UNICORE TSI MANUAL

UNICORE Team

Document Version:	1.0.0
Component Version:	7.12.0
Date:	22 01 2019

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

Contents

Overview

The UNICORE TSI is a daemon running on the frontend of the target resource (e.g. a cluster login node). It provides a remote interface to the operating system, the batch system and the file system of the target resource. It is used by the UNICORE/X server to perform tasks on the target resource, such as submitting and monitoring jobs, handling data, managing directories etc.

The TSI performs the work on behalf of UNICORE users and so must be able to execute processes under different uids and gids. Therefore, in production it must be run with sufficient privileges to allow this (during development and testing it can be run as a normal user).

You can configure the TSI and UNICORE/X to communicate via SSL. In this case, you need a server certificate for the TSI. For details, see Section Section 5.4

The TSI is one point where UNICORE's seamless model meets local variations and so will usually need to be adapted to the target system. This is described in Section Section 6

Note
In production environments, the TSI will run with root privileges. Make sure to read and understand section Section 11 on security and hardening the system.

Prerequisites

The TSI requires Python Version 2.6.7 or later, both Python 2 and 3 are supported. It works only on Unix-style operating systems (e.g. Linux or Mac OS/X), Windows is not (yet) directly supported.

Batch system status checks (*qstat*) will be executed under a system account (usually *unicore*) which is configured in the UNICORE/X server configuration. Note that this system account cannot be *root*, as the TSI will never execute anything as root.

The system account MUST be able to see the batch jobs from all users! If necessary, configure your batch system accordingly. For details on this procedure we refer to the documentation of your batch system.

Installation

The TSI is available either as a generic distribution (part of the UNICORE core server bundle, or as a separate tgz archive) or as a batch system specific package (such as an RPM, deb or tgz for Torque or Slurm).

Batch system specific distribution

Use the installation tools of your operating system to install the package. The following table shows the location of the TSI files.

Table 1: TSI Directory Layout for the OS-specific distribution

Name in this manual	Location	Description
CONF	/etc/unicore/tsi	Configuration files
BIN	/usr/share/unicore/tsi/bin	Start/stop scripts
LIB	/usr/share/unicore/tsi/lib	Python files
LOGS	/var/run/unicore/tsi/logs	Log files

Generic distribution

The generic TSI distribution contains several TSI variations for many popular batch systems.

Before being able to use the TSI, you must install one of the TSI variants and configure it for your local environment.

- Execute the installation script `Install.sh` and follow the instructions to copy all required files into a new TSI installation directory.
- Adapt the configuration as described below

In the following, `TSI_INSTALL` refers to the directory where you installed the TSI. This has the following sub-directories

Table 2: TSI Directory Layout for the generic distribution

Name in this manual	Location	Description
TSI_INSTALL		Base directory chosen during execution of Install.sh
CONF	TSI_INSTALL/conf	Configuration files
BIN	TSI_INSTALL/bin	Start/stop scripts
LIB	TSI_INSTALL/lib	Python files
LOGS	TSI_INSTALL/logs	Log files

File permissions

The permissions on the TSI files should be set to read only for the owner. The default installation procedure will initially take care of this. As the TSI is executed as root you should never leave any TSI files (or directories) writable after any update.

Configuring the TSI

The TSI is configured by editing the `CONF/tsi.properties` and `CONF/startup.properties` files. Please review these two files carefully.

Changes outside the config files should not be necessary, except for new portings and any local adaptations, as detailed in the next section. If changes are made, they should be passed on to the UNICORE developers, so that they can be incorporated into future releases of the scripts. To do that, send mail to unicore-support@lists.sf.net or use the issue tracker at https://sourceforge.net/p/unicore/issues

Verifying

Before starting the TSI, you should make sure that the batch system integration is working correctly. See the section on "Adapting the TSI to your system" below!

UNICORE/X configuration

UNICORE/X configuration is described fully in the relevant UNICORE/X manual. Here we just give the most important steps to get the TSI up and running.

The relevant UNICORE/X config file is the XNJS config file (usually it is called `xnjs_legacy.xml`)

Hostnames and ports

UNICORE/X needs to know the TSI hostname and port:

```
<eng:Property name="CLASSICTSI.machine" value="frontend. ↩
   mycluster.org"/>
<eng:Property name="CLASSICTSI.port" value="4433"/>
```

SSL support

If you wish to setup SSL for the UNICORE/X-to-TSI communication, please refer to section Section 5.4.

ACL support

The TSI (together with UNICORE/X) provides a possibility to manipulate file Access Control Lists (ACLs). To use ACLs, the appropriate support must be available from the underlying file system. Currently only the so called POSIX ACLs are supported (*so called* as in fact the relevant documents POSIX 1003.1e/1003.2c were never finalized), using the popular `setfacl` and `getfacl` commands. Most current file systems provide support for the POSIX ACLs.

Note
that the current version is relying on extensions of the ACL commands which are present in the Linux implementation. In case of other implementation (e.g. BSD) the ACL module should be extended, otherwise the default ACLs (which are used for directories) support will not work.

To enable POSIX ACL support you typically must ensure that:

- the required file systems are mounted with ACL support turned on,
- the `getfacl` and `setfacl` commands are available on your machine.

Configuration of ACLs is performed in the `tsi.properties` file. First of all you can define a location of `setfacl` and `getfacl` programs with `tsi.setfacl` and `tsi.getfacl` properties. By providing absolute paths you can use non-standard locations, typically it is enough to leave the default, non-absolute values which will use programs as available under the standard shell search path. Note that if you will comment any of those properties, the POSIX ACL subsystem will be turned off.

Configuration of ACL support is per directory, using properties of the format: `tsi.acl.PATH`, where *PATH* is an absolute directory path for which the setting is being made. You can provide as many settings as required, the most specific one will be used. The valid values are *POSIX* and *NONE* respectively for POSIX ACLs and for turning off the ACL support.

Consider an example:

```
tsi.acl./=NONE
tsi.acl./home=POSIX
tsi.acl./mnt/apps=POSIX
tsi.acl./mnt/apps/external=NONE
```

The above configuration turns off ACL for all directories, except for everything under `/home` and everything under `/mnt/apps` with the exception of `/mnt/apps/external`.

Warning
Do not use symbolic links or .. or . in properties configuring directories - use only absolute, normalized paths. Currently spaces in paths are also unsupported.

Note
The ACL support settings are typically cached on the UNICORE/X side (for a few minutes). Therefore, after changing the TSI configuration (and after resetting the TSI) you have to wait a bit until the new configuration is applied also in UNICORE/X.

ACL limitations

There is no ubiquitous standard for file ACLs. "POSIX draft" ACLs are by far the most popular however there are several other implementations. Here is a short list that should help to figure out the situation:

- POSIX ACLs are supported on Linux and BSD systems.
- The following file systems support POSIX ACLs: Lustre, ext{2,3,4}, JFS, ReiserFS and XFS.
- Solaris ACLs are very similar to POSIX ACLs and it should be possible to use TSI to manipulate them at least partially (remove all ACL operation won't work for sure and note that usage of Solaris ACLs was never tested). Full support may be provided on request.
- NFS version 4 provides a completely different, and currently unsupported implementation of ACLs.
- NFS version 3 uses ACLs with the same syntax as Solaris OS.
- There are also other implementations, present on AIX or Mac OS systems or in AFS FS.

Note that in future more ACL types may be supported and will be configured in the same manner, just using a different property value.

Enabling SSL for the UNICORE/X - TSI communication

SSL support should be enabled for the UNICORE/X - TSI communication to increase security. This is a MUST when UNICORE/X and TSI run on the same host, and/or user login is possible on the UNICORE/X host, to prevent attackers gaining control over the TSI.

You need

- a private key and certificate for the TSI,
- the CA certificate of the TSI certificate
- the DN (subject distinguished name) of the UNICORE/X servers that shall be allowed to connect to the TSI,
- the CA certificate of the UNICORE/X certificate.

The certificate of the TSI signer CA must be added to the UNICORE/X truststore.

The following configuration options must be set in `tsi.properties` :

- `tsi.keystore` : file containing the private TSI key in PEM format
- `tsi.keypass` : password for decrypting the key
- `tsi.certificate` : file containing the TSI certificate in PEM format
- `tsi.truststore` : file containing the certificate of the accepted CA(s) in PEM format
- `tsi.allowed_dn.NNN` : allowed DNs of UNICORE/X servers in RFC or OpenSSL format

SSL is activated if the keystore file is specified in tsi.properties.

The truststore file contains the CA cert(s):

```
-----BEGIN CERTIFICATE-----

 ... PEM data omitted ...

-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----

 ... PEM data omitted ...

-----END CERTIFICATE-----
```

The `tsi.allowed_dn.NNN` properties are used to specify which certificates are allowed, for example:

```
tsi.allowed_dn.1=CN=UNICORE/X 1, O=UNICORE, C=EU
tsi.allowed_dn.2=CN=UNICORE/X 2, O=UNICORE, C=EU
```

Note
If you do not specify any access control entries, all certificates issued by trusted CAs are allowed to connect to the TSI. Be very careful to prevent illicit access to the TSI!

When UNICORE/X connects, its certificate is checked:

- the UNICORE/X cert has to be valid (i.e. issued by a trusted CA and not expired)
- the subject of the UNICORE/X cert is checked against the configured ACL (list of allowed DNs).

On the UNICORE/X side, set the following property in the `xnjs_legacy.xml` file

```
<!-- enable SSL using the UNICORE/X key and trusted certificates ↩
    -->
<eng:Property name="CLASSICTSI.ssl.disable" value="false"/>
```

Adapting the TSI to your system

Environment and paths

The environment and path settings for the main TSI process and all its child processes (TSI workers) are controlled in the `startup.properties` file.

Important
Please revise the path and environment settings in the main "startup.properties" config file.

These should include the path to all executables required by the TSI, notably the batch system commands, and if applicable, the ACL commands.

As the TSI process runs as root, and switches to the required user/group IDs before each request, setting up the required environment per user has to be done carefully. Per-user settings are usually done on the UNICORE/X level using "IDB templates", please consult the UNICORE/X documentation.

Assigning groups to the current user

The current user will all her groups assigned. On some systems the default Python function used for resolving a user's groups does not see all the groups. If this is the case, set in tsi.properties

tsi.use_id_to_resolve_gids=true

This will use a different implementation via the system command *id -G <username>*.

Batch system integration: BSS.py

This file contains the functions specific to the used batch system, specifically it prepares the job script, deals with job status reporting and job control.

Even if you run a well-supported batch system such as Torque or Slurm, you should make sure that the job status reporting works properly.

Also, any site-specific resource settings (e.g. settings related to GPUs, network topology etc) are dealt with in this file.

Reporting free disk space

UNICORE will often invoke the "df" command which is implemented in the IO.py file in order to get information about free disk space. On some distributed file systems, executing this command can take quite some time, and it may be advisable to modify the "df" function to optimize this behaviour.

Reporting computing time budget

If supported by your site installation, users might have a computing time budget allocated to them. The BSS.py module contains a function `get_budget` that is used to retrieve this budget as a number e.g. representing core-hours. By default, this function returns "-1" to indicate that computing time is not budgeted.

Filtering cluster working nodes

Starting from version 6.5.1 the TSI can filter nodes based on the properties defined for nodes in BSS configuration. It can limit working nodes only to those having shared file system. It can be defined in the `tsi.properties` file by setting the property `tsi.nodes_filter`.

Note that this feature is not working for all batch systems. Currently it is supported in: Torque and SLURM.

Resource reservation

The reservation module Reservation.py is responsible for interacting with the reservation system of your batch system.

Note that this feature is not available for all batch systems. Currently it is included in: Torque and SLURM.

Execution model

The initial TSI process to be started is called the TSI shepherd which will respond to UNICORE/X requests and start up TSI workers to do the work for the UNICORE/X server. The TSI workers connect back to the UNICORE/X server.

It is possible to use the same TSI from multiple UNICORE/X servers.

Since the TSI runs with root privileges, it must authenticate the source of commands as legitimate. To do this, the TSI is initialised with the address(es) of the machine(s) that runs the UNICORE/X. The TSI shepherd will only accept requests from the defined UNICORE/X machine(s). The callback port can be pre-defined in `tsi.properties` as well. If it is undefined, the TSI will attempt to read it from the UNICORE/X connect message.

Note that it is possible to enable SSL on the TSI shepherd port, see below. In SSL mode, there is no check of the UNICORE/X address.

If the UNICORE/X process dies any TSI workers that are connected to UNICORE/X will also die. However, the TSI shepherd will continue executing and will supply new TSI processes when the UNICORE/X server is restarted. Therefore, it is not necessary to restart the TSI daemon when restarting UNICORE/X.

If a TSI worker stops execution, UNICORE/X will request a new one to replace it.

If the TSI shepherd stops execution, then all TSI processes will also be killed. The TSI shepherd must then be restarted, this does not happen automatically.

Directories used by the TSI

The TSI must have access to the "filespace" directory specified in the UNICORE/X configuration (usually the property *XNJS.filespace* in *xnjs_legacy.xml*) to hold job directories. These directories are written with the TSI's uid set to the xlogin for which the work is being performed and so must be world writable. The Unix access mode is *1777*.

Running the TSI

Starting

If installed from an Linux package, the TSI can be started using the init script

```
/etc/init.d/unicore-tsi start
```

The TSI can also be started using the script "BIN/start.sh".

Depending on the shell used to start the TSI it may be necessary to execute these commands through nohup if you want to log out afterwards.

Stopping the TSI

If installed from an Linux package, the TSI can be stopped using the init script

```
/etc/init.d/unicore-tsi stop
```

The main TSI process (shepherd) can be killed (preferably using SIGTERM). Since this results in the killing of all TSI processes this should only be done when the NJS has been stopped. However, under Linux it was found that killing the TSI shepherd will not kill the TSI workers.

The TSI can also be killed using the script "BIN/stop.sh" (cf. section Scripts). This will kill the TSI shepherd and the tree of all spawned processes including the TSI workers.

TSI worker processes (but not the shepherd) will stop executing when the UNICORE/X server it connects to stops executing.

It is possible to kill a TSI worker process but this could result in the failure of a job (but the UNICORE/X server will recover and create new TSI processes).

TSI logging

TBD

Porting the TSI to other batch systems

Most variations are found in the batch subsystem commands, porting to a new BSS usually requires changes to the following files:

- BSS.py
- Reservation.py (reservation functions if applicable)

It is recommended to start from a up-to-date and well-documented TSI, e.g. the Torque or Slurm variation. If you have further questions regarding porting to a new batch system, please use the unicore-support@lists.sf.net or unicore-devel@lists.sf.net mailing lists.

Securing and hardening the system

In a normal multi-user production setting, the TSI runs with root privileges, and thus it is critical to prevent illicit access to the TSI, which would allow accessing or destroying arbitrary user data, as well as impersonating users and generally wreaking havoc.

Once the connection to the UNICORE/X is established, the TSI is controlled via a simple text-based API. An attacker allowed to connect to the TSI can very easily execute commands as any valid (non-root) user.

In non-SSL mode, the TSI checks the IP address of the connecting process, and compare it with the expected one which is configured in the `tsi.properties` file.

In SSL mode, the TSI checks the certificate of the connecting process, by validating it against its truststore which is configured in the `tsi.properties` file.

We recommed the following measures to make operating the TSI secure.

- Prevent all access to the TSI's config and executable files. This is usually done by setting appropriate file permissions, and usually already taken care of during installation. See section Section 4.

- Make sure only UNICORE/X can connect to the TSI. This is most reliably done by configuring SSL for the UNICORE/X to TSI communication. See section Section 5.4.
- If SSL cannot be used, the UNICORE/X should run on a separate machine.
- On the UNICORE/X machine, user login should be impossible. This will prevent bypassing the IP check (in non-SSL mode) and/or accessing the UNICORE/X private key (in SSL mode).
- If you for some reason HAVE to run UNICORE/X and TSI on the same machine, and user login or execution of user commands is possible on that machine, you MUST use SSL, and take special care to protect the UNICORE/X config files and keystore using appropriate file permissions. Not using SSL in this situation is a serious risk! An attacker connecting to the TSI can impersonate any user and access any user's data (except for the root user).
- An additional safeguard is to establish monitoring for UNICORE/X, and kill the TSI in case the UNICORE/X process terminates.

Summarizing, it is critical to protect config files and executable files. We strongly recommend to configure SSL. Using SSL is a MUST when users can login to the UNICORE/X machine.

The TSI API

This document describes the API to the TSI as used by UNICORE/X (more concretely, the XNJS subsystem of UNICORE/X). The parts of the TSI that interact with the target system have been isolated and are documented here with their function calls.

The functions are implemented in the TSI as calls to Python methods. Input data from the UNICORE/X server is passed as arguments to the method. Output is returned to the UNICORE/X server by calling some global methods documented below or by directly accessing the TSI's command and data channels. TSIs are shipped with default implementations of all the functions and can be tailored by changing the supplied code or by implementing new versions of the functions that need to change for the system.

Note that this document is not a complete definition of the API, it is a general overview. The full API specification can be derived by reading the TSI code supplied with a UNICORE release.

Initialisation

For connecting to the UNICORE/X server, a callback mechanism is used. First, the UNICORE/X server will contact the main TSI process to request the creation of a new TSI worker process. The main TSI will call back the UNICORE/X server and create the necessary communications. It will receive any initialisation information send by the UNICORE/X server. After successful creation of the TSI worker process, the UNICORE/X server can communicate with the worker and ask it to execute commands. The UNICORE/X-to-TSI connection uses two sockets, a data and a command socket.

After initialisation is complete, the `process()` function (in the TSI.py module) is entered, which reads messages from the UNICORE/X server and dispatches processing to the various TSI functions.

Messages to the UNICORE/X server

The TSI provides methods to pass messages to the UNICORE/X server. In particular the UNICORE/X server expects every method to call either ok() or failed() at the end of its execution. The messaging methods are implemented in Connector.py:

- `ok(string)` Sends a message to the UNICORE/X server to say that execution of the command was successful.
- `failed(string)` Sends a message to the UNICORE/X server to say that execution of the command failed. The string is sent to the UNICORE/X server as part of the failure message.

Messages have to end with a special tag "ENDOFMESSAGE", since the command sockets are left open for receiving the next command.

User identity and environment setting

In production mode the TSI will be started as a privileged user capable of changing the TSI worker process' uid and gid to the values requested by the UNICORE/X server. This change is made before the TSI executes any external actions. The idenity is passed as a line in the message string sent by the UNICORE/X server, which starts with `#TSI_IDENTITY`.

The TSI performs three types of work: the execution and monitoring of jobs prepared by the user, transfer and manipulation of files on storages and the management of Uspaces (job working directory). Only the first type of work, execution of jobs, needs a complete user environment. The other two types of TSI work use a restricted set of standard commands (mkdir, cp, rm etc) and should not require access to specific environments set up by users. Furthermore, job execution is not done directly by the TSI but is passed on to the local Batch Subsystem which ensures that a full user environment is set before a job is executed. Therefore, the TSI only needs to set a limited user environment for any child processes that it creates. The TSI sets the following environment in any child process:

- `$USER` This is set to the user name supplied by the UNICORE/X server.
- `$LOGNAME` This is set to the user name supplied by the UNICORE/X server.
- `$HOME` This is set to the home directory of the user as given by the target system's password file.
- `$PATH` This is inherited from the parent TSI process (see the `tsi.properties` file).

Localisations of the TSI will also need to set any other environment necessary to access the BSS.

For testing, the TSI may be started as a non-privileged user, when no changing of uid and gid is possible.

Method dispatch

To determine which method to call, the TSI checks the message from the UNICORE/X server for the occurrence of special tags (followed by a new line). For example, the occurrence of a `#TSI_SUBMIT` tag will lead to execution of the `BSS.submit()` function. Before entering any method, user/group ID switching is performed, as explained in the previous section.

Job execution and job control functions

Job submission (#TSI_SUBMIT)

The `submit(string)` function submits a user script to the BSS.

Input

As input, the script to be executed is expected. The string from the UNICORE/X server is processed to replace all instances of $USER by the user's name and $HOME by the user's home directory. No further processing needs to be done on the script.

The UNICORE/X server will embed information in the script that the TSI may need to use. This information will be embedded as comments so no further processing is needed. Each piece of information will be on a separate line with the format:

```
#TSI_<name> <value>
```

If the value is the string *NONE*, then the particular information should not be supplied to the BSS during submission. The information is:

- `#TSI_JOBNAME` This is the name that should be given to the job. If this is `NONE`, the TSI will use a default jobname.
- `#TSI_PROJECT` The user's project (for accounting)
- `TSI_STDOUT` and `#TSI_STDERR` The names for standard output and error files.
- `TSI_OUTCOME_DIR` The directory where to write the stdout and stderr files to. In general this is the same as `#TSI_USPACE_DIR`
- `#TSI_USPACE_DIR` The initial working directory of the script (i.e. the Uspace directory).
- `#TSI_TIME` The run time (wall clock) limit requested by this job in seconds

- `TSI_MEMORY` The memory requirement of the job. The UNICORE/X server supplies this as a "megabytes per node" value
- `#TSI_TOTAL_PROCESSORS` The number of processors required by the job.
- `#TSI_PROCESSORS` The number of processors per node required by the job.
- `#TSI_NODES` The number of nodes required by this job.
- `#TSI_QUEUE` The BSS queue to which this job should be submitted.
- `#TSI_UMASK` The default umask for the job
- `#TSI_EMAIL` The email address to which the BSS should send any status change emails.
- `#TSI_RESERVATION_REFERENCE` If the job should be run in a reservation, this parameter contains the reservation ID.
- `#TSI_ARRAY` If multiple instances of the same job are to be submitted, this contains the list of array IDs, e.g. "1-100", or "2,4,6".
- `#TSI_ARRAY_LIMIT` If multiple instances of the same job are to be submitted, this optionally limits the number of concurrently running instances. E.g. "5" will limit the number of instances to "5".
- `#TSI_BSS_NODES_FILTER <filterstring>` Administrators can define a string in the IDB which is to be used as nodes filter, if the BSS supports this.

In addition to these, additional site-specific resources (e.g. GPUs) can be defined on the UNICORE/X server, which are passed via `#TSI_SSR_<resource_name> <resource_value>` lines.

Output

- Normal: the output is the BSS identifier of the job unless the execution was interactive. In this case the execution is complete when the TSI returns from this call and the output is that from `ok()`.
- Error: `failed()` called with the reason for failure

Script execution (#TSI_EXECUTESCRIPT)

The function `TSI.execute_script()` executes the script directly from the TSI process, without submitting the script to the batch subsystem. This function is used by the UNICORE/X server to create and manipulate the Uspace, to perform file management functions, etc. The UNICORE/X server also uses this to execute user defined code, for example when user precommands or postcommands are defined in execution environments.

Input

The script to be executed. The string from the UNICORE/X server is processed to replace all instances of `$USER` by the user's name and `$HOME` by the user's home directory. No further processing needs to be done on the script. If a `#TSI_DISCARD_OUTPUT` string is present, no output will be gathered.

Output

- Normal: The script has been executed. Concatenated stderr and stdout from the execution of the script is sent to the UNICORE/X server following the `ok()` call.
- Error: `failed()` called with the reason for failure.

Job control

- `#TSI_ABORTJOB` The `BSS.abort_job()` function sends a command to the BSS to abort the named BSS job. Any stdout and stderr produced by the job before the abort takes effect must be saved.
- `#TSI_CANCELJOB` The `BSS.cancel_job()` function sends a command to the BSS to cancel the named BSS job. Cancelling means both finishing execution on the BSS (as for abort) and removing any stdout and stderr.
- `#TSI_HOLDJOB` The `BSS.hold_job()` function sends a command to the BSS to hold execution of the named BSS job. Holding means suspending execution of a job that has started or not starting execution of a queued job. Note that suspending execution can result in the resources allocated to the job being held by the job even though it is not executing and so some sites may not allow this. This is dealt with by the relaxed post condition below. Some sites can hold a job's execution and release the resources held by the job (leaving the job on the BSS so that it can resume execution). This is called freezing. The UNICORE/X server can send a request for a freeze (#TSI_FREEZE) which the TSI may execute, if there is no freeze command initialised the TSI may execute a hold in its place An acceptable implementation is for hold_job to return without executing a command.
- `#TSI_RESUMEJOB` The `BSS.resume_job()` function sends a command to the BSS to resume execution of the named BSS job. Not that suspending execution can result in the resources allocated to the job being held by the job even though it is not executing and so some sites may not allow this. An acceptable implementation is for resume_job to return without executing a command (if hold_job did the same).

Input

All job control functions require the BSS job ID as parameter in the form `#TSI_BSSID <id entifier>`

Output

- Normal: the job control function was invoked. No extra output.
- Error: `failed()` called with the reason for failure.

Detailed job info (#TSI_GETJOBDETAILS)

- `#TSI_GETJOBDETAILS` the `BSS.get_job_details()` function sends a command to the BSS requesting detailed information about the job. The format and content is BSS specific, and is sent to UNICORE/X without any further processing.

Input

All job control functions require the BSS job ID as parameter in the form `#TSI_BSSID <id entifier>`

Output

- Normal: detailed job information sent via `ok()`
- Error: `failed()` called with the reason for failure.

Status listing (#TSI_QSTAT)

This `BSS.get_status_listing()` function returns the status of all the jobs on the BSS that have been submitted through any TSI providing access to the BSS.

This method is called with the TSI's identity set to the special user ID configured in the UNICORE/X server (`CLASSICTSI.priveduser` property). This is because the UNICORE/X server expects the returned listing to contain every UNICORE job from every UNICORE user but some BSS only allow a view of the status of all jobs to privileged users.

Input

None.

Output

- Normal: The first line is *QSTAT*. There follows an arbitrary number of lines, each line containing the status of a job on the BSS with the following format: "id status <queuename>", where `id` is the BSS identifier of the job and `status` is one of: QUEUED, RUNNING, SUSPENDED or COMPLETED. Optionally, the queue name can be listed as well. The output must include all jobs still on the BSS that were submitted by a TSI executing on the target system (including all those submitted by TSIs other than the one executing this command). The output may include lines for jobs on the BSS submitted by other means.

- Error: `failed()` called with the reason for failure.

Getting the user's remaining compute budget (#TSI_GET_COMPUTE_BUDGET)

This `BSS.get_budget()` function returns the remaining compute budget for the user (in core hours) or "-1" if not known or not applicable.

Input

None.

Output

- Normal: A line "USER <nnn>" is sent via `ok()`. The "<nnn>" is the remaining core hours, or "-1" if not applicable or the budget is not known.
- Error: `failed()` called with the reason for failure.

I/O functions

Reading a file (#TSI_GETFILECHUNK)

The `IO.get_file_chunk()` function is called by the UNICORE/X server to fetch the contents of a file.

Input

- `#TSI_FILE <file name>` The full path name of the file to be sent to the UNICORE/X server
- `#TSI_START <start byte>` Where to start reading the file
- `#TSI_LENGTH <chunk length>` How many bytes to return

The file name is modified by the TSI to substitute all occurrences of the string *$USER* by the name of the user and all occurrences of the string *$HOME* by the home directory of the user.

Output

- Normal: The UNICORE/X server has a copy of the request part of the file (sent via the data socket)
- Error: `failed()` is called with the reason for failure.

Writing files (#TSI_PUTFILECHUNK)

The `put_file_chunk()` function is called by the UNICORE/X server to write the contents of one file to a directory accessible by the TSI.

Input

The `#TSI_FILESACTION` parameter contains the action to take if the file exists (or does not): 0 = don't care, 1 = only write if the file does not exist, 2 = only write if the file exists, 3 = append to file.

The `#TSI_FILE` parameter contains the filname and permissions.

The `#TSI_LENGTH` parameter contains the number of bytes to read from the data channel and write to disk.

The TSI replies with TSI_OK, and the data to write is then read from the data channel.

Output

- Normal: The TSI has written the file data.
- Error: `failed()` called with the reason for failure.

File ACL operations (#TSI_FILE_ACL)

The `process_acl` function allows to set or get the access control list on a given file or directory. Please refer to the file `ACL.py` to learn about this part of the API.

Listing directories and getting file information (#TSI_LS)

This function allows to list directories or get information about a single file.

Input

The `#TSI_FILE` parameter contains the file/directory name.

The `#TSI_LS_MODE` parameter contains the kind of listing: "A" = info on a single file, "R" = recursive directory listing, "N" = normal directory listing

Output

- Normal: The TSI writes the listing to the command socket, see the `IO.py` file for a detailed description of the format
- Error: TSI replies with `TSI_FAILED` and the reason for failure.

Getting free disk space (#TSI_DF)

This function allows to get the free disk space for a given path.

Input

The `#TSI_FILE` parameter contains the file/directory name.

Output

- Normal: The TSI writes the disk space info to the command socket, see the `IO.py` file for a detailed description of the format.
- Error: TSI replies with `TSI_FAILED` and the reason for failure.

Resource reservation functions

The TSI offers functionality to create and manage reservations. These are implemented in the file `Reservation.py`, different versions for different scheduling systems exist.

Creating a reservation (#TSI_MAKE_RESERVATION)

This is used to create a reservation.

Input

- `#TSI_RESERVATION_OWNER <xlogin>` The user ID (xlogin) of the reservation owner
- `#TSI_STARTTME <time>` The requested start time in ISO8601 format (yyyy-MM-dd'T'HH:mm:ssZ)
- The requested resources are passed in in the same way as for job submission

Output

- Normal: The command replies with a single reservation ID string.
- Error: `failed()` called with the reason for failure

Querying a reservation (#TSI_QUERY_RESERVATION)

This is used to query the status of a reservation.

Input

- `#TSI_RESERVATION_REFERENCE <reservation_ID>` The reservation reference that shall be queried

Output

- Normal: The command produces two lines. The first line contains the status (UNKNOWN, INVALID, WAITING, READY, ACTIVE, FINISHED or OTHER) and an optional start time (ISO 8601). The second line contains a human-readable description
- Error: `failed()` called with the reason for failure

Cancelling a reservation (#TSI_CANCEL_RESERVATION)

This is used to cancel a reservation.

Input

- `#TSI_RESERVATION_REFERENCE <reservation_ID>` The reservation reference that is to be cancelled

Output

- Normal: `ok()` called with no special output
- Error: `failed()` called with the reason for failure

Contact

UNICORE Homepage: http://www.unicore.eu

Support mailing list: unicore-support@lists.sourceforge.net

Developers mailing list: unicore-devel@lists.sourceforge.net (needs registration)

TSI issues management: https://sourceforge.net/p/unicore/issues