UNICORE TARGET SYSTEM INTERFACE

Sven van den Berghe, *fecit*

Version 1.0.5, 8th January 03

1 General

This document describes the API to the TSI as used by the NJS. This API can be used to develop TSIs for new systems. The parts of the TSI that interact with the target system have been isolated and are documented here with their function calls.

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.

The functions are implemented in the TSI as calls to Perl methods (with the methods loaded through modules).Input data from the NJS is passed as arguments to the method. Output is returned to the NJS by calling some global methods documented below or by directly accessing the TSI's command and data channels.

TSIs will be shipped with complete 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.

1.1 Multithreading

The TSI implementation is single threaded. However, the NJS has been designed to be able to use multiple TSIs. This is done by having more than one TSI worker process running and so any replacement code must be able to handle concurrent calls correctly.

2 Services provided by the main TSI

2.1 Initialisation

The main TSI will contact the NJS and create the necessary communications. It will receive any initialisation information send by the NJS and then process each command from the NJS and call the appropriate method.

2.2 Messages to the NJS

The TSI provides methods to pass messages to the NJS.

In particular the NJS 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 NJS 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 NJS to say that execution of the command failed. The *string* is sent to the NJS as part of the failure message. It is also logged.

debug_report(string)

Logs *string* as a debug message.

2.3 User identity and environment setting

In production mode the TSI will be started as a privileged user capable of changing the process' uid and gid to the user and account[1] requested by the Unicore user. This change is made before the TSI executes any external actions.

The TSI performs three types of work: the execution and monitoring of jobs prepared by the user, transfer and manipulation of files on Storage Servers and the management of Uspaces (including spooled files, outcomes and streamed files). 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 NJS.

$LOGNAME. This is set to the user name supplied by the NJS.

$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.

---

[1] In a test environment the TSI may be started as a non-privileged user and so no changing of uid and gid is possible.

3 submit

This function submits the script to the BSS.

3.1 Input

The script to be executed.

The string from the NJS 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 NJS 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_OUTCOME_DIR

The NJS expects that the stdout and stderr of the job are written to files named stdout and stderr in the directory named as the value of this field.

The actual requirement is that the stdout and stderr files are in the outcome directory when end_processing() tells the NJS that the job has definitely completed execution. The information is provided here because it is possible to tell BSSs where to put stdout and stderr when a job is submitted.

#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 NJS is told through the IDB if this should be supplied as per processor, per node or per job.

#TSI_PROCESSORS

The number of processors per node required by the job.

#TSI_NODES

The number of nodes required by this job.

If the value is 0, then this job is a non-parallel job. If the value is 1, then this job has some parallel characteristic.

#TSI_FASTFS

The amount of storage to allocate on a fast temporary file system for this job (in megabytes). On the Uspace StorageServer.

#TSI_LARGEFS

The amount of storage to allocate on a large temporary file system for this job (in megabytes). On the Uspace StorageServer.

#TSI_HOMEFASTFS

The amount of storage to allocate on a fast temporary file system for this job (in megabytes). On the Home StorageServer.

#TSI_HOMELARGEFS

The amount of storage to allocate on a large temporary file system for this job (in megabytes). On the Home StorageServer.

#TSI_STORAGE_REQUEST <directory> <size>

A request for **size** megabytes of storage in **directory.** This can appear more than once.

#TSI_QUEUE

The BSS queue to which this job should be submitted.

#TSI_EMAIL

The email address to which the BSS should send any status change emails.

#TSI_SWR<name>

A SoftwareResource requested by the job. These names are site specific. One entry for each Software Resource requested by the job.

#TSI_INFO <tag> <value>

An InformationResource contained within the resources requested by the job.

#TSI_PREFER_INTERACTIVE <junk>

The presence of this indicates that the task contained a SoftwareResource whose invocation definition in the IDB says that it should be executed “interactively”. The stdout and stderr files should be placed in the Outcome directory. However, the TSI can reply with an OK and not the BSS id.

3.2 Output

3.2.1 Normal

Post condition: the script is a job on the BSS

The output is the BSS identifier of the job (also used in abort_job, cancel_job, hold_job, resume_job, get_status_listing and end_processing) 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()).

3.2.2 Error

failed_report() called with the reason for failure

4 get_directory

This function is called by the NJS to fetch the contents of all the files in a directory and in all the subdirectories (following symbolic links).

4.1 Input

The full path name of the directory whose file contents need to be sent to the NJS:

`#TSI_DIRECTORY` *`directory_name`*`.`

The *directory_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.

4.2 Output

4.2.1 Normal

Post conditions:

The NJS has a copy of the contents of the files in the directory and in all its subdirectories.

The first output line is the fully expanded directory name (on the command channel).

The TSI writes the contents of the directory on the data channel following this pseudo code:

```
while(files to return) {
  write on command channel: file_name owner_permissions
  write on command channel: file_size
  write on data channel: file contents
}
```

Where the file name is the full path name and the file size is in bytes.

The owner_permission is a single digit (between 0 and 7) giving the owner’s current permissions to set on the file. In the usual Unix convention these are:

0=none

1=eXecute only

2=Write only

3=WX

4=Read only

5=RX

6=RW

7=RWX

4.2.2 Error

failed_report() called with the reason for failure.

5 execute_script

This function executes the script directly from the TSI process, without submitting the script to the batch subsystem. This function is used by the NJS to manipulate the Uspace and Portfolios and to perform some file management functions.

The NJS does not use this to execute any user scripts.

5.1 Input

The script to be executed.

The string from the NJS 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.

5.2 Output

5.2.1 Normal

Post condition: The script has been executed

Concatenated stderr and stdout from the execution of the script is sent to the NJS following the ok_report() call.

5.2.2 Error

failed_report() called with the reason for failure.

6 abort_job

This 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.

The NJS will follow this call with a call to end_processing.

6.1 Input

The BSS identifier of the job to abort as the string *identifier* in:

```
#TSI_BSSID identifier
```

6.2 Output

6.2.1 Normal

Post condition: The job is no longer executing on the BSS.

No extra output.

6.2.2 Error

failed_report() called with the reason for failure.

7 cancel_job

This 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.

The NJS will **not** follow this call with a call to end_processing.

7.1 Input

The BSS identifier of the job to cancel as the string *identifier* in:

```
#TSI_BSSID identifier
```

7.2 Output

7.2.1 Normal

Post condition: The job is no longer execution and stdout and stderr have been deleted.

No extra output.

7.2.2 Error

failed_report() called with the reason for failure.

8 hold_job

This 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[2].

8.1 Input

The BSS identifier of the job to hold as the string *identifier* in:

```
#TSI_BSSID identifier
```

8.2 Output

8.2.1 Normal

Post condition: true.

No extra output.

8.2.2 Error

failed_report() called with the reason for failure.

---

[2] So an acceptable implementation is for hold_job to return without executing a command.

9 resume_job

This 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. This is dealt with by the relaxed post condition below[3].

9.1 Input

The BSS identifier of the job to resume as the string *identifier* in:

```
#TSI_BSSID identifier
```

9.2 Output

9.2.1 Normal

Post condition: the job is executing on the BSS or is queued and will start executing when its turn comes.

No extra output.

9.2.2 Error

failed_report() called with the reason for failure.

---

[3] An acceptable implementation is for resume_job to return without executing a command (if hold_job did the same).

10 get_status_listing

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

10.1 Input

None.

This method is called with the TSI's identity set to the special QSTAT_XLOGIN user from the NJS. This is because the NJS 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.

10.2 Output

10.2.1 Normal

Post condition: the NJS has an up to date list of the state of the jobs on the BSS.

The first line is "QSTAT\n".

There follows an arbitrary number of lines, each line containing the status of a job on the BSS with the following format:

```
identifier status
```

Where *identifier* is the BSS identifier of the job and *status* is one of: QUEUED, RUNNING or SUSPENDED.

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 (the NJS will ignore these lines).

10.2.2 Error

failed_report() called with the reason for failure.

11 end_processing

This function is called by the NJS when it suspects that a job executing on the BSS has completed. This function confirms that the job has finished executing and returns the part of the stderr produced by the job.

After successfully executing this function (on a completed job) stdout and stderr must be in the outcome directory (unless the disposition is to delete the files).

The files can be moved into the Outcome directory by this function or by the way that the BSS submit command was used.

11.1 Input

The BSS identifier of the job to check as the string *identifier* in:

`TSI_BSSID` *`identifier`*`.`

The directory for the result files in:

`TSI_OUTCOME_DIR` *`directory_name`*

The *directory_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.

The final disposition of the files in:

`TSI_DISPOSITION` *`option`*

Where *option* is either `KEEP` or `DELETE`

11.2 Output

11.2.1 Normal

Post conditions:

If the job has finished executing, then and stdout and stderr are in the TSI spool directory (as files named stdout and stderr) The NJS is sent at least the line of the stderr that contains the string “UNICORE EXIT STATUS”. The NJS is also sent the line (if any) containing the string “UNICORE DECISION”.

If the job is still executing, then “TSI_STILLEXECUTING” is sent to the NJS with a call of ok_report().

11.2.2 Error

failed_report() called with the reason for failure.

12 put_files

This function is called by the NJS to write the contents of one or more files to a directory accessible by the TSI.

12.1 Input

TSI_FILESACTION 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

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

The files are read from the data channel following this pseudo code:

```
while(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(file has more bytes) {
    read packet_size from command channel
    read packet_size bytes from data channel
    write bytes to file
  }
}
```

Where “permissions” are the owner's permissions to set on the file (following the convention of the previous section).

12.2 Output

12.2.1 Normal

Post conditions: The TSI has written the files to the directory.

None

12.2.2 Error

failed_report() called with the reason for failure.

13 Version History

8th January 03 Changes for Unicore 4

31 October 01 Extra information in a PutFiles call

28 September 01 Uspace passed to ExecuteScript, interactive flag to submit

23 July 01 Added the file owner's permissions for GetDirectory and PutFiles