UNICORE TSI: MANUAL

UNICORE Team

Document Version:	1.0.0
Component Version:	6.5.0
Date:	02 05 2012

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

Contents

1 Overview

The UNICORE TSI is a Perl 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).

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.

2 Prerequisites

The TSI requires Perl Version 5.004 or later. If you must use a previous version of Perl, then you should read and change the first few lines of the "tsi" file.

3 Installation

The TSI is available either as part of the UNICORE core server bundle, or as a separate package (such as an RPM). In the first case, the TSI files are available in the `tsi/` subdirectory, in the second case they are available in the `/usr/share/unicore/tsi` folder.

The TSI distribution contains TSI variations for many popular batch systems, which are available in the sub-directory "./tsi". Additional TSIs for other environments are available in subdirectory "./tsi_contrib". These are either not tested or are intended for older batch systems. They are provided because they might still be useful for somebody. As a rule of thumb you can assume that those older versions require some modifications in order to work with the current version of UNICORE - please contact the support mailing list and we can provide help.

The NOBATCH TSI is used when no batch sub-system is present. It needs also all files common to all installations from directory ./tsi/SHARED. The specific TSIs with batch sub-system are composed of all common files from ./tsi/SHARED plus the files for the specific operation system and/or batch sub-system, e.g. ./tsi/aix_ll

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 "tsi" file and any other configuration as described below

An essential task of the installation process is the correct setting of the file permissions which is described in the following paragraph.

3.1 Directories

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

Table 1: TSI Directory Layout

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
PERL	TSI_INSTALL/perl	Perl modules and helper scripts
LOGS	TSI_INSTALL/logs	Log files

4 File permissions

The permissions on the TSI Perl files should be set to read only for the owner. As the TSI is executed as root you should never leave any of these files (or the directories) writable after any update.

Note, however, that the tsi_ls and tsi_df files must be world readable (the directory permissions must also be set to world executable), because it has to be read from any user id when executing a ListDirectory request.

The recommended permissions are set by executing the command been generated by a previous call of Install.sh.

In particular, Install_permissions.sh sets the file permissions to world readable for tsi_ls and tsi_df and world executable for the tsi_installation_directory. However, this is not sufficient. All parent directories of tsi_installation_directory have to be world executable as well (world readable is NOT required). For this reason, a short path to the TSI might be preferable.

5 Configuring the TSI

The TSI is configured by editing the TSI files in tsi_installation_directory. Basic configuration is done in the `conf/tsi.properties` file.

The further configuration has been concentrated into the "tsi" file and the part of this file that should be changed is clearly marked. This includes the locations of the commands to interact with the BSS. Additionally you can review the SharedConfiguration.pm file where are additional settings (common to all BSSes/OS TSI variants) which are rarely changed. Again the configuration section is clearly marked there.

Changes outside the above described parts should not be necessary (except for new portings, cf. next paragraph), but if they are made they should be passed on to the UNICORE developers so that they can be incorporated into future releases of the scripts (send mail to unicore-support@lists.sf.net or use the trackers at http://sourceforge.net/projects/unicore).

The necessary changes can be different for different systems and so you should read the first part of your "tsi" file where the required changes are marked and commented.

5.1 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 called `xnjs_legacy.xml`)

5.1.1 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"/>
```

5.1.2 Script locations

The TSI uses the auxiliary script `tsi_ls` to list files. Similarly, a `tsi_df` file is used to report the free disk space. These scripts are supplied with the TSI, and the UNICORE/X configuration needs to be edited so that they can be found. This is done by specifying the full path to the scripts in the configuration file.

In `xnjs_legacy.xml` file, set

```
<eng:Property name="CLASSICTSI.TSI_LS" value="/my_full_tsi_path/perl/tsi_ls"/>
<eng:Property name="CLASSICTSI.TSI_DF" value="/my_full_tsi_path/perl/tsi_df"/>
```

5.2 ACL support

The TSI (together with UNICORE/X from the version 6.4.1 up) provides a possibility to manipulate file Access Control List. 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 relaying 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.

Note
the default ACL support is present from the version 6.4.2.

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 directory `/tmp` (`/` is the most specific setting for `/tmp`), turns on the POSIX ACLs for everything under `/home` and everything under `/mnt/apps` except 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 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.

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

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

5.3 Enabling SSL for the UNICORE/X - TSI communication

SSL support can be enabled for the UNICORE/X - TSI communication to increase security.

SSL is activated if the keystore file is specified in tsi.properties. Keystore and truststore must be in pem format. When UNICORE/X connects, its certificate is checked with the CA cert. If this cert is correctly signed and if it's present in the truststore,

The TSI allows the connection if the following conditions are true: - the XNJS cert is valid (i.e. has been issued by a trusted CA) - the XNJS cert is present in the truststore

In SSL mode, the TSI's IP check is deactivated.

Technically, the SSL code uses the "IO::Socket::SSL" perl module. This module is actively maintained and is present on the most of package managers. For example, it can be downloaded from the CPAN archive and installed manually. You will also require the "Net::SSLeay" module.

On the UNICORE/X side, add a line to the "Core" section of the `xnjs_legacy.xml` file

```
<!-- enable SSL using the normal UNICORE/X key and trusted certificates -->
<eng:LoadComponent>de.fzj.unicore.uas.xnjs.XNJSSecurityConfiguration</eng:LoadComp
```

6 Execution model

The TSI has two modes of execution. The first process to be started is 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 worker 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.

If the UNICORE/X process dies any TSI workers that are connected to the XNJS 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.

7 Directories used by the TSI

The TSI must have access to the "filespace" directory specified in the IDB 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.

8 Starting

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

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

The TSI can be started with or without command line arguments.

When executed with command line arguments the format is:

```
perl tsi njs_machine njs_port my_port
```

where the NJS is executing on njs_machine and is listening for TSI worker connections on njs_port (njs_port must match the first port number in the SOURCE entry of the EXECUTION_TSI section in the NJS's IDB file). A TSI process in shepherd mode will listen for NJS requests on my_port (my_port must match the second port number in the SOURCE entry of the EXECUTION_TSI section in the NJS?s IDB file).

Alternatively, the TSI can be started without command line arguments. In this case the variables $main::njs_machine, $main::njs_port, $main::my_port must be set in the tsi Perl file for your system.

As a third alternative, the TSI can be started using the script "start_tsi" (cf. section Scripts).

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.

9 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 TSI 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 "kill_tsi" (cf. section Scripts). This will kill the TSI shepherd and the tree of all spawned processes including the TSI workers.

TSI worker processes will stop executing when the XNJS stops executing.

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

10 TSI logging

The TSI daemon writes log information to stdout and stderr, to save these they are usually redirected to a file. The logging directory is configured in `tsi.properties` using the `tsi.logdir` property. If this is set to *syslog*, the Linux syslog facility is used.

11 Scripts

Several scripts are available to simplify the starting (and if needed killing) of the TSI. Before using the scripts it might be necessary to adapt the path to Perl in the scripts.

`start_tsi [-d] [conf_dir]`

start_tsi starts the TSI based on the evaluation of the properties file CONF/tsi.properties. The properties file determines the path to the TSI, the NJS machine, and the ports for the connections between TSI processes (shepherd and worker) and the NJS. If conf_dir is not specified the current working directory is searched for the properties file. An example file is available in conf/tsi.properties.

The process number of the shepherd TSI is saved in file conf_dir/LAST_TSI_PIDS.

If the TSI does not send its log information to the NJS, it is saved in current date, time, and the port numbers.

Option -d starts the TSI under the interactive Perl debugger.

`find_pids [conf_dir]`

find_pids evaluates the process number of the shepherd TSI from file conf_dir/LAST_TSI_PIDS. It shows the tree of all child processes (including the TSI workers) which have been spawned by the shepherd process.

`kill_tsi [conf_dir]`

In general, the TSI processes will be stopped through the njs_admin command *tsi stop*. However, there might be situations where this is no longer possible (NJS hangs, ...). kill_tsi uses find_pids to determine all shepherd and worker processes (and their child processes). Finally all these processes are killed.

`../bin/list_log_files type [conf_dir]`

list_log_files is identical to the scripts which are available for the Gateway and the NJS. The script returns the names of all or some of the log files in the default logging directory conf_dir/logs. Please read the corresponding Gateway/NJS documentation for details.

12 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:

- Submit.pm (submission to the BSS)
- GetStatusListing.pm (parsing of the queue status info)
- Reservation.pm (reservation functions if applicable)

These are the files which are found in the `./tsi/<variant>` and `./tsi_contrib/<variant>` sub-directories.

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

13 The TSI API

This document describes the API to the TSI as used by UNICORE/X (XNJS). 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 Perl methods (with the methods loaded through modules). Input data from the XNJS is passed as arguments to the method. Output is returned to the XNJS by calling some global methods documented below or by directly accessing the TSI's command and data channels. TSIs are be 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.

13.1 Initialisation

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

After initialisation is complete, the `infinite_loop()` function (MainLoop.pm module) is entered which reads messages from the XNJS and dispatches processing to the various TSI functions.

13.2 Messages to the XNJS

The TSI provides methods to pass messages to the XNJS. In particular the XNJS expects every method to call either ok_report or failed_report at the end of its execution. The messaging methods are:

- `ok_report(string)` Sends a message to the XNJS to say that execution of the command was successful. The string is also logged as a debug message.
- `failed_report(string)` Sends a message to the XNJS to say that execution of the command failed. The string is sent to the XNJS as part of the failure message. It is also logged.
- `debug_report(string)` Logs string as a debug message.

13.3 User identity and environment setting

In production mode the TSI will be started as a privileged user capable of changing the TSI worker process' uid and gid to the user and account requested by the UNICORE user. 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 XNJS, 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 off 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: * `$USER` This is set to the user name supplied by the XNJS. * `$LOGNAME` This is set to the user name supplied by the XNJS. * `$HOME` This is set to the home directory of the user as given by the target system's password file. * `$PATH` This is inherited from the parent TSI process (see the `tsi` script file). Localisations of the TSI can also set any other environment necessary to access the BSS. This is done through the Perl ENV array.

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

13.4 Method dispatch

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

13.5 Job submission (#TSI_SUBMIT)

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

13.5.1 Input

As input, the script to be executed is expected. The string from the XNJS 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 XNJS 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_STRERR` 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 (in megabytes). The XNJS supplies this as a 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_PREFER_INTERACTIVE <junk>` The presence of this indicates that the task should be executed *interactively* i.e. on the TSI node without submission to the BSS. The TSI can reply with an OK and not the BSS id.

13.5.2 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_report().
- Error: `failed_report()` called with the reason for failure

13.6 Reading files (#TSI_FILECHUNK)

The `get_file_chunk(string)` function is called by the XNJS to fetch the contents of a file.

13.6.1 Input

- `#TSI_FILE <file name>` The full path name of the file to be sent to the XNJS
- `#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.

13.6.2 Output

- Normal: The XNJS has a copy of the request part of the file (sent via the data socket)
- Error: `failed_report()` is called with the reason for failure.

13.7 Writing files (#TSI_PUTFILES)

The #put_files+ function is called by the XNJS to write the contents of one or more files to a directory accessible by the TSI.

13.7.1 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. This action applies to all the files is a call of put_files.

The data to write is then read from the data channel following this pseudo code:

- while there are files to transfer:

- read filename and permissions from command channel
- 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.
- while there are more bytes:
  * read packet_size from command channel
  * read packet_size bytes from data channel
  * write bytes to file

Where *permissions* are the permissions to set on the file.

13.7.2 Output

- Normal: The TSI has written the files to the directory.
- Error: `failed_report()` called with the reason for failure.

13.8 Script execution (#TSI_EXECUTESCRIPT)

This function executes the script directly from the TSI process, without submitting the script to the batch subsystem. This function is used by the XNJS to create and manipulate the Uspace, to perform file management functions, and to execute helper scripts like `tsi_ls`. The XNJS also uses this to execute user defined code, for example

13.8.1 Input

The script to be executed. The string from the XNJS 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 the a `#TSI_DISCARD_OUTPUT` string is present, no output will be gathered.

13.8.2 Output

- Normal: The script has been executed. Concatenated stderr and stdout from the execution of the script is sent to the XNJS following the `ok_report()` call.
- Error: `failed_report()` called with the reason for failure.

13.9 Job control

*+#TSI_ABORTJOB+ The +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 ef must be saved.

- #TSI_CANCELJOB The 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 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 XNJS 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 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).

13.9.1 Input

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

13.9.2 Output

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

13.10 Status listing (#TSI_QSTAT)

This 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 XNJS (CLASSICTSI.priveduser property). This is because the XNJS 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.

13.10.1 Input

None.

13.10.2 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_report()` called with the reason for failure.

13.11 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.pm` to learn about this part of the API.

13.12 Reservation

For the time being, please refer to the file `ResourceReservation.pm` to learn about the resource reservation API.

14 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)