Note that there may be other options available for authentication, try *ucc help-auth* to find out)

A full list of options related to credential and truststore management is available in the following table. You can also get them via the online help using

```
ucc help-auth
```

Table 3: Credential properties

Property name	**Type**	**Default value / mandatory**	**Description**
`credential.path`	filesystem path	*mandatory to be set*	Credential location. In case of *jks*, *pkcs12* and *pem* store it is the only location required. In case when credential is provided in two files, it is the certificate file path.

Table 3: (continued)

Property name	**Type**	**Default value / mandatory**	**Description**
`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).
`credential.password`	string	-	Password required to load the credential.
`credential.keyPath`	string	-	Location of the private key if stored separately from the main credential (applicable for *pem* and *der* types only),
`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.
`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*.

Table 4: Truststore properties

Property name	**Type**	**Default value / mandatory**	**Description**
`truststore.allowProxy`	[ALLOW, DENY]	`ALLOW`	Controls whether proxy certificates are supported.
`truststore.type`	[keystore, openssl, directory]	*mandatory to be set*	The truststore type.

Table 4: (continued)

Property name	**Type**	**Default value / mandatory**	**Description**
`truststore.updateInterval`	integer number	`600`	How often the truststore should be reloaded, in seconds. Set to negative value to disable refreshing at runtime. *(runtime updateable)*
--- Directory type settings ---			
`truststore.directoryConnectionTimeout`	integer number	`15`	Connection timeout for fetching the remote CA certificates in seconds.
`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.
`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.
`truststore.directoryLocations.*`	list of properties with a common prefix	-	List of CA certificates locations. Can contain URLs, local files and wildcard expressions. *(runtime updateable)*
--- Keystore type settings ---			
`truststore.keystoreFormat`	string	-	The keystore type (jks, pkcs12) in case of truststore of keystore type.
`truststore.keystorePassword`	string	-	The password of the keystore type truststore.
`truststore.keystorePath`	string	-	The keystore path in case of truststore of keystore type.

Table 4: (continued)

Property name	**Type**	**Default value / mandatory**	**Description**
--- Openssl type settings ---			
`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)
`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).
`truststore.opensslPath`	filesystem path	`/etc/grid-security/certificates`	Directory to be used for opeenssl truststore.
--- Revocation settings ---			
`truststore.crlConnectionTimeout`	integer number	`15`	Connection timeout for fetching the remote CRLs in seconds (not used for Openssl truststores).

Table 4: (continued)

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

Table 4: (continued)

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

Trust store examples

Here are some examples for commonly used trust store configurations.

Directory trust store with a minimal set of options

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

Directory trust store with more 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
```

Java keystore used as a trust store:

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

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
```

Using Unity

If your UNICORE installation is using the Unity identity management service (see http://www.unity-idm.eu), you can setup UCC to use Unity. In this case you do not need a private key, only a truststore. UCC is configured for Unity using the following properties

```
authenticationMethod=unity
unity.address=https://<host>:<port>/unicore-soapidp/saml2unicoreidp ↩
    -soap/AuthenticationService
unity.username=<your Unity username>
unity.password=<your Unity password>
```

At minimum, you must specify the "authenticationMethod" and "unity.address" parameters. Password and username are optional: if you do not specify them, they will be queried interactively.

In some installations, you can use a so-called OIDC Bearer token instead of username and password. In this case specify only unity.address and the token, like this:

```
authenticationMethod=unity
unity.address=https://<host>:<port>/unicore-soapidp/saml2unicoreidp ↩
    -soap-oidc/AuthenticationService
oauth2.bearerToken=<your oauth token>
```

Support for oidc-agent

For infrastructures using OIDC authentication, UCC supports the *oidc-agent* tool that allows to interact with common OIDC servers to retrieve new access tokens.

Please visit https://github.com/indigo-dc/oidc-agent for more information.

To use oidc-agent, UCC has the following properties

Table 5: Options for oidc-agent

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

Note that you'll also need to configure a Unity server that will accept the OIDC token. Your config file would require at least:

```
authenticationMethod=oidc-agent
unity.address=https://<host>:<port>/unicore-soapidp/saml2unicoreidp ↩
    -soap-oidc/AuthenticationService
oidc-agent.account=<oidc-agent account to be used>
```

Using MyProxy

UCC can retrieve a short lived certificate from a MyProxy server. To setup UCC for this, use the following properties

```
authenticationMethod=MYPROXY
myproxy.host=myproxy.teragrid.org
myproxy.port=7512
myproxy.username=<your MyProxy username>
myproxy.password=<your MyProxy password>
myproxy.expires=12
```

At minimum, you must specify the "authenticationMethod" and "myproxy.host" parameters. The default port of 7512 will be used if not otherwise specified. Password and username are optional: if you do not specify them, they will be queried interactively. The lifetime (in hours) of the short lived certificate can be set with the "myproxy.expires" option, if not set the lifetime is 12 hours.

Client options

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

Table 6: 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.inHandlers`	string	*empty string*	Space separated list of additional handler class names for handling incoming WS messages
`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.outHandlers`	string	*empty string*	Space separated list of additional handler class names for handling outgoing WS messages
`client.securitySessions`	[true, false]	`true`	Controls whether security sessions should be enabled.

Table 6: (continued)

Property name	Type	Default value / mandatory	Description
`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.
		--- HTTP client settings ---	
`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)

Table 6: (continued)

Property name	**Type**	**Default value / mandatory**	**Description**
		--- HTTP proxy settings ---	
`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.

Other options

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

Table 7: Other options for the UCC

Property name	**Description**
`contact-registry`	Do not attempt to contact the registry, even if one is configured

SAML PUSH support

Introduction

UCC supports the "SAML PUSH" mode of authentication. In this mode, the user is authenticated by an attribute assertion signed by a trusted third party. The third party is a Virtual Organisation (VO), SAML 2.0 server such as UVOS or VOMS.

Basic usage

First, retrieve an attribute assertion from VO server, scoped for a particular group you are in, and save it into a file.

```
ucc save-attributes -J https://uvos.example.com:2443 -G /vo/group - ↩
    O assertion.xml
```

If the VO server has been registered in the registry, you may use "auto" instead of server's URL (this is possible only in case of UVOS).

```
ucc save-attributes -J auto -G /vo/group -O assertion.xml
```

Then, you may use this assertion to get access to the server. Most UCC commands support this, for example you may connect to the server using this assertion.

```
ucc connect -A assertion.xml
```

You also may combine those two steps into one:

```
ucc connect -J https://uvos.example.com:2443 -G /vo/group
```

or assuming that UVOS server is in the registry:

```
ucc connect -J auto -G /vo/group
```

However note that using <<←A>>> option is faster: attributes are read from a file, while in the combined scenario the attributes are fetched from a remote VO server prior to invocation of the intended UCC command.

Attribute filtering

You might want to obtain an assertion with only handful of attributes and their values. This is done by using attribute filters.

UCC can display a list of all attributes. Names, scopes, values and descriptions (if available) will be displayed.

```
ucc list-attributes -J auto
```

The last column (F) contains a letter Y next to all attribute values, that pass through attribute filters. There are two kinds of filters: inclusive filters (which specify what attributes should pass) and exclusive filters (which specify what attributes should be filtered out). If not specified, the default inclusive filter approves of all attributes and their values, and the default exclusive filter does not reject anything.

In this example an inclusive filter choose only those attributes, whose name contains the word "xlogin".

```
ucc list-attributes -J auto -I ".*xlogin.*"
```

The value for the <<←I>>> option is a list of semicolon-separated Java regular expressions. The expressions that contain the equal sign are called name-value filters, those which do not are called name filters.

For example, you may choose all attributes with names that contain the word "login", and the "admin" value of all attributes containing the word "role" in their name.

```
ucc list-attributes -J auto -I ".*xlogin.*;.*role.*=admin"
```

You may also exclude attributes. For example, the following filter chooses all attributes but those ending with an "a".

```
ucc list-attributes -J auto -Q ".*a"
```

If you are content with the results of filtering, you can obtain an assertion only with the filtered attributes.

```
ucc save-attributes -J auto -G /vo/group -O assertion.xml -I ".* ↩
    xlogin.*;.*role.*=admin" -Q ".*a"
```

You can also use it directly.

```
ucc connect -J auto -G /vo/group -I ".*xlogin.*;.*role.*=admin" -Q  ↩
    ".*a"
```

Rules for multiple filters

Each attribute is considered as a set of name-value pairs, containing the name of attribute and the value of attribute.

A pair matches a name filter (i.e. a filter defined using a regular expression without equals sign) if the name of the attribute matches the regular expression.

A pair matches a name-value filter (i.e. a filter defined using a regular expression with equals sign) if the name of the attribute matches the left hand of the regular expression, and the value matches the right hand.

For each pair, the following steps are performed:

- if there is a matching exclusive name-value filter, the pair is rejected
- otherwise, if there is a matching inclusive name-value filter, the pair passes
- otherwise, if there is a matching exclusive name filter, the pair is rejected
- otherwise, if there is a matching inclusive name filter, or there are no inclusive name filters, the pair passes
- otherwise, the pair is rejected

The result of filtering are the attributes, containing only those values which passed (after filtering, empty attributes are discarded).

Running jobs

Introduction

The UCC can run jobs specified in a simple job description format Section 7 . In the following it is assumed that you have UCC installed Section 2 and tried some examples Section 3 .

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 many 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 Section 8 for details.

Options overview

The following options are available when running jobs (see also the general options overview in Section 4.

Table 8: 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 (LOCAL or SERVORCH)
`-d,--dryRun`	Only show candidate sites, but do not submit the job
`-s,--sitename <SITE>`	Site where the job shall be run
`-S,--schedule <Time>`	Schedule the submission of the job at the given time
`-j,--jsdl`	Tell UCC that the job file is a JSDL document
`-o,--output <Output_dir>`	Directory for any output produced (default is the current directory)
`-O,--stdout <stdout_name>`	specify a name for the exported standard out (by default: *stdout*)
`-E,--stderr <stderr_name>`	specify a name for the exported standard error (by default: *stderr*)

Resource brokering

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

There are two types of brokers available, which can be selected using the "-B" or "--broker" option.

- LOCAL : brokering is done by UCC itself
- SERVORCH (default) : brokering is done by the "service orchestrator" component of the UNICORE Workflow system.

The SERVORCH broker requires an accessible UNICORE workflow system in version 6.6.0 or later. If this is not available, UCC will fall back to the LOCAL broker. As usual, you can set this option in your UCC preferences file.

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. UCC supports this, of course. The basic idea is that when submitting a job in asynchronous mode, a job descriptor file is written that contains the job's address, and any information about export files.

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 descriptor file (ending in ".job") will be written to your configured output directory.

Get the status of particular jobs

The command

```
ucc job-status <job_desc> <job_desc_2> ...
```

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

Download results

To get stdout, stderr and other files you have marked for export in your job description, do

```
ucc get-output -o <outdir> <job_desc>
```

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

Referencing a job by its EPR (Endpoint reference)

In case you want to check on a job not submitted through UCC, or in case you do not have the job descriptor file any more, you can also refer to a job given its EPR. For example, the "list-jobs" command will produce a list of all job EPRs 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 datamovement functions described in Section 8.

Uploading and executing an executable

To upload and execute a file on a remote server, you might need a small helper script to make the uploaded file executable and run it:

```
#!/bin/sh
chmod +x myapp
./myapp
```

Your ucc job description would then look as follows

```
{
  Executable: "/bin/sh",
  Arguments: ["helper.sh"],
  Imports: [
    {From: "helper.sh", To: "helper.sh"},
    {From: "myapp", To: "myapp"},
  ],
}
```

Note
Recent servers do not require this trick, and you should be able to directly execute your uploaded executable without requiring a helper script.

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 she knows that a certain queue will be empty at that time.

Note
This feature only works with server release 6.4.0 or higher.

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" key word

```
{

 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",

}
```

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
```

Watch out to properly escape any arguments, in order not to interfere with the arguments to UCC.

Job description format

UCC uses a simple 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 or the local machine and any result files to stage out.

A number of sample files can be found in the "samples" directory of your UCC distribution. (on Linux, check also /usr/share/unicore/ucc/samples)

The format used is called JSON, and contains comma-separated key-value mappings, where the values can be simple strings, or lists of values, or maps. String values should be placed in "quotes". Comments are (inofficially) possible using the "#" hash character, as in Unix shell scrips.

Each JSON file must begin and end with curly braces "{ ... }". Several complete job samples can be found in the "samples" directory of the distribution.

Note
Note: quotes "" are needed around the keys and values in case special characters (like *:* or /") appear, if in doubt use quotes!

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

```
ucc run -H
```

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

Note
You may alternatively specify jobs in the JSDL format that is used internally in UNICORE. To do this, run UCC with the "-j" option.

Usually, a UCC job file describe a single batch job on the target system. However there is a UNICORE feature called "parameter sweep" which leads to the creation of multiple batch jobs from a single "template" job. UCC can also create these "sweep jobs", as described in the relevant parts of the job description. Note that a "sweep job" still is treated as a single job by UNICORE. Sweep jobs are very useful if you need to run jobs that are highly similar, and only differ by a parameter setting or even by a different input file.

Site name

You can (optionally) specify on which site (if available) the job should be run.

```
  Site: "DEMO-SITE",
```

If you do not specifiy anything UCC will select a site that will match your requirements (at least those that UCC checks for).

Specifying the application or executable

You can specify a UNICORE application by name and version, or using a (machine dependent) path to an executable file.

```
#using application name and version
{
   ApplicationName: "Date",
   ApplicationVersion: "1.0",
}
```

Note the comma-separation and the curly braces. To call an executable,

```
#using an executable

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

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", FOO: "bar"} ,

}
```

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.

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. In UCC, 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.

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.

Job data management

In general your job will require data files, either from your client machine, or from some remote location. An important concept in UNICORE is the job's workspace (also called *Uspace*, which is the default location into which files are placed. The same applies to result files: by default, files will be downloaded from the job's workspace.

However, other remote storage locations are supported, too.

The remote location can be given as a full UNICORE URI, or using the more user friendly (but slower) "u6://" notation. Read more on remote locations in Section 8.

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 files into the job workspace

To import files from your local computer or from remote sites to the job's working directory on the remote UNICORE server, there's the "Imports" keyword. Here is an example Import section that specifies three imports:

```
{

   Imports: [

   # import a local file into the job workspace
    { From: "/work/data/fileName", To: "uspaceFileName" },

   # import a set of PDF files into the Uspace
    { From: "/work/data/pdf/*.pdf", To: "/" },

   # import a remote file from a UNICORE storage
    { From: "u6://DEMO-SITE/Home/testfile", To: "otherUspaceFile"  ↩
        },
   ]

}
```

If for some reason it may happen that the local file does not exist, and you want the job to run anyway, there is a flag "FailOnError" that can be set to "false" :

```
   Imports: [
   # do not fail on errors for this import:
    { From: "/work/data/fileName", To: "uspaceFileName",  ↩
        FailOnError: "false", },
   ]
```

Recent UNICORE servers (7.3 and later) support optimized input data handling. When multiple jobs or workflows use the same input data files, it may be possible to just create a symbolic link instead of physically copying the file. The tell UNICORE that this is OK, you must set a "ReadOnly" flag on the import:

```
Imports: [
# tell UNICORE that the file can be sym-linked, if possible
 { From: "/work/data/fileName", To: "uspaceFileName", ReadOnly:  ↩
     "true", },
]
```

Note
UCC supports simple wild cards ("*" and "?") for importing exporting LOCAL files, and for remote files if the server is version 7.x or later. Wildcards do not work for server-to-server imports and exports on 6.x servers.

Importing files into other storage locations

A UNICORE site may supports multiple storages (for example, a TMP or SCRATCH directory). To instruct the server to stage-in a file into such a storage, the "Filesystem" tag may be used. For example to stage-in a file into SCRATCH space, the following Imports definition can be used:

```
Imports: [

# import a file from a remote storage into the SCRATCH space on  ↩
    the target resource
 { From: "u6://DEMO-SITE/Home/work/data/fileName", To: "fileName ↩
     ", Filesystem: "SCRATCH", },
```

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:

```
Imports: [

 { From: [ "u6://DEMO-SITE/Home/work/data/file1",
           "u6://DEMO-SITE/Home/work/data/file2",
           "u6://DEMO-SITE/Home/work/data/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.

Exporting result files from the job workspace

To export files from the job's working directory to your local machine or to some remote storage, use the "Exports" keyword. 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: "u6://DEMO-SITE/Home/results/myjob/stdout ←
        " },

   ]

}
```

As a special case, UCC also supports downloading files from other UNICORE storages using the Exports keyword:

```
{
   Exports: [
    #this exports a file from a UNICORE storage
    { From: "u6://DEMO-SITE/Work/somefile", To: "/home/me/somefile" ←
         },
   ]
}
```

The protocol to be used for imports and exports can be chosen using the "Preferred Protocols" entry, containing a space-separated list of protocols:

```
{

   Preferred protocols: "BFT RBYTEIO",

}
```

If not specified, BFT will be used.

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

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

and similarly for exports.

Servers 7.9.0 and later also support OAuth Bearer token for HTTP data transfers.

```
{
   Imports: [
     { From: "https://someserver/some/file", To: "input_data",
       Credentials: { BearerToken: "some_token" },
     },
   ]
}
```

You can leave the token value empty, `BearerToken:""`, if the server already has your token by some other means.

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.

Resources

A job definition can have a Resources section specifying the resources to request on the remote system. For example

```
  Resources: {

    # memory per node (bytes, you may use the common "K","M" or "G ↩
       ")
    Memory: 2G ,

    # walltime (seconds, use "min", "h", or "d" for other units)
    Runtime: 86400 ,

    # Total number of requested CPUs
    CPUs: 64 ,

    # you may optionally give the number of nodes
    #Nodes: 2 ,
    # together with the CPUs per node
    #CPUsPerNode: 32,

    # Custom resources (site-dependent!)
    StackLimitPerThread : 262144,

    # Operating system
    Operating system: LINUX,   #MACOS, WINNT, ...

    # Resource reservation reference
    Reservation: job1234,

  }
```

Note that you can also specify a reservation reference if your batch system supports this and you have made a resource reservation.

Execution environments

To run a job in a special execution environments (as supported by the server), you can use the following syntax.

```
  Execution environment: {
      Name: ...,
      Arguments: {
        ArgName1: "value1", ArgName2: "value2", ...
      },
      Options: [ ... ],
      Precommands: [ ... ],
      Postcommands: [ ... ],
      User precommand: "..." ,
      RunUserPrecommandOnLoginNode: "false",
      User postcommand: "..." ,
      RunUserPostcommandOnLoginNode: "true",
    },
```

The user pre and post commands can be run either separately on the login node (default) or be placed in the job script. This is achieved using the "RunUserPrecommandOnLoginNode" and "RunUserPostcommandOnLoginNode" directives, which can be set to true or false.

Miscellaneous options

Selecting the remote login and/or group

In case you have multiple logins or Unix groups on the remote site mapped to the same credential, you can select the user name and/or group to use as follows

```
  User name: yourlogin,
  Group: yourgroup,
```

Hint: you can get a list of your logins/groups on the site by executing

```
  ucc list-sites -s SITENAME -l
```

Specifying a 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" tag:

```
Project: "my_project",
```

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
```

Hint: if you want to explicitely switch off the email notification, use "NONE" as email value. This might be necessary because older UNICORE server versions try to use the email address from your certificate (if present).

Specifying the job name

The job name can be set simply by

```
  Name: Test job
```

Specifying the status check interval for batch mode

Once a job is started, it is often not useful to check its status every few seconds, because the job might be running several minutes or more. Especially in batch mode it can reduce the load on the servers if the update interval is chosen longer. This can be achieved by using the following setting (this only affects batch mode!):

```
  Update interval: 60,   #only check once a minute (default is one ↩
      second)
```

Specifying the "lifetime" of the job

If you want to specify a lifetime of the job, and not rely on the server default, you can use the lifetime attribute:

```
  Lifetime: 12h,   #sec, min, h, d
```

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.

Specifying remote locations

Remote locations can be specified in two ways. The first way is to use a URI that includes protocol, storage server and filename, for example

```
BFT:https://mygateway:8080/SITE/services/StorageManagement?res= ↩
    default_storage#/file
```

which specifies a file named "/file" on the storage instance "https://mygateway:8080/SITE/services/StorageManagement?res
using the BFT protocol.

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

This explicit format is sometimes inconvenient, so you can use a shorter, more intuitive format. This is also a URI, but you need to know only the name of the virtual site (target system), and the storage or job id. For example

```
unicore6://SITE/Home/file?protocol=PROTOCOL
```

or shorter

```
u6://SITE/Home/file?protocol=PROTOCOL
```

This will resolve the current user's "Home" storage at the target system named "SITE". Note that if you do not specify the protocol, the BFT protocol will be used as default.

You can also refer to a job's working directory on a given site. For this, you will need the unique ID of that job, which you can get for example using the *list-jobs* command. For example,

```
u6://SITE/1f3bc2e2-d814-406e-811d-e533f8f7a93b/outfile
```

refers to the file "outfile" in the working directory of the given job on the "SITE" target system. And

```
ucc ls -l u6://SITE/1f3bc2e2-d814-406e-811d-e533f8f7a93b/
```

will list the job’s working directory.

It is also possible to refer to storage services that are registered in the registry using their name, for example

```
u6://SHARE/myfiles/a_file
```

can be used to refer to the shared storage named "SHARE" if it is registered in the registry.

Though convenient, the method using "unicore6://" is much slower, and will generate some network traffic. If you do a lot of operations on the same resource, you should use the *resolve* command to find out the URI of the resource, and use that later.

The resolve command

This will figure out the "real" address for a "unicore6://" URL as defined above.

```
ucc resolve u6://SHARE/
```

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 u6://SHARE/pdfs
ucc cp u6://SHARE/test.pdf .
ucc cp u6://SHARE/test1 u6://SHARE/test3 data/
```

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 transfers:

```
ucc cp u6://SHARE/*.pdf u6://Demo-SITE/Home/
```

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" u6://SHARE/*.pdf u6://Demo-SITE/Home/
```

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 u6://OTHER-SITE/Home/test.txt u6://DEMO-SITE/ ↩
    Home/test.txt)
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.

General commands

mkdir

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

Example

```
ucc mkdir u6://DEMO-SITE/Home/testdirectory/data/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 u6://DEMO-SITE/Home/testdirectory/data/pdfs
```

rename

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

Examples

```
ucc rename u6://DEMO-SITE/Home/testdirectory/data/foo1.pdf / ↩
    testdirectory/data/foo2.pdf
```

will rename the file "foo1.pdf" to "foo2.pdf", leaving it in the same directory.

```
ucc rename u6://DEMO-SITE/Home/testdirectory/data /testdirectory/ ↩
    data_new
```

will rename the "data" directory.

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 u6://DEMO-SITE/Home/testdirectory/foo.txt
```

Finding data

ls

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

```
ucc ls u6://DEMO-SITE/Home -l -H
```

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

```
ucc ls u6://DEMO-SITE/Home/.bashrc -l -m
```

find

This command is a similar to the well-known Unix utility, however much less powerful. It allows to do recursive listings and retrieve files matching certain conditions. Currently only "name match" is available. For example to get all PDF files on a storage,

```
ucc find -r -l u6://DEMO-SITE/Home/ -N .pdf
```

The *find* command is currently implemented synchronously, and may thus run into a network timeout when it takes too long. This limitation will be overcome in future versions of this command.

Deprecated commands

While the old get-file, put-file and copy-file are still available, you should use the simpler *cp* instead.

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.

Metadata management functions

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

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, alternatively the "-m" option can be used to directly give the metadata service URL.

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 file), which is specified using the "-f" option.

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

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. The extraction process is asynchronous, so a "Task" service address will be returned which can be used to monitor the extraction process using the "wsrf getproperties" command.

Workflows

Introduction

UCC supports the UNICORE workflow system and allows to submit workflows to the workflow engine or single jobs to the service orchestrator (broker).

The workflows are executed server-side, and UCC is used only for submitting, managing data and getting results.

Command overview

The following commands are provided. More details and examples follow below.

- `workflow-submit` : submit a workflow file
- `workflow-control` : abort or resume a running workflow
- `list-workflows` : list information about workflows
- `broker-run` : use the service orchestrator to find matching sites or run single jobs

Basic use

To check the availability of workflow services, issue the following command

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

This should show at least an accessible workflow engine and service orchestrator.

The distribution contains some example workflow files in the <[?]> directory that you can edit and submit.

```
ucc workflow-submit yourworkflow.swf
```

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 <[?]> command without an explicit workflow address

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

Managing workflow data

During workflow execution, data files will be produced that the workflow system will move to a storage location that is accessible for the individual jobs. Usually this location is created automatically by UCC before the workflow is submitted, using a special UNICORE service called a storage factory. If you want to influence this decision, UCC allows to select the storage factory to be used via the "-f" option to the workflow-submit command:

```
ucc workflow-submit -f <factory-url> <workflow-file>
```

You can check the available factories with the system-info command. If not specified, UCC will use the fist storage factory it finds in the registry.

In general, storages that are dynamically created will be deleted when the workflow is deleted. To persistently store data, you need to make sure to export important result files to a persistent location (e.g. your Home on UNICORE site, or a persistent storage).

Alternatively you can directly specify a storage URL, either using the convenient "u6://. . . " notation, or as a real network URL:

```
ucc workflow-submit -S u6://MY-SITE/Home <workflow-file>

ucc workflow-submit -S https://my-gateway/SITE/services/ ↩
    StorageManagement?res=myuser-Home <workflow-file>
```

Importing local data for use by a workflow

If you have local files that need to be imported before starting the workflow, you have to specify this using a normal UCC job file that contains only an "Imports" section:

```
{
  #stage-in specification

  Imports: [
    {From: local-file.sh, To: "c9m:${WORKFLOW_ID}/input.sh"}
  ],
}
```

When submitting the workflow, add the "-u <filename>" option to specify the imports file.

This will cause UCC to copy the local file "local-file.sh" to the workflow storage space. You can refer to this file in your workflow using the "global" name "c9m:. . . ", say in a script activity:

```
....
        <jsdl:DataStaging>
        <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
        <jsdl:FileName>input.sh</jsdl:FileName>
          <jsdl:Source>
            <jsdl:URI>c9m:${WORKFLOW_ID}/input.sh</jsdl:URI>
          </jsdl:Source>
        </jsdl:DataStaging>

....
```

The workflow system will resolve the name at runtime and your file will be used. This allows you to group your files by workflow ID.

Downloading output files

You can use the usual `get-file` command to download files using the "global IDs" used by the workflow engine. Hint: the `workflow-info` command will list the files that are produced by the workflow.

Workflow templates

If the workflow's `Documentation` section contains `Argument` definitions, 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 simple example workflow

```
<s:Workflow xmlns:s="http://www.chemomentum.org/workflow/simple"
          xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
          xmlns:jsdl-p="http://schemas.ggf.org/jsdl/2005/11/jsdl- ↩
              posix"
          xmlns:jsdl-u="http://www.unicore.eu/unicore/jsdl- ↩
              extensions">

  <s:Documentation>
  <!-- template parameter definitions -->
  <jsdl-u:Argument>
    <jsdl-u:Name>UC_PRECISION</jsdl-u:Name>
    <jsdl-u:ArgumentMetadata>
       <jsdl-u:Type>choice</jsdl-u:Type>
       <jsdl-u:ValidValue>low</jsdl-u:ValidValue>
       <jsdl-u:ValidValue>medium</jsdl-u:ValidValue>
       <jsdl-u:ValidValue>high</jsdl-u:ValidValue>
    </jsdl-u:ArgumentMetadata>
  </jsdl-u:Argument>
  </s:Documentation>

  <s:Activity Id="date" Name="JSDL">
   <s:JSDL>
      <jsdl:JobDescription>
        <jsdl:Application>
          <jsdl:ApplicationName>ComputeIt</jsdl:ApplicationName>
          <jsdl-p:POSIXApplication>
```

```
            <jsdl-p:Argument>--precision ${UC_PRECISION}</jsdl-p: ↩
                Argument>
          </jsdl-p:POSIXApplication>
        </jsdl:Application>
      </jsdl:JobDescription>
    </s:JSDL>
  </s:Activity>

</s:Workflow>
```

This defines a parameter `UC_PRECISION` which is used as application argument in a job. The `ArgumentMetadata` follow the same specification as application metadata in the UNICORE IDB. The UCC will now try to take the value for `UC_PRECISION` from the supplied `.u` file. As an example, the .u file could contain

```
{
  "UC_PRECISION" : "high",
}
```

Before submission, UCC will then replace this value, resulting in the workflow

```
<s:Workflow xmlns:s="http://www.chemomentum.org/workflow/simple"
          xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
          xmlns:jsdl-p="http://schemas.ggf.org/jsdl/2005/11/jsdl- ↩
              posix"
          xmlns:jsdl-u="http://www.unicore.eu/unicore/jsdl- ↩
              extensions">

  <s:Documentation>
  <!-- ... -->
  </s:Documentation>

  <s:Activity Id="date" Name="JSDL">
   <s:JSDL>
      <jsdl:JobDescription>
        <jsdl:Application>
          <jsdl:ApplicationName>ComputeIt</jsdl:ApplicationName>
          <jsdl-p:POSIXApplication>
            <jsdl-p:Argument>--precision high</jsdl-p:Argument>
          </jsdl-p:POSIXApplication>
        </jsdl:Application>
      </jsdl:JobDescription>
    </s:JSDL>
  </s:Activity>

</s:Workflow>
```

As this is just string replacement, variables can be used anywhere in the workflow, including application definitions, file staging etc.

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 (Section 7) 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", ".jsdl" or ".xml".

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
```

UCC can also process JSDL files, to batch-process these, use the "-j" option:

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

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 9: 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)

Table 9: (continued)

Option (short and long form)	Description
`-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,--noResourceCheck`	Do not check if the necessary application is available on the target system (will increase performance a bit)
`-X,--noFetchOutcome`	Do not fetch standard output and error
`-S,--submitOnly`	Only submit the jobs, do not wait for them to finish
`-M,--maxNewJobs`	Limit the number of job submissions (default: 100)
`-s,--sitename`	Specify which site to use
`-W,--siteWeights`	Specify a file containing site weights
`-j,--jsdl`	Assume jobs are in JSDL format instead of the default JSON .u files

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".

OGSA-BES functions

Assuming you have successfully installed UCC Section 2, this section shows you how to manage and monitor jobs on OGSA-BES services using UCC. The set of commands not only supports the UNICORE implementation, but may also work with implementations in other Grid middlewares compliant with OGF's OGSA-BES specification.

OGSA-BES Setup

In UNICORE style, users are required to provide a Registry URL inside the preferences file. For BES users it is not always the case that an endpoint is advertised via a UNICORE Registry. Therefore, the configuration options allow user to modify this behaviour.

```
contact-registry=[true|false]
```

Users who whish to disable UCC calling the registry can set the "contact-registry" option to false. By default the "contact-registry" option is true.

When setting "contact-registry" to false, OGSA-BES users must provide at least one BESFactory URL using the following format.

```
bes.1=https://site1.com/services/BESFactory
bes.2=https://site2.com/services/BESFactory
bes.3=https://site3.com/services/BESFactory
bes.4=file:///tmp/bes-jugene.xml
bes.5=/tmp/bes-juropa.xml
...
```

If the "contact-registry" option is set to false and no OGSA-BES URL is specified, UCC will report an error. To use an XML endpoint reference (EPR) read from a file for contacting a BESFactory service, the contents of a EPR file must validate against the WS-Addressing's endpoint reference schema. See below the contents of the sample endpoint reference file,

```
<wsa:EndpointReference xmlns:wsa="http://www.w3.org/2005/08/ ↩
    addressing">
  <wsa:Address>https://localhost:8080/DEMO-SITE/services/BESFactory ↩
      ?res=default_bes_factory</wsa:Address>
</wsa:EndpointReference>
```

In the above XML snippet, under the "Address" tag, you must specify the URL of a target BESFactory service.

For the sake of convenience, here is an XML infoset representation taken from the WS-Addressing specification:

```
<EndpointReference>
    <Address>xs:anyURI</Address>
    <ReferenceParameters>xs:any*</ReferenceParameters> ?
    <Metadata>xs:any*</Metadata>?
</EndpointReference>
```

Running and monitoring OGSA-BES jobs

UCC provides an easy to use command for submitting jobs on OGSA-BES complaint endpoints. To send a job read from a JSDL file,

```
ucc bes-submit-job -j hellompi.xml -s bes.3 -v
```

Alternatively, the job can be submitted using a BESFactory URL or endpoint reference file path.

```
ucc bes-submit-job -j hellompi.xml -s https://example3.com/services ↩
    /BESFactory -v
```

or

```
ucc bes-submit-job -j hellompi.xml -s file:///tmp/bes-jugene.xml -v
```

The JSON job description Section 7 can also be used, although only a subset of JSON constructs are supported for the OGSA-BES extensions.

Users can fetch the job status by specifying the descriptor (.job) file. This file is automatically generated after a successful execution of "bes-submit-job" command. Example:

Fetch job status example:

```
ucc bes-job-status jobid.job
```

Job can be terminated using a job descriptor file:

```
ucc bes-terminate-job jobid.job
```

To list BESFactory properties:

```
ucc bes-list-att -s bes.1
```

The above command will result in BESFactory's properties without jobs information. To see the list of the user's jobs on a BESFactory

```
ucc bes-list-job -s bes.1
```

Get OGSA-BES job outputs

The UCC OGSA-BES extension allow users to fetch job's output through the descriptor (.job) file. The command is called "bes-get-output". It supports downloading standard error and output files. The common usage scenario is, as soon as user issues the command, UCC will attempt to download the output files. Before downloading these files, UCC probes the job's status to check whether the job has successfully finished. If it is, then the output files will be fetched instantly, otherwise UCC waits until the job approaches completion.

Get job output example:

```
ucc bes-get-output jobid.job
```

Note that, at the moment the bes-get-output command only supports UNICORE based BES implementations. However, it can be used against other BES services which implement UNICORE specific server extensions such as BES-Activity and UNICORE's native StorageManagementService.

Enabling username/password authentication

Some BES implementations support authentication using username and password. To add username and password to the messages sent by UCC, the UCC preferences file must contain the following settings

```
#
# setup username/password
#
uas.security.out.handler.classname=de.fzj.unicore.bes.security. ↩
    UsernameOutHandler
de.fzj.unicore.bes.security.UsernameOutHandler.wsUserName=< ↩
    your_username>
de.fzj.unicore.bes.security.UsernameOutHandler.wsPassword=< ↩
    your_password>
de.fzj.unicore.bes.security.UsernameOutHandler. ↩
    wsActivateUsernameProfile=true
```

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 usually started by

```
ucc shell
```

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
```

Exiting the shell

To exit, type `exit` or press CTRL-D

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
```

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.

Starting with server version 7.3, UNICORE supports ACLs on a per-service instance basis. This means, that you can give other users access to your target systems, jobs, storages,

For example, you might have access to an S3 cloud storage via UNICORE, and you want to securely share data on this resource. Or, you want to allow others to check job 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.

Editing ACLs

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

```
ucc share <URL>
```

to showe the current ACL for the given resource, where "URL" is the full WSRF service URL of the resource, e.g.

```
ucc share https://localhost:8080/DEMO-SITE/services/ ↩
    StorageManagement?res=demo-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>
```

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.

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.

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.

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]
    ...
  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 *ToggleJob-Submission* 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=cdfdafc5-0274-464d-ac4a ↩
    -463f46c942fa
SUCCESS, service reply: Job information for cdfdafc5-0274-464d-ac4a ↩
    -463f46c942fa
{Info=Action ID       : cdfdafc5-0274-464d-ac4a-463f46c942fa
Action type     : JSDL
Status          : DONE (trans.: none)
Result          : SUCCESSFUL [Success.]
Owner           : CN=Demo User,O=UNICORE,C=EU
Exec. Definition: <JobDefinition xmlns="http://schemas.ggf.org/jsdl ↩
    /2005/11/jsdl">
  <JobDescription>
    <JobIdentification>
      <JobName>Date</JobName>
    </JobIdentification>
    <Application>
      <jsdl:POSIXApplication xmlns:jsdl="http://schemas.ggf.org/ ↩
          jsdl/2005/11/jsdl-posix">
        <jsdl:Executable>/bin/date</jsdl:Executable>
      </jsdl:POSIXApplication>
    </Application>
    <Resources/>
  </JobDescription>
</JobDefinition>
Orig. Definition: <JobDefinition xmlns="http://schemas.ggf.org/jsdl ↩
    /2005/11/jsdl">
  <JobDescription>
    <JobIdentification>
      <JobName>Date</JobName>
    </JobIdentification>
    <Application>
      <ApplicationName>Date</ApplicationName>
      <POSIXApplication xmlns="http://schemas.ggf.org/jsdl/2005/11/ ↩
          jsdl-posix"/>
    </Application>
    <Resources/>
  </JobDescription>
</JobDefinition>
Processing context: de.fzj.unicore.xnjs.ems. ↩
    ProcessingContext@4308cff0
Application Info: AppInfo for Date 1.0
Job log:
Thu Sep 08 09:25:03 CEST 2016: Created with ID cdfdafc5-0274-464d- ↩
    ac4a-463f46c942fa
Thu Sep 08 09:25:03 CEST 2016: Created with type 'JSDL'
Thu Sep 08 09:25:03 CEST 2016: Client: Name: CN=Demo User,O=UNICORE ↩
    ,C=EU
Xlogin: uid: [schuller], gids: [addingOSgroups: true]
Role: user: role from attribute source
Security tokens: User name: CN=Demo User,O=UNICORE,C=EU
Consignor DN: CN=Demo User,O=UNICORE,C=EU
Delegation to consignor status: true, core delegation status: true
Message signature status: UNCHECKED
Client's original IP: 127.0.0.1
Thu Sep 08 09:25:04 CEST 2016: Using default execution environment.
Thu Sep 08 09:25:04 CEST 2016: No staging in needed.
Thu Sep 08 09:25:04 CEST 2016: Status set to READY.
Thu Sep 08 09:25:04 CEST 2016: Status set to PENDING.
Thu Sep 08 09:25:04 CEST 2016: Incarnated resources: [CPUsPerNode ↩
    =1.0, MemoryPerNode=2.68435456E8, ArraySize=1, Nodes=1.0,  ↩
    ArrayLimit=64, CPUTime=3600.0]
Thu Sep 08 09:25:04 CEST 2016: Command is:
#!/bin/sh
# ....
chmod u+x /bin/date 2> /dev/null
rm -f /opt/unicore-servers/FILESPACE/cdfdafc5-0274-464d-ac4a-463 ↩
    f46c942fa//UNICORE_SCRIPT_EXIT_CODE
 /bin/date

echo $? > /opt/unicore-servers/FILESPACE/cdfdafc5-0274-464d-ac4a ↩
```

```
    -463f46c942fa//UNICORE_SCRIPT_EXIT_CODE


Thu Sep 08 09:25:04 CEST 2016: TSI reply: submission OK.
Thu Sep 08 09:25:04 CEST 2016: Submitted to classic TSI as [ ↩
    schuller NONE] with BSSID=3269519504309
Thu Sep 08 09:25:13 CEST 2016: Exit code 0
Thu Sep 08 09:25:13 CEST 2016: Job completed on BSS.
Thu Sep 08 09:25:14 CEST 2016: Status set to DONE.
Thu Sep 08 09:25:14 CEST 2016: Result: Success.
Thu Sep 08 09:25:14 CEST 2016: Total: 10.01 sec., Stage-in: 0.05  ↩
    sec., Stage-out: 0.00 sec., Datamovement: 0 %}
```

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

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 XMLNAME OPERATOR VALUE
```

where XMLNAME is the name of an XML Element from the WSRF resource properties document.

For example, to list all running 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
```

etc.

Table 10: Filtering options

Operator (long and short form)	**Description**
`equals, eq`	String equality (ignoring case)

Table 10: (continued)

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

Low-level operations

UCC supports several low-level operations using the "wsrf" command.

To destroy a resource,

```
ucc wsrf destroy <Address>
```

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

```
ucc wsrf getproperties <Address>
```

To extend the lifetime of a resource

```
ucc wsrf extend <Address> <Days>
```

These commands can be abbreviated, e.g. + ucc wsrf d <Address>

Scripting

UCC can execute Groovy scripts. Groovy (http://groovy.codehaus.org) 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 UNICORE and UCC work.

Script context

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

Table 11: Variables accessible for scripts

variable	description	Java type
registry	A preconfigured client for accessing the registry	de.fzj.unicore.uas.client.IRegistryQuery
configurationProvider	Security configuration provider (keystore, etc)	de.fzj.unicore.ucc.authn.UCCConfigurationProvider
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

Examples

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

Here is a script that will delete all your finished jobs (use at your own risk):

Groovy example: delete all your jobs

```
/*
* remove all jobs
*/

//import UNICORE/X client classes
import de.fzj.unicore.uas.client.*;
import eu.unicore.security.wsutil.client.authn. ↩
    DelegationSpecification

//helper: kills a job with the given status (SUCCESSFUL, RUNNING,  ↩
    ...)

def kill(job, statuscode){
   if (job.status.toString() == statuscode)job.destroy()
}

//iterate over TSSs and remove all jobs
def lister = new de.fzj.unicore.uas.lookup.SiteLister(registry, ↩
    configurationProvider)

lister.each {
    it.jobs.each{
         messageWriter.message "Job at "+it.address.stringValue
         securityProperties = configurationProvider. ↩
            getClientConfiguration(it,DelegationSpecification. ↩
            STANDARD)
         kill(new JobClient(it, securityProperties),"SUCCESSFUL")
    }
}
```

Groovy example: list available storages

```
/*
* list available storages
*/
import de.fzj.unicore.uas.client.*
import eu.unicore.security.wsutil.client.authn. ↩
    DelegationSpecification
import javax.xml.namespace.QName

//porttype of storage service
def SMSPORT=new QName("http://unigrids.org/2006/04/services/sms"," ↩
    StorageManagement")

//method to extract storage name from a storage client
def findName(epr){
  securityProperties = configurationProvider.getClientConfiguration ↩
      (epr,DelegationSpecification.STANDARD)
  sms=new StorageClient(epr, securityProperties)
  return sms.resourcePropertiesDocument.storageProperties. ↩
      fileSystem.name
}

//list storages from registry
registry.listAccessibleServices(SMSPORT).each {
    name=findName(it)
    messageWriter.message "Storage <"+name+"> at "+it.address. ↩
        stringValue
}

//list storages attached to target systems
def lister = new de.fzj.unicore.uas.lookup.SiteLister(registry, ↩
    configurationProvider)

lister.each {
    it.storages.each {
        name=findName(it)
            messageWriter.message "Storage <"+name+"> at "+it. ↩
                address.stringValue
    }
}
```

Frequently asked questions

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, 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.

How can I set the HTTP connection timeout?

In your properties file, set

```
client.http.connection.timeout=<timeout in milliseconds>
```

How can I log the SOAP messages sent and received by UCC?

In your properties file, set

```
#log messages
client.messageLogging=true
```

which will log the messages on INFO level to the log file.

How can I generate a proxy cert and add it to my message in order to use e.g. GridFTP?

In your properties file, add

```
#enable proxy cert out handler
client.outHandlers=de.fzj.unicore.uas.security.ProxyCertOutHandler
```

which will add a handler that creates a proxy cert and adds it to the message.

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 Section 6.

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

Using the *wsrf* command, and the UNIX grep utility, this is very easy, for example

```
ucc wsrf getproperties https://localhost:8080/DEMO-SITE/services/ ↩
    Registry?res=default_registry | grep Address
```

will list the addresses of all services registered in the registry.

How can I use plain JSDL files instead of a .u JSON file for job submission?

Add the "-j" option when submitting a job.

I get strange errors related to security

Please read the general UNICORE FAQ on www.unicore.eu[the UNICORE website] which contains descriptions of many common errors.

Are the JSDL documents and workflow documents validated?

The JSDL documents passed to the run command and to submit-workflow are validated, and any errors are logged. If you wish UCC to stop in case of validation errors, you need to set a property

```
ucc.validation.fail_on_errors=true
```