Table of Contents

Table of Contents

1 User Guide

Table of Contents

1 User Guide

1 User Guide

1.1 Overview of Basic User Tasks

Univa Grid Engine offers the following basic commands, tools and activities to accomplish common user tasks in the cluster.

TABLE: Basic Tasks and Their Corresponding Commands

Task	Command
Submit Jobs	qsub, qresub, qrsh, qlogin, qsh, qmake, qtcsh
Check Job Status	qstat
Modify Jobs	qalter, qhold, qrls
Delete Jobs	qdel
Check Job Accounting After Job End	qacct
Display Cluster State	qstat, qhost, qselect, qquota
Display Cluster Configuration	qconf

The next sections provide detailed descriptions of how to use these commands in a Univa Grid Engine cluster.

1.2 A Simple Workflow Example

Using Univa Grid Engine from the command line requires sourcing the settings file to set all necessary environment variables. The settings file is located in the "<Univa Grid Engine installation path>/<Univa Grid Engine cell>/common" directory. This directory contains two settings files: `settings.sh` for bourne shell, bash and compatible shells, and `settings.csh` for csh and tcsh.

For simplicity, this document refers to the <Univa Grid Engine installation path> as `$SGE_ROOT` and the <Univa Grid Engine cell> as `$SGE_CELL`. Both environment variables are set when the settings file is sourced.

Source the settings file. Choose one of the following commands to execute based on the shell type in use.

- bourne shell/bash:

```
# . $SGE_ROOT/$SGE_CELL/common/settings.sh
```

- csh/tcsh:

```
# source $SGE_ROOT/$SGE_CELL/common/settings.csh
```

Now that the shell is set up to work with Univa Grid Engine, it is possible to check which hosts are available in the cluster by running the `qhost` command.

Sample qhost output:

```
# qhost
HOSTNAME                ARCH         NCPU  LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
-------------------------------------------------------------------------------
global                  -               -     -       -       -       -       -
kailua                  lx-amd64        4  1.03    7.7G    2.2G    8.0G     0.0
halape                  lx-x86          2  0.00  742.8M   93.9M  752.0M     0.0
kahuku                  lx-amd64        2  0.01  745.8M  103.8M  953.0M     0.0
```

The sample qhost output above shows three hosts available, all of which run Linux (lx-), two in 64 bit (amd64), one in 32 bit mode (x86). One provides 4 CPUs; the other two just 2 CPUs. Two hosts are idle but have approximately 740 MB RAM available, while the third is loaded by 25% (LOAD divided by NCPU) and has 7.7 GB RAM in total.

This sample cluster has more than enough resources available to run a simple example batch job. Use the qsub command to submit a batch job. From the example job scripts in $SGE_ROOT/examples/jobs, submit sleeper.sh.

```
# qsub $SGE_ROOT/examples/jobs/sleeper.sh
Your job 1 ("Sleeper") has been submitted
```

The qsub command sent the job to the Qmaster to determine which execution host is best suited to run the job. Follow the job's different stages with the qstat command:

- Immediately after submission, the job is in state "qw" ("queued, waiting") in the pending job list.

qstat shows the submit time (when the job was submitted to the Qmaster from the qsub command on the submit host).

```
# qstat
job-ID  prior   name       user         state submit/start at     queue           slots ja-ta
---------------------------------------------------------------------------------------------
      1 0.00000 Sleeper    jondoe       qw    03/10/2011 19:58:35                     1
```

- A few seconds later, qstat shows the job in state "r" (running) and in the run queue "all.q" on host "kahuku".

Since the job is running, qstat shows the start time (when the job was started on the execution host). A priority was automatically assigned to the job. Priority assignment is explained later in the document.

```
# qstat
job-ID  prior   name       user         state submit/start at     queue           slots ja-ta
---------------------------------------------------------------------------------------------
      1 0.55500 Sleeper    jondoe       r     03/10/2011 19:58:42 all.q@kahuku        1
```

- [changed]Between the states "qw" and "r", the job is in state "t" ("transferring") for a short time. Occasionally, this state can also be seen in the qstat output.[/changed]

While a job is running, use the qstat -j <job-ID> command to display its status:

```
# qstat -j 1
==============================================================
job_number:                 1
exec_file:                  job_scripts/1
submission_time:            Thu Mar 11 19:58:35 2011
owner:                      jondoe
uid:                        1000
group:                      users
gid:                        100
sge_o_home:                 /home/jondoe
sge_o_log_name:             jondoe
sge_o_path:                 /gridengine/bin/lx-amd64:/usr/local/sbin:/usr/local/bin:/usr/sbin
sge_o_shell:                /bin/tcsh
sge_o_workdir:              /gridengine
sge_o_host:                 kailua
account:                    sge
hard resource_list:         hostname=kailua
mail_list:                  jondoe@kailua
notify:                     FALSE
job_name:                   Sleeper
jobshare:                   0
shell_list:                 NONE:/bin/sh
env_list:
job_args:                   3600
script_file:                /gridengine/examples/jobs/sleeper.sh
binding:                    NONE
usage    1:                 cpu=00:00:00, mem=0.00000 GBs, io=0.00003, vmem=8.008M, maxvmem=8
binding    1:               NONE
scheduling info:            (Collecting of scheduler job information is turned off)
```

This simple sleeper job does nothing but sleep on the execution host. It doesn't need input, but it outputs two files in the home directory of the user who submitted the job: `Sleeper.o1` and `Sleeper.e1`. The `Sleeper.e1` file contains whatever the job printed to stderr, and it should be empty if the job ran successfully. The `Sleeper.o1` file contains what the job printed to stdout, for example:

```
Here I am. Sleeping now at: Thu Mar 10 20:01:10 CET 2011
Now it is: Thu Mar 10 20:02:10 CET 2011
```

Univa Grid Engine also keeps records of this job, as shown with the `qacct` command:

```
# qacct -j 1
==============================================================
qname        all.q
hostname     kailua
group        users
owner        jondoe
project      NONE
department   defaultdepartment
jobname      Sleeper
jobnumber    10
taskid       undefined
account      sge
priority     0
qsub_time    Thu Mar 10 19:58:35 2011
start_time   Thu Mar 10 19:58:42 2011
```

```
end_time     Thu Mar 10 19:59:43 2011
granted_pe   NONE
slots        1
failed       0
exit_status  0
ru_wallclock 61
ru_utime     0.070
ru_stime     0.050
ru_maxrss    1220
ru_ixrss     0
ru_ismrss    0
ru_idrss     0
ru_isrss     0
ru_minflt    2916
ru_majflt    0
ru_nswap     0
ru_inblock   0
ru_oublock   176
ru_msgsnd    0
ru_msgrcv    0
ru_nsignals  0
ru_nvcsw     91
ru_nivcsw    8
cpu          0.120
mem          0.001
io           0.000
iow          0.000
maxvmem      23.508M
arid         undefined
```

Refer to the `accounting(5)` man page for the meaning of all the fields output by the `qacct` command.

1.3 Displaying Univa Grid Engine Status Information

1.3.1 Cluster Overview

Several commands provide different perspectives on Univa Grid Engine cluster status information.

- `qhost` displays the status of Univa Grid Engine hosts, queues and jobs from the host perspective.
- `qstat` shows information about jobs, queues, and queue instances.
- `qconf` command, which is mainly used by the administrator for configuring the cluster, also shows the configuration of the cluster. Use it to understand why the cluster makes some decisions or is in a specific state.

1.3.2 Hosts and Queues

Univa Grid Engine monitoring and management centers around two main configuration object types: hosts and queues.

- A host represents a node in the cluster, physical or virtual. Each host has an associated host configuration object that defines the properties of that host. In addition, UGE has a global host configuration object that defines default values for all host properties. Any host that either does not have an associated host configuration object or has a host configuration object that does not set values for all host properties will inherit all or some property values from the global host configuration object.
- A queue is a set of global configuration properties that govern all instances of the queue. An instance of a queue on a specific host inherits its queue configuration properties from the queue. A queue instance may, however, explicitly override some or all of the queue configuration properties.
- Jobs are executed on a host within the context of a queue instance. Pending jobs wait in a global pending job list where they wait to be assigned by the scheduler to a queue instance.

Univa Grid Engine provides the following commands to display the states of these objects or to configure them:

- `qhost` shows the cluster status from the execution host perspective.
- `qstat` shows the cluster status from the job or queue perspective.
- `qconf` displays the cluster configuration and allows administrators to change configurations.

1.3.2.1 qhost

The `qhost` command shows the cluster status from the execution host perspective.

```
# qhost
```

Calling just `qhost` by itself prints a table that lists the following information about the execution hosts:

- architectures
- number of cores
- current load
- total RAM
- currently used RAM
- total swap space
- currently used swap space

The line "global" appears there, representing the global host, a virtual configuration object that provides defaults for all attributes of the real hosts that are not filled by real data. It's listed here just for completeness.

```
# qhost -q -j
```

- Using the "-j" option, `qhost` lists all currently running jobs beneath the hosts on which they are running.
- Using the "-q" option, `qhost` displays all queues that have instances on a host

beneath the corresponding host.
- Using both switches at once, it's possible to get a comprehensive overview over the cluster in a relatively compact output format.

To prevent lengthy output in larger clusters, `qhost` provides several options to filter the output.

- Use the "-h hostlist" option to display only the information about the listed hosts.
- Use the "-l attr=val,..." option to specify more complex filters. See section "Requestable Attributes" for more details.

For example, the following command displays only hosts of a specific architecture:

```
# qhost -l arch=lx-amd64
```

- Use the "-u user,..." option to show only jobs from the specified users. This implies the "-j" option.
- Use the "-F [attribute]" option to list either all the resources an execution host provides or just the selected ones.

See the `qhost(1)` man page for a detailed description of all options.

1.3.2.2 qstat

To view the cluster from the queue or job perspective, use the `qstat` command.

- Without any option, the `qstat` command lists all jobs of the current user.
- The "-ext" option can be added to most options of `qstat` and causes more attributes to be printed.
- With the "-u "*"" option (the asterisk must be enclosed in quotes!), the jobs of all users are displayed. With "-u <user,...>" only the jobs of the specified users are listed.
- With the "-g c" option, the status of all cluster queues is displayed.
- [changed]The "-j <job-ID> option prints information about the specified job of the current user. With a list of job-IDs or "*", this information is printed for the specified jobs or all jobs of the current user.
- The "-j" option without any job-ID prints information about all peding jobs of the current user.[/changed]

```
# qstat -f
```

- The "-f" option shows the full output of all queue instances with the jobs running in them (by default, just the jobs of the current user; add '-u "*"' to get all jobs listed for all users).

```
# qstat -F
```

- The "-F" option shows all resources the queue instances provide.

The following are several options to filter queues:

- by name (-q queue_list)
- by any provided resource (-l resource_list)
- by queue state (-qs {a|c|d|o|s|u|A|C|D|E|S})
- by parallel environments (-pe pe_list)
- access permissions for specific users (-U user_list) and to filter out queue instances where no job of the current or specified user(s) is running.

Jobs can also be filtered.

- by state (-s {p|r|s|z|hu|ho|hs|hd|hj|ha|h|a})
- by the job submitting user (-u user_list)

1.3.3 Requestable Resources

Each Univa Grid Engine configuration object (global, queue, host) has several resources whose values are either reported by loadsensors, reported by the OS or configured by a manager or an operator. These are resources such as the execution host architecture, number of slots in the queue, current load of the host or configured complex variables. A job can request that it be executed in an environment with specific resources. These requests can be hard or soft: a hard request denotes that a job can run only in an environment that provides at least the requested resource, while a soft request specifies that the job should be executed in an environment that fulfills all soft requests as much as possible. In all commands, no matter if they are made for job submission or if they are made for listing the provided resources, the option to specify the requested resources is always "-l <resource>=<value>". Each resource has a value of one of the following types:

- boolean, integer
- float
- string
- regular expression string

For example, the following command submits a job that can run on hosts with Solaris on a 64-bit Sparc CPU:

```
  # qsub -l arch=sol-sparc64 job
```

By default, this is a hard request. To specify it as a soft request, the command would change to the following:

```
  # qsub -soft -l arch=sol-sparc64 job
```

The "-soft" option denotes that all following "-l resource=value" requests should be seen as soft requests. With "-hard" the requests can be switched back to hard requests. This can be switched as often as necessary, as shown in the following example:

```
  # qsub -soft -l arch=sol-sparc64 -hard -l slots>4 -soft -l h_vmem>300M -hard -l num_cpus>2 jo
```

Using wildcards in resource requests is also permitted.

```
# qsub -l arch="sol-*" job
```

This command requests the job to be scheduled on any Solaris host. (NOTE: The quotes (") are necessary to prevent the shell from expanding the asterisk "*").

To show the list of resources a queue instance provides, enter the following command:

```
# qstat -F
```

Sample `qstat` output is shown below.

```
queuename                      qtype resv/used/tot. load_avg arch           states
---------------------------------------------------------------------------------
all.q@kailua                   BIPC  0/0/40         1.14     lx-amd64
      hl:arch=lx-amd64
      hl:num_proc=4
      hl:mem_total=7.683G
      hl:swap_total=7.996G
      hl:virtual_total=15.679G
      hl:load_avg=1.140000
      hl:load_short=1.150000
      hl:load_medium=1.140000
      hl:load_long=1.310000
      hl:mem_free=2.649G
      hl:swap_free=7.996G
      hl:virtual_free=10.645G
      hl:mem_used=5.034G
      hl:swap_used=0.000
      hl:virtual_used=5.034G
      hl:cpu=17.100000
      hl:m_topology=SCTTCTT
      hl:m_topology_inuse=SCTTCTT
      hl:m_socket=1
      hl:m_core=2
      hl:m_thread=4
      hl:np_load_avg=0.285000
      hl:np_load_short=0.287500
      hl:np_load_medium=0.285000
      hl:np_load_long=0.327500
      qf:qname=all.q
      qf:hostname=kailua
      qc:slots=40
      qf:tmpdir=/tmp
      qf:seq_no=0
      qf:rerun=0.000000
      qf:calendar=NONE
      qf:s_rt=infinity
      qf:h_rt=infinity
      qf:s_cpu=infinity
      qf:h_cpu=infinity
      qf:s_fsize=infinity
      qf:h_fsize=infinity
      qf:s_data=infinity
      qf:h_data=infinity
      qf:s_stack=infinity
      qf:h_stack=infinity
      qf:s_core=infinity
      qf:h_core=infinity
      qf:s_rss=infinity
      qf:h_rss=infinity
```

```
qf:s_vmem=infinity
qf:h_vmem=infinity
qf:min_cpu_interval=00:05:00
```

The resource list consists of three fields: <type>:<name>=<value>. The type is composed of two letters.

- The first letter denotes the origin of this resource.
  - "h" for host
  - "q" for queue
- The second letter denotes how the value is acquired.
  - "l" for load sensor
  - "f" for fixed, i.e. statically configured in the cluster, host or queue configuration
  - "c" for constant

1.3.4 User Access Permissions and Affiliations

In Univa Grid Engine, there are three general categories of users:

TABLE: User Categories

User Category	Description
managers	By default, there is always one default manager, the Univa Grid Engine administrator. Managers have universal permission in Univa Grid Engine.
operators	Operators have the permissions to modify the state of specific objects, e.g. enable or disable a queue.
other users	All other users only have permission to submit jobs, to modify and delete their own jobs, and to get information about the cluster status.

Managers are defined by the global manager list, which can be accessed through `qconf` options:

TABLE: `qconf` Options for Updating the Global Manager List

Option	Description
-am user_list	add user(s) to the manager list
-dm user_list	delete user(s) from the manager list
-sm	show a list of all managers

`qconf` provides the similar options for operators:

TABLE: `qconf` Options for Updating the Operator List

Option	Description
-ao user_list	add user to the operator list
-do user_list	delete user from the operator list

-so	show a list of all operators

By default, all users known to the operating system can use Univa Grid Engine as normal users.

Each object of Univa Grid Engine uses the configuration values set in "user_list" and "xuser_list" to determine who is allowed to use an object. The "user_list" explicitly allows access, whereas the "xuser_list" explicitly disallows access. This access is controlled through corresponding but opposite values. For example, the lists have values "acl" and "xacl" which function exactly opposite of each other. If a user is disallowed in the global cluster configuration (by using "xacl"), he may not use any object of Univa Grid Engine: he may not submit any job, but he can still get information from the cluster using `qstat`, `qhost` and so on.

Users mentioned in the "user_list" are allowed to use Grid Engine, but users mentioned in the "xuser_list" are disallowed. If a user is mentioned in both, the "xuser_list" takes precedence, so he is disallowed to use the object. If a "user_list" is defined, only users mentioned there are allowed to use the object. If a "xuser_list" is defined and the "user_list" is undefined, then all users except the ones mentioned in the "xuser_list" are allowed to use the object.

Note that the "user_list" and "xuser_list" accept only user sets, not user names. So it's necessary to define user sets before using these options of `qconf`.

TABLE: `qconf` Options for Updating the User List

Option	Description
-au user_list listname_list	add user(s) to user set list(s)
-Au fname	add user set from file
-du user_list listname_list	delete user(s) from user set list(s)
-dul listname_list	delete user set list(s) completely
-mu listname_list	modify the given user set list
-Mu fname	modify user set from file
-su listname_list	show the given user set list
-sul	show a list of all user set lists

A user set contains more information than just the names of the users in this set: see the man page access_list(5) for details. User sets can be defined by specifying UNIX users and primary UNIX groups, which must be prefixed by an '@' sign. There are two types of user sets: Access lists (type "ACL") and departments (type "DEPT). Pure access lists allow enlisting any user or group in any access list. When using departments, each user or group enlisted may only be enlisted in one department, in order to ensure a unique assignment of jobs to departments. To jobs whose users do not match with any of the users or groups enlisted under entries the defaultdepartment is assigned.

TABLE: Man Pages to See for Further Reference

Subject	Man Pages

"user_list" and "xuser_list"	sge_conf(5), queue_conf(5), host_conf(5) and sge_pe(5)
"acl" and "xacl" lists	project(5)
user lists format	access_list(5)
options to specify users and user sets	qconf(1)

1.4 Submitting Batch Jobs

1.4.1 What is a Batch Job?

A batch job is a single, serial work package that gets executed without user interaction. This work package can be any executable or script that can be executed on the execution host. Attached to this work package are several additional attributes that define how Univa Grid Engine handles the job and that influence the behavior of the job.

1.4.2 How to Submit a Batch Job

From the command line, batch jobs are submitted using the `qsub` command. Batch jobs can also be submitted using the deprecated GUI `qmon` or using the DRMAA interface.

Batch jobs are typically defined by a script file located at the submit host. This script prepares several settings and starts the application that does the real work. Univa Grid Engine transfers this script to the execution host, where it gets executed. Alternately, the script can be read from `stdin` instead of from a file. For a job that is just a binary to be executed on the remote host, the binary is typically already installed on the execution host, and therefore does not need to be transferred from the submit host to the execution host.

1.4.2.1 Example 1: A Simple Batch Job

To submit a simple batch job that uses a job script and default attributes, run the following command:

```
# qsub $SGE_ROOT/examples/jobs/simple.sh
```

If this command succeeds, the `qsub` command should print the following note:

```
Your job 1 ("simple.sh") has been submitted
```

Now check the status of the job while the job is running:

```
# qstat
```

If `qstat` doesn't print any information about this job, it has already finished. Note that `simple.sh` is a short running job.

The output of the job will be written to ~/simple.sh.o1 and the error messages to ~/simple.sh.e1, where "~" is the home directory on the execution host of the user who submitted the job.

1.4.2.2 Example 2: An Advanced Batch Job

`qsub` allows several attributes and requirements to be defined using command line options at the time the job is submitted. These attributes and requirements can affect how the job gets handled by Univa Grid Engine and how the job script or binary is executed. For example, the following command defines these attributes of the job:

```
  # qsub -cwd -S /bin/xyshell -i /data/example.in -o /results/example.out -j y example.sh arg1
```

TABLE: Explanation of Command Line Options in Example 2

Option	Description
-cwd	The job will be executed in the same directory as the current directory
-S /bin/xyshell	The shell `/bin/xyshell` will be used to interpret the job script.
-i /data/example.in	The file "/data/example.in" on the execution host will be used as input file for the job.
-o /results/example.out	The file "/results/example.out" on the execution host will be used as output file for the job.
-j y	Job output to stderr will be merged into the "/results/example.out" file.
example.sh arg1 arg2	The job script is "example.sh" must exist locally and gets transferred to the execution host by Univa Grid Engine. arg1 and arg2 will be passed to this job script.

1.4.2.3 Example 3: Another Advanced Batch Job

```
  # qsub -N example3 -P testproject -p -27 -l a=lx-amd64 example.sh
```

TABLE: Explanation of Command Line Options in Example 3

Option	Description
-N example2	The job will get the name "example3" instead of the default name which is the name of the job script.
-P testproject	The job will be part of the project "testproject".
-p -27	The job will be scheduled with a lower priority than by default.
-l a=lx-amd64	The job can get scheduled only to a execution host that provides the architecture "lx-amd64".
example.sh	The job script without any arguments.

1.4.2.4 Example 4: A Simple Binary Job

```
# qsub -b y firefox
```

The "-b y" option tells Univa Grid Engine that this is a binary job; the binary does already exist on the execution host and doesn't have to be transferred by Univa Grid Engine from the submit to the execution host.

See the `qsub(5)` man page for an explanation of all possible `qsub` options.

1.4.3 Specifying Requirements

`qsub` provides three options to specify the requirements that must be fulfilled in order to run the job on the execution host. These are requirements like the host architecture, available memory, required licenses, specific script interpreters installed, and so on.

These resource requirements are specified on the `qsub` command line using the "-l" option.

For example, to ensure the job gets scheduled only to a host that provides the architecture type "lx-x86", i.e. Linux on a x86 compatible 32 bit CPU, issue the following `qsub` option:

```
# qsub -l arch=lx-x86 my_job.sh
```

Specifying several requirements at once and using wildcards inside a requirement are possible, as in the following example:

```
# qsub -l a="sol-*|*-amd" -l h="node??" job.sh
```

This example specifies that the job requests must be scheduled to a host whose architecture string starts wit "sol-" and/or ends with "amd64". At the same time, the hostname of the execution host must start with "node" and have exactly two additional trailing characters.

There are two different kinds of requests: hard and soft requests.

- A hard request must be fulfilled in order to schedule the job to the host.
- A soft request should be fulfilled. Grid Engine tries to fulfill as many soft requests as possible.

Be default, all requests specified by the "-l" option are hard requests. The "-soft" option switches the behavior: starting with the "-soft" option, all subsequent requests are considered soft requests. A "-hard" option in the command line switches back to hard requests. "-hard" and "-soft" can be specified as often as necessary.

Example:

```
# qsub -soft -l host="node??" -hard -l h_vmem=2G -l arch="sol*" -soft -l cpu=4
```

As described above in the section "Requestable Resources", the attributes that are provided by all queue instances can be listed using `qstat`:

```
# qstat -F
```

To specify a particular queue instance, use the -q option:

```
# qstat -F -q all.q@kailua
```

As an alternative to specifying job requirements on the command line each time a job is submitted, default requirements can be specified by the job submitting user and the Univa Grid Engine administrator.

Requirements are evaluated in the following order:

- Request files
- Requests in job script
- Command line

Options defined later (e.g., at command line) override options defined earlier (e.g., in the job script). Note that soft and hard requirements are collected separately.

1.4.3.1 Request Files

Request files allow options to be set automatically for all jobs submitted. Request files are read in the following order:

1. The global request file $SGE_ROOT/$SGE_CELL/default/sge_request
2. The private request file $HOME/.sge_request
3. The application specific request file $cwd/.sge_request
4. The qsub command line

Since the request files are read in order, any option defined in more than one of them is overridden by the last-read occurrence, except for options that can be used multiple times on a command line. The resulting options are used as if they were written in the `qsub` command line, while the real `qsub` command line is appended to it, again overriding options that were specified in one of the three files. At any time, the "-clear" option can be used to discard all options that were defined previously.

In these request files, each line can contain one or more options in the same format as in the `qsub` command line. Lines starting with the hash sign (#) in the first column are ignored. See the `sge_request(5)` man page for additional information.

1.4.3.2 Requests in the Job Script

Submit options can also be defined in the jobs script. Each line of the job script that starts with "#$" or with the prefix that is defined using the "-C" option is considered to be a line that contains submit options, as in the following example:

```
#!/bin/sh

#$ -P testproject
#$ -o test.out -e test.err
```

```
  echo "Just a test"
```

These options are read and parsed before the job is submitted and are added to the job object. The location where in the job script these options are defined does not matter, but the order matters - if two options override each other, the last one wins.

1.5 Monitoring and Controlling Jobs

1.5.1 Getting Status Information on Jobs

The command line tool **qstat** delivers all the available status information for jobs. *qstat* supplies various possibilities to present the available information.

TABLE: The Most Common Ways to Use `qstat`

Command	Description
qstat	Without options, *qstat* lists all jobs but without any queue status information.
qstat -f	The *-f* option causes qstat to display a summary information of all cause including its load accompanied by the list of all queued as also all pending jobs.
qstat -ext	The *-ext* option causes qstat to displays usage information and the ticket consumption of each job.
qstat -j <job_id>	The *-j* option causes qstat to display detailed information of a currently queued job.

Examples:

```
# qstat
job-ID  prior   name       user         state submit/start at     queue
------------------------------------------------------------------------------------------------
      4 0.55500 job1       user1        r     04/28/2011 09:35:34 all.q@host1
      5 0.55500 job2       user1        r     04/28/2011 09:35:34 all.q@host2
      6 0.55500 job3       user1        r     04/28/2011 09:35:34 all.q@host2

# qstat -f
queuename                      qtype resv/used/tot. load_avg arch          states
---------------------------------------------------------------------------------
all.q@host1                    BIPC  0/3/10         0.04     lx-amd64
     16 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
     18 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
     23 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
---------------------------------------------------------------------------------
all.q@host2                    BIPC  0/3/10         0.04     lx-x86
     15 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
     19 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
     22 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
---------------------------------------------------------------------------------
all.q@host3                    BIPC  0/3/10         0.04     sol-amd64
     14 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
     17 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
     21 0.55500 Sleeper    user1        t     04/28/2011 09:36:44     1
---------------------------------------------------------------------------------
all.q@host4                      BIPC  0/3/10         1.35     lx-amd64
     20 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
     24 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
     25 0.55500 Sleeper    user1        r     04/28/2011 09:36:44     1
```

It is also possible to be informed by the Univa Grid Engine system via mail on the status change of a job. To use this feature it necessary to set the *-m* option while submitting the job. This option is available for *qsub*, *qsh*, *qrsh*, *qlogin* and *qalter*.

TABLE: Mail Options to Monitor Jobs

Option	Description
b	Send mail at the beginning of a job.
e	Send mail at the end of a job.
a	Send mail when job is aborted or rescheduled.
s	Send mail when job is suspended.
n	Send no mail (default).

Example: Univa Grid Engine will send mail at the beginning as well as the end of the job:

```
# qsub -m be test_job.sh
```

1.5.2 Deleting a Job

To delete a job, the **qdel** binary is used.

TABLE: Optional `qdel` Parameters

Parameter	Description
-f <job_id[s]>	Forces the deletion a job even if the responsible execution host does not respond.
<job_id > -t <range>	Deletes specific tasks of an array job. It is also possible to delete a specific range of array jobs.
-u <user_list>	Deletes all job of the specified user.

The behavior of how Univa Grid Engine handles a forced deletion can be altered by using the following qmaster parameters. This option can be set via **qconf -mconf** as **qmaster_params**.

TABLE: `qmaster` Parameters for Forced Job Deletion

Parameter	Description

ENABLE_FORCED_QDEL	If this parameter is set, users are allowed to force job deletion on their own jobs. Otherwise only the Univa Grid Engine managers are allowed to perform those actions.
ENABLE_FORCED_QDEL_IF_UNKNOWN	If this parameter is set, qdel <job_id> will automatically invoke a forced job deletion if the host, where the job is running, is of *unknown status*.

Examples:

Delete all jobs in the cluster (only possible for Univa Grid Engine managers):

```
# qdel -u "*"
```

Delete tasks 2-10 out of array job with the id 5:

```
# qdel 5 -t 2-10
```

Forced deletion of jobs 2 and 5:

```
# qdel -f 2 5
```

1.5.3 Re-queuing a Job

A job can be rescheduled only if its *rerun* flag is set. This can be done either at time of submission via the *-r* option of **qsub**, or belatedly via the *-r* option of **qalter** as well as via the *rerun* configuration parameter for queues. This *rerun* configuration can be set with **qconf -mq <queue_name>**.

Examples:

```
# qsub -r yes <job_script>
```

```
# qalter -r yes <job_id>
```

There are two different ways to reschedule jobs.

Examples:

Reschedule a job:

```
# qmod -rj <job_id[s]>
```

Reschedule all jobs in a queue:

```
# qmod -rq <queue|queue_instance>
```

Rescheduled jobs are designated **Rr** (e.g. shown by *qstat*).

Example:

```
# qstat -f
queuename                      qtype resv/used/tot. load_avg arch          states
---------------------------------------------------------------------------------
all.q@host1                    BIPC  0/2/10         0.01     lx-amd64
    53 0.55500 Sleeper     user1        Rr    05/02/2011 15:31:10     2
---------------------------------------------------------------------------------
all.q@host2                    BIPC  0/2/10         0.01     lx-x86
    53 0.55500 Sleeper     user1        Rr    05/02/2011 15:31:10     2
---------------------------------------------------------------------------------
all.q@host3                    BIPC  0/1/10         0.03     sol-amd64
    53 0.55500 Sleeper     user1        Rr    05/02/2011 15:31:10     1
---------------------------------------------------------------------------------
all.q@host4                    BIPC  0/0/10         0.06     lx-amd64
```

1.5.4 Modifying a Waiting Job

To change attributes of a pending job **qalter** is used.

qalter is able to change most of the characteristics of a job even those which were set as embedded flags in the script files. Consult the submit(1) main page in regards to the options that can be altered (e.g. the job script).

1.5.4.1 Altering Job Requirements

It is also possible to alter the requirements of a pending job which have been defined via the **-l** flag at time or submission.

Example:

Submit a job to host1

```
# qsub -l h=host1 script.sh
```

Alter the host-requirement of this job (with the assumed job-id 45) to host2

```
# qalter -l h=host2 45
```

Note
By altering requested requirements the with **-l**, keep in mind that the requirements become the new requirements thus the requirements which do **not** require change must be re-requested.

Example:

Submit a job with the requirement to run on host1 and and on queue2:

```
# qsub -l h=host1,q=queue2 script.sh
```

Alter the host-requirement of this job (with the assumed job-id 45) to host5 and re-request queue2 as requirement

```
# qalter -l h=host5,q=queue2 45
```

If queue2 is NOT stated in the qalter-call, the job will run on any available queue in host5.

1.5.4.2 Changing Job Priority

To change the priority of a job the **-p** option of **qalter** can be used. It is possible to alter the priority within the range between **-1023** and **1024** whereas a negative number decreases priority and a positive one to increases it.

If not submitted differently, the default priority is 0.

As previously mentioned, a user can only alter his own jobs and in this case, a user is only able to **decrease** the priority of a job. To increase the priority, the user needs to be either Univa Grid Engine administrator or Univa Grid Engine manager.

Examples:

Increase the job priority of job 45:

```
# qalter -p 5 45
```

Decrease the job priority of 45:

```
# qalter -p -5 45
```

1.5.5 Obtaining the Job History

To get the history of a job and its accounting information use **qacct**.

qacct parses the accounting file written by *qmaster* and lists all available information for a given job. This includes accounting data such as wall-clock time, cpu-time or memory consumption as also the host where job ran and e.g. the exit-status of the job script. The default Univa Grid Engine accounting file resists in *<sge_root>/<cell>/common/accounting*. See **accounting(5)** for more information e.g. how the file is composed and what information is stored in it.

Example: Show the accounting information of job 65:

```
# qacct -j 65
==============================================================
qname        all.q
hostname     host1
group        users
owner        user1
project      NONE
department   defaultdepartment
jobname      Sleeper
jobnumber    65
```

```
taskid       undefined
account      sge
priority     0
qsub_time    Mon May  9 14:27:32 2011
start_time   Mon May  9 14:27:35 2011
end_time     Mon May  9 14:28:20 2011
granted_pe   mytestpe
slots        5
failed       0
exit_status  0
ru_wallclock 45
ru_utime     0.026
ru_stime     0.019
ru_maxrss    1856
ru_ixrss     0
ru_ismrss    0
ru_idrss     0
ru_isrss     0
ru_minflt    10649
ru_majflt    0
ru_nswap     0
ru_inblock   0
ru_oublock   24
ru_msgsnd    0
ru_msgrcv    0
ru_nsignals  0
ru_nvcsw     101
ru_nivcsw    26
cpu          0.045
mem          0.000
io           0.000
iow          0.000
maxvmem      17.949M
arid         undefined
```

1.6 Other Job Types

1.6.1 Array Jobs

Array jobs are, as mentioned in Types of Workloads being Managed by Univa Grid Engine, those that start a batch job or a parallel job multiple times. Those simultaneously-run jobs are called tasks. Each job receives an unique ID necessary to identify each of them and distribute the workload over the array job.

Submit an array job:

The default output- and error-files are *job_name.[o|e]job_id* and *job_name.[o|e]job_id.task_id*. This means that Univa Grid Engine creates an output- and an error-file for each task plus one for the superordinate array-job. To alter this behavior use the -o and -e option of qsub. If the redirection options of qsub are use (-o and/or -e), the results of the individual will be merged into the defined one.

TABLE: Available Pseudo Environment Variables

Pseudo env variable	Description
$USER	User name of the submitting user
$HOME	Home directory of the submitting user
$JOB_ID	ID of the job
$JOB_NAME	Name of the job
$HOSTNAME	Hostname of the execution host
$SGE_TASK_ID	ID of the array task

The -t option of qsub indicates the job as an array job. The -t option has the following syntax:

```
qsub -t n[-m[:s]] <batch_script>
```

TABLE: `-t` Option Syntax

n	indicates the start-id.
m	indicates the max-id.
s	indicates the step size.

Examples:

qsub -t 10 array.sh submits a job with 1 task where the task-id is 10.

qsub -t 1-10 array.sh submits a job with 10 tasks numbered consecutively from 1 to 10.

qsub -t 2-10:2 array.sh submits a jobs with 5 tasks numbered consecutively with step size 2 (task-ids 2,4,6,8,10).

Besides the pseudo environment variables already mentioned, the following variables are also exposed which can be used in the script file:

TABLE: Pseudo Environment Variables Available for Scripts

Pseudo env variable	Description
$SGE_TASK_ID	ID of the array task
$SGE_TASK_FIRST	ID of the first array task
$SGE_TASK_LAST	ID of the last array task
$SGE_TASK_STEPSIZE	step size

Example of an array job script:

```
#!/bin/csh

# redirect the output-file of the batch job
#$ -o /tmp/array_out.$JOB_ID
# redirect the error-file of the batch job
#$ -e /tmp/array_err.$JOB_ID
```

```
# starts data_handler with data.* as input file
/tmp/data_handler -i /tmp/data.$SGE_TASK_ID
```

Alter an array job:

It is possible to change the attributes of array jobs. But the changes will only affect the pending tasks of an array job. Already running tasks are untouched.

Configuration variables (see *sge_conf(5)*):

max_aj_instances indicates the maximum number of instances of an array job which can run simultaneously.

max_aj_tasks indicates the maximum number of tasks a array job can have.

It is also possible to limit the maximum number of concurrently running tasks of an array job via the *-tc* switch of qsub.

Example:

Submit a job with 20 tasks but only 10 of then can run concurrently. *qsub -t 1-20 -tc 10 array.sh*

1.6.2 Interactive Jobs

Usually, Univa Grid Engine uses its own built-in mechanism to establish a connection to the execution host. It is possible to change this to e.g. ssh oder telnet, of course.

Configuration variable	Description
qlogin_command	Command to execute on local host if qlogin is started.
qlogin_daemon	Daemon to start on execution host if qlogin is started.
rlogin_command	Command to execute on local host if qrsh is started **without** a command name as argument to execute remotely.
rlogin_daemon	Daemon to start on execution host if qrsh is started **without** a command name as argument to execute remotely.
rsh_command	Command to execute on local host if qrsh is started **with** a command name as argument to execute remotely.
rsh_daemon	Daemon to start on execution host if qrsh is started **with** a command name as argument to execute remotely.

Example of a qlogin configuration:

```
qlogin_command   /usr/bin/telnet
qlogin_daemon    /usr/sbin/in.telnetd
```

The configured commands (qlogin_command, rlogin_command and rsh_command) are started with the execution host, the port number and, in case of rsh_command, also the command name to execute as arguments.

Example:

```
  /usr/bin/telnet exec_host 1234
```

Consult sge_conf(5) for more information.

1.6.2.1 qrsh and qlogin

qrsh without a command name as argument and qlogin submit an interactive job to the queuing system which starts a remote session on the execution host where the current local terminal is used for I/O. This is similar to rlogin or a ssh session without a command name.

qrsh with a command executes the command on the execution host and redirects the I/O to the current local terminal. By default, qrsh with command does not open a pseudo terminal (PTY), other than qlogin and qrsh without command, on the execution host. It simply pipes the in- and output to the local terminal. This behavior can by changed via the *-pty yes* option as there are applications that rely on a PTY.

Those jobs can only run in INTERACTIVE queues unless the jobs are not explicitly marked as non-immediate job using the *-now no* option.

1.6.2.2 qtcsh

qtcsh is a fully compatible extension of the UNIX C-shell clone tcsh (it is based on tcsh version 6.08). qtcsh provides an ordinary tcsh with the capacity to run certain defined applications distributed within the Univa Grid Engine system. Those defined applications will run in the background as an interactive qrsh call and has to be pre-defined in the **.qtask**-file.

.qtask file format:

```
[!]<app-name> <qrsh-options>
```

The optional exclamation point indicates that the users .qtask file is not allowed to overwrite the global .qtask file if set.

Example:

```
# This causes within a qtcsh-session that all rm-calls are invoked via qrsh on the denoted host
rm -l h=fileserver_host
```

This means that a

```
rm foo
```

within an qtcsh-session will be translated into

```
qrsh -l =h fileserver_host rm foo
```

1.6.2.3 qmake

qmake facilitates the possibility to distribute Makefile processing in parallel over the Univa Grid Engine. It is based on GNU Make 3.78.1. All valid options for qsub and qrsh are also available for qmake. Options which has to be passed to GNU Make has to be placed after the "--"-separator.

Syntax:

```
qmake [ options ] -- [ gmake options ]
```

Typical examples how to use qmake:

```
qmake -cwd -v PATH -pe compiling 1-10 -- -debug
```

This call changes the remote execution host into the current working directory, exports the **$PATH** environment variable and requests between 1 and 10 slots in the parallel environment *compiling*. This call is listed as one job in the Univa Grid Engine system.

This means that Univa Grid Engine starts up to 10 qrsh sessions depending on available slots and what is needed by GNU Make. The option -debug will, as it is after the "--"-separator, be passed to the GNU Make instances.

As there is no special architecture requested, Univa Grid Engine assumes the one set in the environment variable **$SGE_ARCH**. If it is not set, qmake will produce a warning and start the make process on any available architecture.

```
qmake -l arch=lx26-amd64 -cwd -v PATH --
```

Other than the example above, qmake is not bound to a parallel environment in this case. qmake will start an own qrsh job for every GNU Make rule listed in the Makefiles.

Furthermore, qmake support two different modes of invocation:

Interactive mode: qmake invoked by command line implicitly submits a qrsh-job. On this master machine the parallel make procedures will be started and qmake will distribute the make targets and steps to the other hosts which are chosen.

Batch mode: If qmake with the **--inherit** option is embedded in a simple batch script the qmake process will inherit all resource requirements from the calling batch job. Eventually declared parallel environments (pe) or the *-j* option in the qmake line within the script will be ignored.

Example:

```
#!/bin/csh
qmake --inherit --
```

Submit:

```
qsub -cwd -v PATH -pe compiling 1-10 <shell_script>
```

1.6.2.4 qsh

qsh opens a **xterm** via an interactive X-windows session on the execution host. The display is directed either to the X-server indicated by the *$DISPLAY* environment variable or the one which was set by the *-display* qsh command line option. If no display is set, Univa Grid Engine tries to direct the display to 0.0 of the submit host.

1.6.3 Parallel Jobs

A parallel job runs simultaneously across multiple execution hosts. To run parallel jobs within the Univa Grid Engine system it is necessary to set up **parallel environments** (pe). It is customary is to have several of such parallel environments e.g. for the different MPI implementations which are used or different ones for tight and loose integration. To take advantage of parallel execution, the application has to support this. There are a dozen software implementations that support parallel tasks like *OpenMPI*, *LAM-MPI*, *MPICH* or *PVM*.

Univa Grid Engine supports two different ways of executing parallel jobs:

- Loose Integration

  Univa Grid Engine generates a custom machine file listing all execution hosts chosen for the job. Univa Grid Engine does not control the parallel job itself and its distributed tasks. This means that there is no tracking of resource consumption of the tasks and no way to delete runaway tasks. However, it is easy to set up and nearly all parallel application technologies are supported.

- Tight Integration

  Univa Grid Engine takes control of the whole parallel job execution. This includes spawning and controlling of all parallel tasks. Unlike the *Loose Integration* Univa Grid Engine is able to track the resource usage correctly including all parallel tasks as also to delete runaway tasks via *qdel*. However the parallel applications has to support the tight Univa Grid Engine integration (e.g. OpenMPI which has to be built with *--enable-sge*).

1.6.3.1 Parallel Environments

Setup a parallel environment

```
qconf -ap my_parallel_env
```

This will create a parallel environment with the name *my_parallel_env*. In the opening editor it is possible to change the properties of the pe.

TABLE: Properties of the Parallel Environment (PE)

Property	Description
pe_name	The name of the parallel environment. This one has to be specified at job submission.
slots	The maximum number of slots which can be used/requested concurrently.
user_lists	User-sets which are allowed to use this pe. If NONE is set, everybody is allowed to use this pe.
xuser_lists	User-sets which are **not** allowed to use this pe. If NONE is set, everybody is allowed to use this pe.
start_proc_args	This command is started prior the execution of the parallel job script.
stop_proc_args	This command proceeds the execution of the parallel job script finished.
allocation_rule	The allocation rule is interpreted by the scheduler and helps to determine the distribution of parallel processes among the available execution hosts. There are three different rules available: • **<int>:** This defines the number of max processes allocated at each host. • **$fill_up:** All available slots on a host will be used (filled up). If there are no more slots available on this particular host, the remaining processes will be distributed to the next host. • **$round_robin:** All processes of a parallel job will be uniformly distributed of the Univa Grid Engine system.
control_slaves	This options is in control when the parallel environment is loose or tightly integrated.
job_is_first_task	This parameter indicates if the job submitted already contains one of the parallel tasks.
urgency_slots	For pending jobs with a slot range pe request, the number of slots is not determined. This setting specifies the method to be used by Univa Grid Engine to assess the number of slots such jobs might finally get. These methods are available: • **<int>:** This integer number is used as prospective number of slots. • **min:** The slot range minimum is used as prospective number of slots. • **max:** The slot range maximum is used as prospective number of slots.

	- **avg:** The average of all numbers occurring within the job's pe range request is assumed.
accounting_summary	If set to *TRUE*, the accounting summary of all tasks are combined in one single accounting record otherwise every task is stored in an own accounting record. This option is only considered if *control_slaves* is also set.

TABLE: Examples and Templates for MPI and PVM

Example/Template	Parallel Environment
/SGE_ROOT/mpi/	MPI and MPICH
/SGE_ROOT/pvm/	PVM

See sge_pe(5) for detailed information.

1.6.3.2 Submitting Parallel Jobs

TABLE: Parameters to Submit a Parallel Job

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td><b>-pe parallel_environment n[-[m]][-]m</b></td><td>This parameter indicates that this is a parallel job.
<b>Note</b>
For declaring the parallel_environment the wildcard character * is allowed (e.g. <i>mpi*</i>).
<b>TABLE: Allowed Range Specifications for Job</b>
<table>
<tr><th>Allowed Range Specification</th><th>Description and Example</th></tr>
<tr><td><b>n-m</b></td><td>Minimum n slots and Maximum m slots
2-10</td></tr>
<tr><td><b>m</b></td><td>This is an abbreviation for m-m. Exactly m slots are needed.
10</td></tr>
<tr><td><b>-m</b></td><td>This is an abbreviation for 1-m.
-10</td></tr>
<tr><td><b>n-</b></td><td>At least n slots are needed but as much as possible slots are wanted.
10-</td></tr>
</table></td></tr>
<tr><td><b>-masterq queue</b></td><td>With this parameter it is possible to define on which queue the master task has run.</td></tr>
</table>

Example:

```
qsub -pe mpi_pe 4-10 -masterq super.q mpi.sh
```

See submit(1) for more information.

1.6.4 Hadoop Jobs as a Special Kind of Parallel Job

Refer to the first few subsections in the chapter The Univa Grid Engine Hadoop Integration in the Administrator's Guide for an overview of Hadoop and a general introduction on how Hadoop is integrated into Univa Grid Engine and the benefits over using Hadoop alone. The following sections are focused on explaining how to construct and handle Hadoop jobs embedded into Univa Grid Engine.

As the title suggests, Hadoop are parallel jobs using the special parallel environment `hadoop`. So submitting a Hadoop job can be as simple as in the following example:

```
qsub -pe hadoop 8 hadoop-test.sh
```

where the resulting Hadoop job will be run on 8 nodes. A user does not need to be concerned about whether the Hadoop environment is installed properly or whether a Hadoop job might conflict with other workloads (Hadoop or other). As a matter of fact, this is one advantage of running Hadoop under Univa Grid Engine control. A MapReduce cluster will be set up appropriately for each job and potential conflicts, like having multiple Hadoop job trackers per node, are excluded. Further advantages include full process control of Univa Grid Engine over Hadoop (`hadoop` is a tightly integrated PE) and thereby complete accounting records and clean termination of all processes which Hadoop has spawned in case of a `qdel` command.

ℹ **Note**
Suspend/resume of Hadoop jobs is not supported by the Hadoop framework! The results are unpredictable!

A job as simple as the above is, however, of not much practical use because it fails to provide information about the HDFS data which the Hadoop job must access while running. This is accomplished with the `-l hdfs_input=<data-path>` option. So the above example could become

```
qsub -pe hadoop 8 -l hdfs_input=/data/sample hadoop-test.sh
```

The path specified for the input data needs to be an absolute HDFS path. This input path request is then translated into corresponding requests for HDFS data blocks as well as primary and secondary racks. This translation that confirms that the input data path is correct are performed by a Job Submission Verifyer (JSV). The JSV is mandatory to be requested in the job submission also or else the Hadoop job will get rejected. That JSV is provided by the script jsv.sh in the Hadoop integration installation directory of the Univa Grid Engine cluster. Ask the administrator for its location. With the JSV request the Hadoop job submission example expands to the following:

```
qsub -pe hadoop 8 -l hdfs_input=/data/sample -jsv /share/hadoop/jsv.sh hadoop-test.sh
```

assuming that `/share/hadoop` is where the Hadoop integration has been installed.

The final building block for a properly-constructed Hadoop job is the job shell script itself. In its simplest form, it will just invoke the `hadoop` command as one would when running Hadoop workloads interactively. There is one additional requirement, however. The Hadoop job needs to be pointed to the Hadoop configuration directory which was created for the job by the Univa Grid Engine / Hadoop integration. This is accomplished with the `--config` option of the `hadoop` command. The configuration for the job itself resides in `$TMPDIR/conf`. So a job might look as follows:

```
#!/bin/sh
#$ -S /bin/sh

$HADOOP_HOME/bin/hadoop --config $TMPDIR/conf jar $HADOOP_HOME/hadoop-*-examples.jar \
   grep /data/web/search-patterns output 'Grid Engine'
```

This job would grep for the term `Grid Engine` in the directory `/data/web/search-patterns` and would store the output in the directory `/data/web/output`. If this job were to be executed on 16 nodes then it would get submitted as follows:

```
qsub -pe hadoop 16 -jsv /share/hadoop/jsv.sh -l hdfs_input=/data/web/search grep.sh
```

assuming that the job script is stored in the file `grep.sh` and again assuming that the Hadoop integration package is installed in `/share/hadoop`. The path to the input data is provided with the `-l` option as described above and the `-jsv` option is used as well in accordance with the aforementioned description.

1.6.4.1 Running an Interactive Hadoop Command Session Within a Hadoop Job

It is also possible to use a Hadoop job as a kind of container inside which multiple Hadoop commands can be run in an interactive session. This means that the resource pool (the compute nodes) allocated for that job and the MapReduce cluster created for that job will be reused across those Hadoop commands. For this it is necessary to find the identifier for the MapReduce job tracker and use that job tracker for running the interactive Hadoop commands. Here is an example procedure which accomplishes this:

```
% qsub -pe hadoop 16 -l hdfs_input=/data/web -jsv <hadoop_copy>/jsv.sh $SGE_ROOT/examples/jobs/
Your job 15 ("Sleeper") has been submitted
% qstat -j 15 | grep context
context:                    hdfs_jobtracker=grid01:9001,hdfs_jobtracker_admin=http://grid01:500
% cp -R $HADOOP_HOME/conf .
% vi ./conf/mapred.xml
% cat ./conf/mapred.xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>



<configuration>
    <property>
        <name>mapred.job.tracker</name>
        <value>grid01:9001</value>
    </property>
</configuration>
% $HADOOP_HOME/bin/hadoop --config ./conf jar $HADOOP_HOME/hadoop-*-examples.jar grep search-pa
% $HADOOP_HOME/bin/hadoop --config ./conf jar $HADOOP_HOME/hadoop-*-examples.jar grep search-pa
...
% qdel 15
```

The first step in the example is to submit a Hadoop job which acts as a dummy keeping the MapReduce cluster active and the nodes allocated. This is accomplished by submitting a sleeper job which will run for 10 hours. The `qstat` / `grep` combination then makes use of the fact that the Univa Grid Engine / Hadoop integration stores information about the Hadoop job tracker in the **context** of the job (see here for more information regarding the job context). The identifier for the job tracker, `grid01:9001` in this example, then needs to be put in the `mapred.xml` file which resides in the MapReduce configuration directory being used for the Hadoop commands. The next lines demonstrate how this is achieved. Interactive Hadoop commands can then be executed when using a configuration directory with the job tracker adjustments. When done with all commands, the job can be terminated.

1.6.4.2 Monitoring Hadoop Jobs

Basic job monitoring for Hadoop jobs is no different from regular Univa Grid Engine jobs, i.e. the command `qstat` can be used. Given that Hadoop jobs are parallel jobs, it is possible to use `qstat -t` or `qstat -g t` to track the nodes on which the master and slave tasks of an Hadoop are running.

In addition, one can also use the Hadoop mechanisms to monitor jobs. To access them requires the information stored in the job context as already referred to in the last section. Two context variables are important here: hdfs_jobtracker, containing the URI of the job tracker service for this job, and hdfs_jobtracker_admin, being the URL for the job tracker's administrative web interface. As described above, those context variables are contained in the output of `qstat -j <job_id>`.

To monitor jobs, simply point a browser to the URL of the job tracker. Refer to the Hadoop documentation for information on the features of this tracking method.

1.6.5 Jobs with Core Binding

Today's execution hosts are usually multi-socket and multi-core systems with a hierarchy of different caches and a complicated internal architecture. In many cases it is possible to exploit the execution host's topology in order to increase the user application, performance and therefore the overall cluster throughput. The Univa Grid Engine provides a complete subsystem, which not just provides information about the execution host topology, it also allows the user to allow the application to run on specific CPU cores. Another use is so that the administrator can ensure via JSV scripts that serial user jobs are using just one core, while parallel jobs with more granted slots can be run on multiple CPU cores. In Univa Grid Engine core binding on Linux execution hosts is turned on by default, while on Solaris hosts it must be enabled per execution host by the administrator (see Enabling and Disabling Core Binding).

1.6.5.1 Showing Execution Host Topology Related Information

By default, the qhost output shows the number of sockets, cores and hardware supported threads on Linux kernel versions 2.6.16 and higher and on Solaris execution hosts:

```
> qhost
HOSTNAME                ARCH         NCPU NSOC NCOR  NTHR LOAD  MEMTOT  MEMUSE  SWAPTO  SWAPUS
----------------------------------------------------------------------------------------------
global                  -               -    -    -     -       -       -       -       -
host1                   lx-amd64        1    1    1     1 0.16  934.9M  150.5M 1004.0M     0.0
host2                   lx-amd64        4    1    4     1 0.18    2.0G  390.8M    2.0G     0.0
host3                   lx-amd64        1    1    1     1 0.06  492.7M   70.2M  398.0M     0.0
```

There are also several topology related host complexes defined after an Univa Grid Engine standard installation:

```
> qconf -sc
...
m_core              core       INT         <=    YES         NO          0        0
m_socket            socket     INT         <=    YES         NO          0        0
m_thread            thread     INT         <=    YES         NO          0        0
m_topology          topo       RESTRING    ==    YES         NO          NONE     0
m_topology_inuse    utopo      RESTRING    ==    YES         NO          NONE     0
```

The host specific values of the complexes can be shown in the following way:

```
> qstat -F m_topology,m_topology_inuse,m_socket,m_core,m_thread

queuename                      qtype resv/used/tot. load_avg arch          states
---------------------------------------------------------------------------------
all.q@host1                    BIPC  0/0/10         0.00     lx26-amd64
        hl:m_topology=SC
        hl:m_topology_inuse=SC
        hl:m_socket=1
        hl:m_core=1
        hl:m_thread=1
---------------------------------------------------------------------------------
all.q@host2                    BIPC  0/0/10         0.00     lx26-amd64
        hl:m_topology=SCCCC
        hl:m_topology_inuse=SCCCC
        hl:m_socket=1
        hl:m_core=4
        hl:m_thread=4
---------------------------------------------------------------------------------
all.q@host3                    BIPC  0/0/10         0.00     lx26-amd64
        hl:m_topology=SC
        hl:m_topology_inuse=SC
        hl:m_socket=1
        hl:m_core=1
        hl:m_thread=1
```

`m_topology` and `m_topology_inuse` are topology strings. They encode sockets (`S`), cores (`C`), and hardware supported threads (`T`). Hence `SCCC` denotes one socket host with a quad core CPU and SCTTCTTSCTTCTT would encode a two socket system with a dual-core CPU on each socket, which supports hyperthreading. The difference between the two strings is that </code>m_topology</code> remains unchanged, even when core bound jobs are running on the host, while m_topology_inuse displays the cores, which are currently occupied (with lowercase letters). For example SccCC denotes a quad-core CPU, which has <GEfullname> jobs bound on the first and second core.

m_socket denotes the number of sockets on the host.

m_core is the total number of cores, the host offers.

m_thread is the total number of hardware supported threads the host offers.

1.6.5.2 Requesting Execution Hosts Based on the Architecture

In order to request specific hosts for a job, all the complexes described in the sub-section above can be used. Because and are regular expression strings (type RESTRING) special symbols like * can be used as well.

Important: The core binding requests are hints for the execution host on how to execute the application. If such a binding request cannot be fulfilled, (because a specific core is in use already or the execution host does not have as many cores a requested) the execution daemon starts the application without any binding. In order to compensate for this drawback, the administrator can configure the cluster (via JSV scripts) so that the number of requested slots equals the number of cores requested for binding.

In the following example a quad core CPU is requested:

```
> qsub -b y -l m_topoloy=SCCCC sleep 60
```

This does not correspond to:

```
> qsub -b y -l m_core=4,m_socket=1 sleep 60
```

Because the latter request does also match to a hexacore or higher CPU because m_core is defined as "<=".

In order to get an host with a free (currently unbound) quadcore CPU:

```
> qsub -b y -l m_topology_inuse=SCCCC sleep 60
```

In order to get an host with at least one quad core CPU, wich is currently not used by a core bound job:

```
> qsub -b y -l m_topology_insuse="*SCCCC*" sleep 60
```

1.6.5.3 Requesting Specific Cores

Univa Grid Engine supports multiple schemata in order to request cores on which the job should be bound. Several adjoined cores can be specified with the linear:<amount>

request. In some cases it could be useful to distribute the job over sockets, this can be achieved with the `striding:<stepsize>:<amount>` request. Here the `stepsize</core> denotes the distance between two successive cores. The stepsize can be aligned with a <code>m_topology` request in order to get the specific architecture. The most flexible request schema is `explicit:<socket,core>[:<socket,core>[...]]`. Here the cores can be selected manually based on the socket number and core number.

Examples:

Bind a job on two successive cores if possible:

```
> qsub -b y -binding linear:2 sleep 60
```

Request a two-socket dual-core host and bind the job on two cores, which are on different sockets:

```
> qsub -b y -l m_topology=SCCSCC -binding striding:2:2 sleep 60
```

Request a quad socket hexa-core execution host and bind the job on the first core on each socket:

```
> qsub -b y -l m_topology=SCCCCCCSCCCCCCSCCCCCCSCCCCCC -binding explicit:0,0:1,0:2:0,3,0 sleep
```

1.6.6 Checkpointing Jobs

Checkpointing delivers the possibility to save the complete state of a job and to restart from this point of time if the job was halted or interrupted. Univa Grid Engine supports two kinds of Checkpointing jobs: the user-level and the kernel-level Checkpointing.

1.6.6.1 User-Level Checkpointing

User-Level Checkpointing jobs have to do their own checkpointing by writing restart files at certain times or algorithmic steps. Applications without an integrated user-level checkpointing can use a checkpointing library like the Condor project.

1.6.6.2 Kernel-Level Checkpointing

Kernel-Level Checkpointing must be provided by the executing operating systems. The checkpointing job itself does not need to do any checkpointing. This is done by the OS entirely.

1.6.6.3 Checkpointing Environments

To execute and run checkpointing jobs environments, similar to parallel jobs, are necessary to control how, when and how often checkpointing should be done.

TABLE: Handle Checkpointing Environments with `qconf`

Parameter	Description
-ackpt	add a checkpointing environment
-dckpt *<ckpt_env>*	delete the given checkpointing environment
-mckpt *<ckpt_env>*	modify the given checkpointing environment
-sckpt *<ckpt_env>*	show the given checkpointing environment

A checkpointing environment is made up of the following parameters:

TABLE: Handle Checkpointing Environments Parameters

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>ckpt_name</td><td>The name of the checkpointing environment.</td></tr>
<tr><td>interface</td><td>The type of the checkpointing which should be used. Valid types:
<table>
<tr><td>hibernator</td><td>The Hibernator kernel-level checkpointing is interfaced.</td></tr>
<tr><td>cpr</td><td>The SGI kernel-level checkpointing is used.</td></tr>
<tr><td>cray-ckpt</td><td>The Cray kernel-level checkpointing is used.</td></tr>
<tr><td>transparent</td><td>Univa Grid Engine assumes that the job submitted within this environment uses a checkpointing library such as the mentioned Condor.</td></tr>
<tr><td>userdefined</td><td>Univa Grid Engine assumes that the job submitted within this environment uses a its private checkpointing method.</td></tr>
<tr><td>application-level</td><td>Uses all interface commands configured in the checkpointing object. In case of one of the kernel level checkpointing interfaces the restart_command is not used.</td></tr>
</table></td></tr>
<tr><td>ckpt_command</td><td>Command which will be executed by Univa Grid Engine to initiate a checkpoint.</td></tr>
<tr><td>migr_command</td><td>Command which will be executed by Univa Grid Engine during a migration of a checkpointing job from one host to another.</td></tr>
<tr><td>restart_command</td><td>Command which will be executed by Univa Grid Engine if a previously checkpointed job is restarted.</td></tr>
<tr><td>clean_command</td><td>Command which will be executed by Univa Grid Engine after a checkpointing job is completed.</td></tr>
<tr><td>ckpt_dir</td><td>Directory where checkpoints are stored.</td></tr>
<tr><td>ckpt_signal</td><td>A UNIX signal which is sent by Univa Grid Engine to the job when a checkpoint is initiated.</td></tr>
</table>

when	Point of time when checkpoints are expected to be generated. Valid values for this parameter are composed by the letters s, m, x and r and any combinations thereof without any separating character in between:	
	s	The job is checkpointed, aborted and if possible migrated if the corresponding execution daemon is shut down on the job's machine.
	m	Checkpoints are generated periodically at the *min_cpu_interval* interval defined by the queue in which a job executes.
	x	A job is checkpointed, aborted and if possible, migrated as soon as the job is suspended (manually as well as automatically).
	r	A job is rescheduled (not checkpointed) when the job host goes into an unknown state and the time interval *reschedule_unknown* defined in the global/local cluster configuration is exceeded.

1.6.6.4 Submitting a Checkpointing Job

```
# qsub -ckpt <cpkt_env> -c <when_options> job
```

The *-c* option is not mandatory. It can be used to override the *when* parameters stated in the checkpointing environment.

1.6.6.5 Example of a Checkpointing Script

The environment variable **RESTARTED** is set for checkpointing jobs that are restarted. This variable can be used to skip e.g. preparation steps.

```
#!/bin/sh
#$ -S /bin/sh

# Check if job was restarted/migrated
if [ $RESTARTED = 0 ]; then
  # Job is started first time. Not restarted.
  prepare_chkpt_env
  start_job
else
  # Job was restarted.
  restart_job
fi
```

1.6.7 Immediate Jobs

Univa Grid Engine tries to start such jobs *immediately* or not at all. If, in case of array jobs, not all tasks can be scheduled immediately, none will be started.

To indicate an immediate job, the *-now* option has to be declared with the parameter *yes*.

Example:

```
# qsub -now yes immediate_job.sh
```

The *-now* option is available for **qsub**, **qsh**, **qlogin** and **qrsh**.

In case of qsub *no* is the default value for the *-now* option, in case of qsh, qlogin and qrsh vice versa.

1.6.8 Reservations

With the concept of Advance Reservations (AR) it is possible to reserve specific resources for a job, an user or a group in the cluster for future use. If the AR is possible (resources are available) and granted it is assigned an ID.

1.6.8.1 Configuring Advance Reservations

To be able to create advance reservations the user has to be member of the *arusers* list. This list is created during the Univa Grid Engine installation.

Use qconf to a user to the *arusers* list.

```
# qconf -au username arsusers
```

1.6.8.2 Creating Advance Reservations

qrsub is the command used to create advance reservations and to submit them to the Univa Grid Engine system.

```
# qrsub -a <start_time> -e <end_time>
```

The start and end times are in *[[CC]YY]MMDDhhmm[.SS]* format. If no start time is given, Univa Grid Engine assumes the current time as the start time. It is also possible to set a *duration* instead of an end time.

```
# qrsub -a <start_time> -d <duration>
```

The duration is in hhmm[.SS] format. **Examples:** The following example reserves an slot in the queue *all.q* in host *host1* starting at 04-27 23:59 for 1 hour.

```
# qrsub -q all.q -l h=host2 -a 04272359 -d 1:0:0
```

Many of the options available for qrsub are the same as for qsub.

1.6.8.3 Monitoring Advance Reservations

qrstat is the command to list and show all advance reservations known by the Univa Grid Engine system. To list all configured advance reservations type:

```
# qrstat
```

To list a special advance reservation type:

```
# qrstat <ar_id>
```

Every submitted AR has an own ID and a special state.

TABLE: Possible Advance Reservation States

State	Description
w	Waiting - Granted but start time not yet reached
r	Running - Start time reached
d	Deleted - Deleted manually
W	Warning - AR became invalid but start time is not yet reached
E	Error - AR became invalid and start time is reached

Examples:

```
# qrstat
ar-id   name       owner        state start at            end at              duration
----------------------------------------------------------------------------------------
    1              user1        w     04/27/2011 23:59:00 04/28/2011 00:59:00 01:00:00
```

```
# qrstat -ar 1
id                             1
name
owner                          user1
state                          w
start_time                     04/27/2011 23:59:00
end_time                       04/28/2011 00:59:00
duration                       01:00:00
submission_time                04/27/2011 15:00:11
group                          users
account                        sge
resource_list                  hostname=host1
granted_slots_list             all.q@host1=1
```

1.6.8.4 Deleting Advance Reservations

'*qrdel* is the command to delete an advance reservation. The command requires at least the ID or the name of the AR.

Example:

```
# qrdel 1
```

A job which refers to an advance reservation which is in deletion will also be removed. The AR will not be removed until all referring jobs are finished!

1.6.8.5 Using Advance Reservations

Advance Reservations can be used via the **-ar <ar_id>** parameter which is available for *qsub*, *qalter*, *qrsh*, *qsh* and *qlogin*.

Example:

```
# qsub -ar 1 reservation_job.sh
```

1.7 Submission, Monitoring and Control via an API

1.7.1 The Distributed Resource Management Application API (DRMAA)

The Distributed Resource Management Application API is the industry-leading open standard of the Open Grid Forum (www.ogf.org) DRMAA working group (www.drmaa.org) for accessing DRMS. The goal of the API is to provide an external interface to applications for basic tasks, like job submission, job monitoring and job control. Since this standard is adapted by most DRMS vendors it offers a very high investment protection, when developing a DRM aware software application, because it can be easily transferred to another DRM. Univa Grid Engine supports all DRMAA concepts, which allows for the movement of existing DRMAA applications from different DRM vendors.

1.7.2 Basic DRMAA Concepts

DRMAA version 1.0 specifies a set of functions and concepts. Each DRMAA application must contain an initialization and disengagement function which must be called at the beginning and at the end respectively. In order to do something useful a new DRMAA session must be created or one existing must be re-opened. When re-opening a DRMAA session, the job IDs of the session can be reused in order to obtain the job status and gain job control. In order to submit jobs, a standard job template must be allocated and filled out according to needs with the job name and the corresponding parameters. This job template than can then be submitted with a job submission routine. There are two job submission routines specified: One for individual jobs and one for array jobs. A job can be monitored and controlled (e.g. holding, releasing, suspending, resuming) once the job is complete and the exit status can be checked. Additionally DRMAA specifies a set of error codes. In order to exploit additional functionality, which is only available in Univa Grid Engine, the standard will allow this with either the native specification functionality or with job categories.

1.7.3 Supported DRMAA Versions and Language Bindings

Univa Grid Engine supports currently the DRMAA v1.0 standard and is shipped with a fully featured DRMAA C binding v1.0 and a DRMAA Java binding v1.0. The standards can be downloaded at www.drmaa.org.

1.7.4 When to Use DRMAA

Writing applications with DRMAA has several advantages: High job submission throughput with Univa Grid Engine, the defined workflow is independent from underlying DRM, it is much easier to use in programming languages like C or Java, and it is a widely known and adapted standard backed by an experienced community.

1.7.5 Examples

1.7.5.1 Building a DRMAA Application with C

1.7.5.1.1 Compiling, Linking and Running the C Code DRMAA Example

In order to compile a DRMAA application, the drmaa.h must include the file and the DRMAA library must be available. The drmaa.h file can be found in the $SGE_ROOT/include directory and the libraries are installed in $SGE_ROOT/lib/$ARCH.

In the following example the root installation directory ($SGE_ROOT) is /opt/uge800 and the architecture is lx-amd64.

```
> gcc -I/opt/uge800/include -L/opt/uge800/lib/lx-amd64 -ldrmaa -o yourdrmaaapp youdrmaaapp.c
```

In order to run yourdrmaaapp the Univa Grid Engine environment must be present and the path to the shared DRMAA library must be set.

```
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/uge800/lib/lx-amd64
> ./yourdrmaaapp
```

1.7.5.1.2 Job Submission, Waiting and Getting the Exit Status of the Job

In the following example a job session is initially opened with drmaa_init(). The return code of the all calls indicate the success of a function (DRMAA_ERRNO_SUCCESS) or if an error has occurred. In the case of an error, the error string with the corresponding message is returned. In order to submit a job, a job template must be allocated with drmaa_allocate_job_template() and the DRMAA_REMOTE_COMMAND parameters must be set. After a successful job submission with drmaa_run_job() the application waits until the job is scheduled and eventually finished. Then the exit code of the job is accessed and printed before the job session is closed by drmaa_exit.

```
#include <stdio.h>
#include "drmaa.h"

int main(int argc, char **argv) {

   /* err contains the return code of the called functions */
   int err = 0;

   /* allocate a string with the DRMAA string buffer length */
   char errorstr[DRMAA_ERROR_STRING_BUFFER];

   /* allocate a buffer for the job name */
   char jobid[DRMAA_JOBNAME_BUFFER];

   /* pointer to a job template */
   drmaa_job_template_t *job_template = NULL;

   /* DRMAA status of a job */
   int status = 0;

   /* if job exited normally */
   int exited = 0;

   /* exit code of the job */
   int exitstatus = 0;

   /* create a new DRMAA session */
   err = drmaa_init(NULL, errorstr, DRMAA_ERROR_STRING_BUFFER);

   /* test if the DRMAA session could be opened  */
   if (err != DRMAA_ERRNO_SUCCESS) {
      printf("Unable to create a new DRMAA session: %s\n", errorstr);
      return err;
   }

   /* allocate a job template */
   err = drmaa_allocate_job_template(&job_template, errorstr,
                                     DRMAA_ERROR_STRING_BUFFER);

   /* test if the DRMAA job template could be allocated */
   if (err != DRMAA_ERRNO_SUCCESS) {
      printf("Unable to allocate a new job template: %s\n", errorstr);
      /* close the DRMAA session and exit */
      err = drmaa_exit(errorstr, DRMAA_ERROR_STRING_BUFFER);
      if (err != DRMAA_ERRNO_SUCCESS) {
         printf("Unable to close DRMAA session: %s\n", errorstr);
      }
      return err;
   }

   /* specify the job */
   err = drmaa_set_attribute(job_template, DRMAA_REMOTE_COMMAND, "./job.sh",
                             errorstr, DRMAA_ERROR_STRING_BUFFER);

   if (err != DRMAA_ERRNO_SUCCESS) {
      printf("Unable to set the remote command name: %s\n", errorstr);
      /* close the DRMAA session and exit */
      err = drmaa_exit(errorstr, DRMAA_ERROR_STRING_BUFFER);
      if (err != DRMAA_ERRNO_SUCCESS) {
         printf("Unable to close DRMAA session: %s\n", errorstr);
      }
      return err;
   }

   /* submit the job */
   err = drmaa_run_job(jobid, DRMAA_JOBNAME_BUFFER, job_template, errorstr,
                       DRMAA_ERROR_STRING_BUFFER);

   /* wait for the job */
   err = drmaa_wait(jobid, NULL, 0, &status, DRMAA_TIMEOUT_WAIT_FOREVER,
                    NULL, errorstr, DRMAA_ERROR_STRING_BUFFER);

   if (err != DRMAA_ERRNO_SUCCESS) {
      printf("Unable to wait for the job: %s\n", errorstr);
      /* close the DRMAA session and exit */
      err = drmaa_exit(errorstr, DRMAA_ERROR_STRING_BUFFER);
      if (err != DRMAA_ERRNO_SUCCESS) {
         printf("Unable to close DRMAA session: %s\n", errorstr);
      }
      return err;
   }

   /* print the exit status of the job if terminated normally (and don't
    * check a function error) */
   drmaa_wifexited(&exited, status, NULL, 0);

   if (exited == 1) {
      drmaa_wexitstatus(&exitstatus, status, NULL, 0);
      printf("Exit status of the submitted job: %d\n", exitstatus);
   }

   /* free the job template */
   err = drmaa_delete_job_template(job_template, errorstr, DRMAA_ERROR_STRING_BUFFER);

   if (err != DRMAA_ERRNO_SUCCESS) {
      printf("Unable to delete the job template: %s\n", errorstr);
      /* close the DRMAA session and exit */
      err = drmaa_exit(errorstr, DRMAA_ERROR_STRING_BUFFER);
      if (err != DRMAA_ERRNO_SUCCESS) {
         printf("Unable to close DRMAA session: %s\n", errorstr);
      }
      return err;
   }

   /* close the DRMAA session and exit */
   err = drmaa_exit(errorstr, DRMAA_ERROR_STRING_BUFFER);
   if (err != DRMAA_ERRNO_SUCCESS) {
      printf("Unable to close DRMAA session: %s\n", errorstr);
      return err;
   }

   return 0;
}
```

1.7.5.2 Building a DRMAA Application with Java

When writing a Java DRMAA application it must be taken into account that the Java DRMAA library internally is based on the C DRMAA implementation. The implication is that Java DRMAA is fast, but this native code dependency must be handled properly. The DRMAA application must be run on a submission host with an enabled Univa Grid Engine environment.

1.7.5.2.1 Compiling and Running the Java Code DRMAA Example

In order to compile a Java DRMAA application the Java `CLASSPATH` variable must point to `$SGE_ROOT/lib/drmaa.jar`. Alternatively the `-cp` or `-classpath` parameter can be passed to the Java compiler at the time of compilation.

```
> javac -cp $SGE_ROOT/lib/drmaa.jar Sample.java
```

To run the application the native code library (`libdrmaa.so`) must be available in the `LD_LIBRARY_PATH` environment variable.

In this example `$SGE_ROOT` is expected to be `/opt/uge800`.

```
> export LD_LIBRARY_PATH=LD_LIBRARY_PATH:/opt/uge800/lib/linux
> java -cp $SGE_ROOT/lib/drmaa.jar:./ Sample
```

1.7.5.2.2 Job Submission, Waiting and Getting the Exit Status of the Job

The following example has the same behavior as the C example in the section above. First a DRMAA job session is created through a factory method (line 19-22). A new session is opened with the `init()` call (line 23). After a job template is allocated (line 26) and the remote command parameter (line 29) and the job argument (line 32) is set accordingly, the wait method does not terminate as long the job runs (line 39). Finally the exit status of the job is checked (line 41-47), the job template is freed (line 50) and the session is closed (line 53).

```
000 import java.util.Collections;
001 import org.ggf.drmaa.*;
002
003 public class Sample {
004
005    public static void main(String[] args) {
006
007       Sample sample = new Sample();
008
009       try {
010          sample.example1();
011       } catch (DrmaaException exception) {
012          /* something went wrong */
013          System.out.println("DRMAA Error: " + exception.getMessage());
014       }
015    }
016
017    public void example1() throws DrmaaException {
018       /* get the class, which is needed for creating a session */
019       SessionFactory factory = SessionFactory.getFactory();
020
021       /* create a new session */
022       Session s = factory.getSession();
023       s.init(null);
024
025       /* create a new job template */
026       JobTemplate jobTemplate = s.createJobTemplate();
027
028       /* set "sample.sh" as job script */
029       jobTemplate.setRemoteCommand("/path/to/your/job.sh");
030
031       /* set an additional argument */
032       jobTemplate.setArgs(Collections.singletonList("myarg"));
033
034       /* submit the job */
035       String jobid = s.runJob(jobTemplate);
036       System.out.println("The job ID is: " + jobid);
037
038       /* wait for the job */
039       JobInfo status = s.wait(jobid, Session.TIMEOUT_WAIT_FOREVER);
040
041       /* check if job exited (and was not aborted) */
042       if (status.hasExited() == true) {
043          System.out.println("The exit code of the job was: "
044                                 + status.getExitStatus());
045       } else {
```

```
         System.out.println("The job didn't finished normally.");
      }

      /* delete the job template */
      s.deleteJobTemplate(jobTemplate);

      /* close DRMAA session */
      s.exit();
   }

}
```

1.7.6 Further Information

Java **DRMAA** related information can be found in the **doc** directory (HTML format). Further information about **DRMAA** specific attributes can be found in the **DRMAA** related **man** pages:

drmaa_allocate_job_template, drmaa_get_next_attr_value, drmaa_misc, drmaa_synchronize, drmaa_attributes, drmaa_get_next_job_id, drmaa_release_attr_names, drmaa_version, drmaa_control, drmaa_get_num_attr_names, drmaa_release_attr_values, drmaa_wait, drmaa_delete_job_template, drmaa_get_num_attr_values, drmaa_release_job_ids, drmaa_wcoredump, drmaa_exit, drmaa_get_num_job_ids, drmaa_run_bulk_jobs, drmaa_wexitstatus, drmaa_get_attribute, drmaa_get_vector_attribute, drmaa_run_job, drmaa_wifaborted, drmaa_get_attribute_names, drmaa_get_vector_attribute_names, drmaa_session, drmaa_wifexited, drmaa_get_contact,drmaa_init, drmaa_set_attribute, drmaa_wifsignaled, drmaa_get_DRMAA_implementation, drmaa_jobcontrol, drmaa_set_vector_attribute, drmaa_wtermsig, drmaa_get_DRM_system, drmaa_job_ps, drmaa_strerror, jsv_script_interface, drmaa_get_next_attr_name, drmaa_jobtemplate, drmaa_submit

1.8 Advanced Concepts

Besides the rich set of basic functionality discussed in the previous sections, Univa Grid Engine offers several more sophisticated concepts at time of job submission and during job execution. This chapter describes such functionality, which becomes important for more advanced users.

1.8.1 Job Dependencies

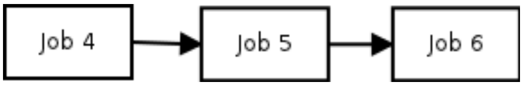
In many cases the jobs, which are submitted with Univa Grid Engine are not self-contained. Those jobs are usually arranged in a kind of workflow with more or less complex job dependencies. Such workflows can be mapped to Univa Grid Engine with the submission parameter `hold_jid <jobid list>`. The `<jobid list>` contains one or a comma separated list of ids of existing jobs of which the submitted job is waiting for before it can be scheduled. In order get the job IDs, submit the jobs with a name (`-N <name>`) and use the name as ID. Alternatively the **qsub** parameter `-terse` can be used, which transforms the command line result of **qsub** so that only the job id is returned. This makes it very simple to use within scripts.

1.8.1.1 Examples

In the following examples, basic workflow control patterns (see www.workflowpatterns.com) are mapped into a Univa Grid Engine job workflow.

1.8.1.1.1 Sequence Pattern

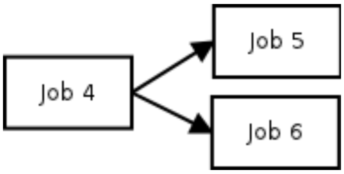
The most simple workflow pattern is the sequence pattern. It is used when a bunch of job must be executed in a pre-defined order. With Univa Grid Engine it is possible to submit all jobs at once but the order is still guaranteed.



```
qsub -b y /bin/sleep 60
Your job 4 ("sleep") has been submitted

qsub -b y -hold_jid 4 /bin/sleep 60
Your job 5 ("sleep") has been submitted

qsub -b y -hold_jid 5 /bin/sleep 60
Your job 6 ("sleep") has been submitted

> qstat
job-ID  prior   name       user         state submit/start
----------------------------------------------------------
      4 0.55500 sleep      daniel       r      03/01/2011 1
      5 0.00000 sleep      daniel       hqw   03/01/2011 15
      6 0.00000 sleep      daniel       hqw   03/01/2011 15
```

1.8.1.1.2 Parallel Split/Fork Pattern

The fork pattern is used when a job sequence involves tasks that are executed in parallel. In this case two or more jobs depend on just one job, meaning they are scheduled after the job is complete. In Univa Grid Engine, this is mapped through setting the hold job ID value of multiple jobs to the same job.



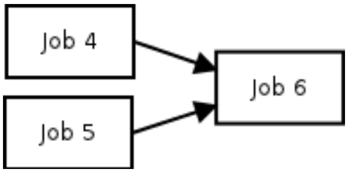
```
qsub -terse -b y /bin/sleep 60
4

qsub -b y -hold_jid 4 /bin/sleep 60
Your job 5 ("sleep") has been submitted

qsub -b y -hold_jid 4 /bin/sleep 60
Your job 6 ("sleep") has been submitted

> qstat
job-ID  prior   name       user         state submit/start at     queue
-----------------------------------------------------------------------
        4 0.55500 sleep      daniel       r     03/01/2011 16:00:50 all
        5 0.00000 sleep      daniel       hqw   03/01/2011 16:00:58
        6 0.00000 sleep      daniel       hqw   03/01/2011 16:01:00

> qstat
job-ID  prior   name       user         state submit/start at     queue
```

```
-------------------------------------------------------------------------------------------
        5 0.00000 sleep      daniel       r   03/01/2011 16:00:58
        6 0.00000 sleep      daniel       r   03/01/2011 16:01:00
```

In this example job 5 and 6 depending from job 4. After job 4 finishes both jobs a

1.8.1.1.3 Synchronization Pattern

With the synchronization pattern, a job starts (is scheduled) when all dependencies are fulfilled, i.e. that all of the waiting jobs have completed. It is usually used after parallel sections induced by the parallel split/fork pattern or when a job is one which finalizes the work of multiple jobs (post processing).



```
qsub -b y /bin/sleep 60
Your job 4 ("sleep") has been submitted

qsub -b y /bin/sleep 120
Your job 5 ("sleep") has been submitted

qsub -b y -hold_jid 4,5 /bin/sleep 60
Your job 6 ("sleep") has been submitted

> qstat
job-ID  prior   name       user         state submit/start at     queue
-------------------------------------------------------------------------------------------
      4 0.55500 sleep      daniel       r      03/01/2011 16:24:50 all.
      5 0.00000 sleep      daniel       r      03/01/2011 16:24:54 all.
      6 0.00000 sleep      daniel       hqw   03/01/2011 16:24:57

> qstat
job-ID  prior   name       user         state submit/start at     queue
-------------------------------------------------------------------------------------------
      5 0.00000 sleep      daniel       r      03/01/2011 16:24:54 all.
      6 0.00000 sleep      daniel       hqw   03/01/2011 16:24:57
```

In this example the job 6 depends on two previously submitted jobs. The hold sta ended.

1.8.2 Using Environment Variables

During job execution a number of environment variables are set from Univa Grid Engine and are available for the executing script/binary. These variables contain information about Univa Grid Engine specific settings, job submission related information and other details. Additionally the user can specify at time of submission using the `-v` and `-V` parameter self-defined environment variables. While `-v` expects a list of `variable=value` pairs, which are passed-through from job submission to the jobs environment, the `-V` parameter transfers **all** environment variables from the job submission context into the jobs execution context.

```
qrsh -v answer=42 myscript.csh
```

In `myscript.csh` `$answer` has the value 42.

```
setenv answer 42
```

```
qrsh -V myscript.csh
```

In `myscript.csh` `$answer` has the value 42.

In the following tables all Univa Grid Engine environment variables available during job execution are listed:

TABLE: Standard Job Environment Variables

Variable Name	Semantic
SGE_ARCH	The architecture of the host on which the job is running.
SGE_BINARY_PATH	The absolute path to the Univa Grid Engine binaries.
SGE_JOB_SPOOL_DIR	The directory where the Univa Grid Engine shepherd stores information about the job.
SGE_JSV_TIMEOUT	Timeout value (in seconds), when the client JSV will be restarted.
SGE_STDERR_PATH	The absolute path to the standard error file, in which Univa Grid Engine writes errors about job execution.
SGE_STDOUT_PATH	The absolute path to the standard output file, in which Univa Grid Engine writes the output of the job.
SGE_STDIN_PATH	The absolute path to file, the job uses as standard input.
ENVIRONMENT	Univa Grid Engine fills in `BATCH` to identify it as an Univa Grid Engine job submitted with qsub.
HOME	Path to the home directory of the user.
HOSTNAME	Name of the host on which the job is running.
JOB_ID	ID of the Univa Grid Engine job.
JOB_NAME	Name of the Univa Grid Engine job.
JOB_SCRIPT	Name of the script, which is currently executed.
LOGNAME	Login name of the user running the job on the execution host.
PATH	The default search path of the job.
QUEUE	The name of the queue in which the job is running.
REQUEST	The name of the job specified with the -N option.
RESTARTED	Indicates if the job was restarted (`1`) or if it is the first run (`0`).
SHELL	The login shell of the user running the job on the execution host.
TMPDIR	The absolute path to the temporary directory on the execution host.
TMP	The absolute path to the temporary directory on the execution host.
TZ	The timezone set from the execution daemon.

USER	The login name of the user running the job.

TABLE: Job Submission Related Job Environment Variables

Variable Name	Semantic
SGE_O_HOME	The home directory on the submission host.
SGE_O_HOST	The name of the host, on which the job is submitted.
SGE_O_LOGNAME	The login name of the job submitter.
SGE_O_MAIL	The mail directory of the job submitter.
SGE_O_PATH	The search path variable of the job submitter.
SGE_O_SHELL	The shell of the job submitter.
SGE_O_TZ	The time zone of the job submitter.
SGE_O_WORKDIR	The working directory path of the job submitter.

TABLE: Parallel Jobs Related Job Environment Variables

NHOSTS	The number of hosts on which this parallel job is executed.
NQUEUES	The number of queues on which this parallel job is executed.
NSLOTS	The number of slots this parallel job uses (1 for serial jobs).
PE	Only available for parallel jobs: The name of the parallel environment in which the job runs.
PE_HOSTFILE	Only available for parallel jobs: The absolute path to the pe_hostfile.

TABLE: Checkpointing Jobs Related Job Environment Variables

SGE_CKPT_ENV	Checkpointing jobs only: Selected checkpointing environment.
SGE_CKPT_DIR	Checkpointing jobs only: Path of the checkpointing interface.

TABLE: Array Jobs Related Job Environment Variables

SGE_TASK_ID	The task number of the array job task the job represents. If the job is not an array task, the variable contains `undefined`.
SGE_TASK_FIRST	The task number of the first array job task. If the job is not an array task, the variable contains `undefined`.
SGE_TASK_LAST	The task number of the last array job task. If the job is not an array task, the variable contains `undefined`.
SGE_TASK_STEPSIZE	Contains the step size of the array job. If the job is not an array task, the variable contains `undefined`.

1.8.3 Using the Job Context

Sometimes it is necessary that a job makes its internal state visible to `qstat`. This can be done with the job execution context. Job context variables can be initially set on job submission time with the `-ac name=value` parameter and altered/added and deleted during run-time with `qalter -ac` or `-dc` switch.

In the following example a job script makes the internal job state visible to the `qstat` client.

The `context_example.sh` job script looks like the following:

```
#!/bin/sh

sleep 15

$SGE_BINARY_PATH/qalter -ac STATE=staging $JOB_ID

sleep 15

$SGE_BINARY_PATH/qalter -ac STATE=running $JOB_ID

sleep 15

$SGE_BINARY_PATH/qalter -ac STATE=finalizing $JOB_ID
```

Now the job with the context `STATE=submitted` is submitted and the context is filtered with the `grep` command every 15 seconds.

```
> qsub -ac STATE=submitted context_example.sh
Your job 4 ("context_example.sh") has been submitted
> qstat -j 4 | grep context
context:                    STATE=submitted
> sleep 15
> qstat -j 4 | grep context
context:                    STATE=staging
> sleep 15
> qstat -j 4 | grep context
context:                    STATE=running
> sleep 15
> qstat -j 4 | grep context
context:                    STATE=finalizing
```

1.8.4 Transferring Data

A common way to transfer input and output data to and from the user application is to use a distributed or network file system like NFS. While this is easy to handle for the user applications, the performance can be a bottleneck, especially when the data is accessed multiple times sequentially. Hence Univa Grid Engine provides interfaces and environment variables for delegated file staging in order to support the user with hookpoints for accessing and transferring data in different ways. In the following section these approaches for transferring user data as well as their advantages and drawbacks are discussed.

1.8.4.1 Transferring Data within the Job Script

While the job script is transferred from the submission host to the execution host, the data the job is working on remains unknown and therefore unreflected by the `qsub` command. If the necessary input and output files are only available through a slow network file system on the execution host, they can be staged in and out from the job itself to the local host. In order to do so, Univa Grid Engine creates a local temporary directory for each job and deletes it automatically after the job ends. The absolute path to this local directory is as `$TMPDIR` environment variable available during job run-time. In the following example an I/O intensive job copies the input data set from the NFS exported home directory of the user to the local directory and the results back to the home directory.

```
#!/bin/sh
...
# copy the data from the exported home directory to the temporary directory
cp ~/files/largedataset.csv $TMPDIR/largedataset.csv
# do data processing
...
# copy results back to user home directory
cp $TMPDIR/results ~/results
```

1.8.4.2 Using Delegated File Staging in DRMAA Applications

The Univa Grid Engine DRMAA implementation comes with built-in support for file staging. The administrator must configure appropriate **prolog** and **epilog** scripts, which are executed before the DRMAA jobs starts and after the DRMAA job ends. Theses scripts can be configured in the global configuration (`qconf -mconf`), in the host configuration (`qconf -mconf hostname`), and in the queue configuration (`qconf -mq queuename`). The script that is executed depends on the scripts which are configured. The host configuration overrides the global configuration and the queue configuration dominates the host configuration.

In order to make the job and epilog script job obvious, a set of variables is defined by Univa Grid Engine. These variables can be used in the configuration line, where the path to the pro- and epilog is defined.

Delegated File Staging Variables

Variable	Semantic
$fs_stdin_file_staging	Indicates if file staging for stdin is turn on (1) or off (0).
$fs_stdout_file_staging	Indicates if file staging for stdout is turn on (1) or off (0).
$fs_stderr_file_staging	Indicates if file staging for stderr is turn on (1) or off (0).
$fs_stdin_host	Name of host where the data originates.
$fs_stdout_host	Name of the host where the data goes.
$fs_stderr_host	Name of the host where the error file goes.
$fs_stdin_path	The absolute path to the input file on the input host.
$fs_stdout_path	The absolute path to the output file on the output host.
$fs_stderr_path	The absolute path to the error file on the output host.
$fs_stdin_tmp_path	The absolute path to the temporary input file.
$fs_stdout_tmp_path	The absolute path to the temporary output file.
$fs_stderr_tmp_path	The absolute path to the temporary error file.

1.8.4.2.1 Example: Copy the DRMAA Job Output File

In the following example an epilog script is parametrized in a way that the DRMAA job output file is copied after the job ends to an user defined host and directory.

```
qconf -mconf
...
epilog /path/to/epilog.sh $fs_stdout_file_staging $fs_stdout_host $fs_stdout_path $fs_stdout_tm
...
```

The `epilog.sh` script looks like the following:

```
#!/bin/sh

doFileStaging=$1
outputHost=$2
outputHostPath=$3
tmpJobPath=$4

if [ "x$doFileStaging" = "x1" ]; then

   # transfer file from execution host to host specified in DRMAA file
   echo "Copy file $tmpJobPath to host $outputHost to $outputHostPath"
   scp $tmpJobPath $outputHost:$outputHostPath

fi
```

Finally the DRMAA delegated file staging must be turned on:

```
qconf -mconf
 ...
 delegated_file_staging true
```

After this is configured by the Univa Grid Engine administrator everything is the prepared from the Univa Grid Engine side. The DRMAA application now has to determine where to copy the information of the output file in the job template. The following code example shows how to accomplish this with Java DRMAA.

```
/* enable transfer output file to host "yourhostname" in file "/tmp/JOBOUTPUT" */
jobTemplate.setOutputPath("yourhostname:/tmp/JOBOUTPUT");
/* disable transfer input file, enable transfer output file, disable transfer error file *
FileTransferMode mode = new FileTransferMode(false, true, false);
jobTemplate.setTransferFiles(mode);
```

Go back to the Univa Grid Engine Documentation main page.