UNIVA

Univa Corporation

Grid Engine Documentation

Grid Engine Introductory Guide

Author:
Univa Engineering

Version:
8.6.3

September 27, 2018

Contents

1 Basic Univa Grid Engine Functionality **1**

1.1 Functionality Overview . . . 1

1.1.1 What is a Resource? . . . 1

1.1.2 What is a Workload? . . . 1

1.1.3 Allocating Workloads to Resources . . . 1

1.1.4 How Policies Govern Univa Grid Engine Behavior . . . 2

1.2 Typical Use Case Scenarios for Univa Grid Engine . . . 2

1.2.1 Running Batch Workloads on a Cluster of Servers . . . 2

1.2.2 Running Parallel Workloads in a Cluster . . . 2

1.2.3 Running Interactive Work on a Pool of Server Resources . . . 3

1.2.4 Repeating a Calculation on a Large Amount of Data . . . 4

1.2.5 Sharing Resources Among Users . . . 4

2 Concepts and Components **5**

2.1 Univa Grid Engine Components and Corresponding Host Types . . . 5

2.1.1 SGE_QMASTER and the Master Host . . . 5

2.1.2 SGE_EXECD and the Execution Hosts . . . 7

2.1.3 Univa Grid Engine Client Commands and Submission/Administration Hosts 8

2.1.4 The Accounting and Reporting Database . . . 9

2.2 What is a Cell? . . . 9

2.3 Types of Workloads being Managed by Grid Engine . . . 10

2.3.1 Batch Jobs . . . 10

2.3.2 Parallel Jobs . . . 10

2.3.3 Interactive Jobs . . . 10

2.3.4 Array Jobs . . . 10

2.3.5 Checkpointing Jobs . . . 11

2.3.6 Immediate Jobs . . . 11

2.4 How Workload Gets Queued in Univa Grid Engine . . . 11

2.5 Univa Grid Engine Queues and Cluster Queues . . . 11

2.5.1 Understanding Queue Subordination . . . 11

2.6 Expressing Capabilities and Capacities . . . 11

2.6.1 What is a *Complex*? 11
2.6.2 What is a *Quota*? 13
2.6.3 What is a *Load Sensor*? 13
2.6.4 What is a *Resource Request*? 15
2.7 Reservations 15
2.7.1 Automatic Resource Reservation 15
2.7.2 Advance Reservations 15
2.8 Determining the Scheduling Order 15
2.8.1 What is *Priority*? 16
2.8.2 What is a *Ticket*? 16
2.8.3 What is an *Urgency*? 17
2.9 Calendar Schedules 17
2.10 Types of Users and User Lists 17
2.10.1 Administrators 18
2.10.2 Operators 18
2.10.3 Users 18
2.10.4 User Groups 18
2.10.5 User Access Lists 18
2.10.6 Departments 18
2.10.7 Projects 18
2.11 Cluster Status Data Spooling Methods 19
2.11.1 Classic Spooling 19
2.11.2 BerkeleyDB Spooling 19

1 Basic Univa Grid Engine Functionality

1.1 Functionality Overview

Univa Grid Engine is commonly referred to as a Distributed Resource Management (DRM) system, a Workload and Resource Management system or as a Workload Scheduler. All of these are arguably descriptive and correct. Univa Grid Engine does indeed manage resources of a distributed nature and the workloads designated to occupy or consume those resources while executing. In doing so, Univa Grid Engine employs advanced policies to determine schedules for allocating workloads to the resources in a near optimal fashion.

1.1.1 What is a Resource?

A *resource* in this context is anything which can be

- defined, e.g. availability of data on a server (*has_data_part_1*) or the location of the machine (*rack-4*)
- counted, e.g. the usage of software licenses
- measured, e.g. the available space on a specific file system (as measured by a script or binary)
- that which the operating system reports, e.g. CPU usage, available memory, I/O usage

This is a versatile platform which accurately models resources in a data center, be they physical resources, virtual resources, static characteristics or other attributes to be managed.

1.1.2 What is a Workload?

Univa Grid Engine supports a broad set of workloads. In general, anything that could be executed from a command-line can be submitted to Univa Grid Engine and it will be executed as a *job* under the control of Univa Grid Engine. Examples for suitable workloads are shell scripts and executable binaries as well as interactive sessions or virtual machines. More introductory information regarding supported workloads can be found in the User's Guide.

Workloads may have requirements which need to be satisfied in order for them to function as desired. Those can include dependencies on certain operating systems and the release or patch level, requirements for available memory, disk space or CPU and core counts. Other requirements can be the availability of software licenses to be utilized as part of an application embedded in the workload or a minimum of free disk space on a specific file system.

These requirements are stated when submitting a workload and will instruct Univa Grid Engine how to handle the corresponding job.

1.1.3 Allocating Workloads to Resources

Univa Grid Engine keeps an inventory of the resources controlled and their status. It knows, for example, the number of servers it controls, the number of CPUs and cores they each have, the amount of memory they have, how those servers are already utilized. Together with the requirements submitted along with the workloads, Univa Grid Engine is able to determine which workloads will fit on which resource or which work will have to get finalized so that resources get freed up which a certain workload requires.

In order to make scheduling decisions like which workload to place where and which workload to give precedence over another there is only one additional thing missing: knowledge about the desired operational behavior as expressed by *policies*.

1.1.4 How Policies Govern Univa Grid Engine Behavior

There is whole host of what Univa Grid Engine calls policies, rule systems and behavioral patterns supported by Univa Grid Engine which allow an administrator to declare to Univa Grid Engine how resources are to be utilized and how workloads are to be prioritized. It is possible, for instance, to control quotas for resource usage broken down by user groups or individual users. Or it is straight forward to make execution order dependent on importance (e.g. as measured by cost) of the resources which a workload will consume.

1.2 Typical Use Case Scenarios for Univa Grid Engine

The following sub-sections briefly introduce a few scenarios in which Univa Grid Engine is typically used.

1.2.1 Running Batch Workloads on a Cluster of Servers

Probably the most simple and most standard use for employing Univa Grid Engine is to utilize a cluster of servers as an opaque resource (for the end users) to process batch jobs. A common approach would be to inform end users submitting jobs about how to build and submit their batch jobs to suit the underlying Univa Grid Engine set-up (as configured by the administrator). End users will then be able to submit any number of jobs to Univa Grid Engine, specifying the requirements of those jobs along with submission.

The jobs would be taken by Univa Grid Engine and either executed immediately on suitable resources or be queued up and executed whenever the required resources become available and when the order of precedence among jobs makes them eligible for execution.

If, as in this scenario, the workloads are batch jobs, i.e. if they do not need any user interaction upon start-up or during execution, then Univa Grid Engine has full flexibility of scheduling those jobs for execution whenever it is optimal following the policy configuration set forth by the administrator. Other job types might restrict what Univa Grid Engine can accomplish. An interactive workload will, for instance, require the presence of the end-user at a terminal to provide the required inputs; thus the workload can only be scheduled immediately or at an agreed upon time.

1.2.2 Running Parallel Workloads in a Cluster

Parallel workloads are programs which are comprised of a set of tasks executing concurrently on separate resources. There are different types of parallel jobs. Most important are two:

- Distributed memory applications: These have their tasks executed only on servers connected to a network so that each task only uses the memory on the server. The application comprised of all the distributed tasks utilizes distributed memory.
- Shared memory applications: The tasks for such applications execute on resources which have access to shared memory. This is most typically the case on a single system with several CPUs or cores but one shared memory. These tasks would each occupy a CPU or core but could access the same memory on the server.

Univa Grid Engine supports all kinds of parallel applications. In particular, it allows end users to specify in generic terms the type of parallel application submitted (e.g. distributed or shared memory) and the number of servers, CPUs or cores those applications will require. It is even possible to specify ranges for the number of servers, CPUs or cores. So a distributed memory application could be submitted along with specifying it can run on four servers but, if available, should be executed using up to 16 servers.

Such parallel job submissions will be handled by Univa Grid Engine in a similar manner to how it was described for batch jobs (parallel jobs often are batch jobs at the same time). Batch jobs are taken by Univa Grid Engine and executed on suitable resources as soon as sufficient resources are available and as soon as a particular job is due to be started.

The use case of parallel jobs is special in that allocating resource concurrently imposes specific requirements on Univa Grid Engine as does ensuring that all tasks executing concurrently are controlled as a single job. So if a parallel job is canceled while it executes then it has to be ensured that all tasks of the parallel job are terminated.

1.2.3 Running Interactive Work on a Pool of Server Resources

Systems like Univa Grid Engine are often used for managing batch workloads and are even referred to as *batch schedulers*. It is, however, entirely feasible and quite beneficial to also run interactive work on top of Univa Grid Engine. In such case the end user would not explicitly select a resource on which to run, log onto that server and start the interactive work but would rather specify the type of system being required to Univa Grid Engine together with the interactive work to be executed (e.g. a GUI or terminal session). Univa Grid Engine would then select an available resource and create an interactive session on the selected server for the user.

Running interactive workloads through Univa Grid Engine offers benefits in the following scenarios:

- Avoiding conflicts or overloaded resources if the same pool or resources is to be used for batch and interactive work
- Optimizing the use of resources and providing reliable service specifically in case of large server and user counts
- Farming out resource intensive work (e.g. the memory intensive web browsing) to a backend cluster while the servers providing terminal and GUI sessions do not get overloaded and thus stay responsive
- Reducing IT cost by more optimally utilizing resources
- Increasing the efficiency of end users by easing interactive access to the resources required for their work
- Provide controlling, monitoring, reporting and accounting for interactive workloads

1.2.4 Repeating a Calculation on a Large Amount of Data

The same calculations often need to be repeated over and over again each time on different input data. This scenario is encountered in parameter studies in which a simulation or analytic computation depends on the variability of many parameters. Another area in which large data sets serve as the input for an ever repeated calculation is data mining, i.e. properties are to be extracted from massive data sets by applying the same algorithms on parts of the data. A case very similar to data mining is found in Bio Technology, where genetic or protein databases are searched and analyzed against probes. Yet another case for repeated calculation over large data sets appears in the animated film creation and computerized special effects industry. In this case, it is the computer model of a scene and the many scenes sequences which make up a film which account for the large quantity of data. In this application, the steps to be repeated are things like object rendering, visibility filtering or shadow models.

1.2.5 Sharing Resources Among Users

One of the main motivations to use a DRM system like Univa Grid Engine is to share a common pool of resources among users and user groups. Here are a few examples for which resources can be shared.

Sharing the Compute Nodes In the vast majority of cases Univa Grid Engine is being used for distributing workloads coming from different users across the compute nodes (i.e. the servers) in a computer network, often also referred to as *cluster*. In managing the distribution of workloads on behalf of those users, Univa Grid Engine effectively enables users to share the nodes in that cluster. They do not need to be consumed with monitoring the nodes and selecting those which are suitable and not occupied by someone else. Univa Grid Engine accomplishes this with full transparency. Users need not worry about a node going down or maintenance downtime on nodes or new nodes being added to the cluster. Univa Grid Engine will make sure all those changes in the cluster environment will be handled appropriately.

Sharing Licenses Univa Grid Engine can be configured to manage almost any kind of resource, a very important example being software licenses. Software licenses can be a scarce resource, e.g. if there are not enough licenses available so that all work being submitted will be able to receive a license, and in many cases it becomes the most expensive and thus most important asset to be managed. Here are few examples for license management with Univa Grid Engine:

- It is possible to manage *node-locked* licenses, i.e. licenses which are bound to a specific compute node in the cluster. In such a case, Univa Grid Engine will make sure that jobs requiring such a license to run will be dispatched to a node with an unused license bound to it.
- It is likewise possible to manage *floating* licenses. These are licenses which can be used on any node in the cluster but for which a quota exists, i.e. the total amount of licenses being used at the same time is restricted to a certain number. Univa Grid Engine can be configured to keep track of the number of licenses being in use and those still available. It will not dispatch jobs which require a license when the floating license quota has been met.

- Many software vendors integrate their software with a license manager as FlexNet Publisher (formerly called FLEXlm). Those license managers are a separate software infrastructure which have the software license entitlements deposited which a site owns. The applications have been integrated with that license manager and *check out* needed licenses during run-time. Univa Grid Engine can also be configured to interact with the license manager and align the workload management with the availability of licenses as controlled by the license manager. I.e. jobs requiring a license will only get dispatched if the license manager has a corresponding license available.

Sharing Storage Space In analogy to sharing compute nodes or software license, Univa Grid Engine can also enable users and the jobs they are submitting to share resources such as storage space. A typical example is scratch space on a network file system. Some applications require large amounts of temporary file space (also known as scratch space) while running. That file space will no longer needed after the applications has finished. So it makes sense for a site to set aside a certain amount of scratch space to be shared - not too much to be cost effective but just enough for an average amount of such applications to be able to run simultaneously. Univa Grid Engine can monitor the capacity of resources like file space and can ensure that applications only get started if there is enough space available for them to function properly.

2 Concepts and Components

2.1 Univa Grid Engine Components and Corresponding Host Types

The Univa Grid Engine system is comprised of a set of components, i.e. daemons and client commands. The following sections discuss those components, their role in the Univa Grid Engine system and specific characteristics which need to apply to hosts running those components. As a quick overview, the following categories for components exist:

- Central control components, such as the *qmaster* and the *scheduler*
- Execution agents
- Client commands, such as the command-line interface facilities and the graphical user interfaces

2.1.1 SGE_QMASTER and the Master Host

The *qmaster* is the central control and information point of a Univa Grid Engine installation. The name of the qmaster executable is `sge_qmaster` and it runs as a daemon process once executed. The qmaster keeps track of all status information in a Univa Grid Engine cluster. For example, it knows the load situation of all hosts under Univa Grid Engine control, the jobs they execute, the jobs awaiting dispatch, about all users registered with the system and their roles, about the policies and other configurations which the Univa Grid Engine administrators have set up and about any other status information that has relevance in Univa Grid Engine. The qmaster is also in touch with all other components in a Univa Grid Engine system; it maintains regular exchanges with the execution agents and it is contacted by all client commands for whatever information to be retrieved or tasks to be executed.

To ensure that a qmaster can be stopped and restarted without loss of information, data is saved to a file system or into the compiled-in BerkeleyDB which then stores the data onto disk storage. Even if the qmaster is down, jobs will be executed in a Univa Grid Engine system. New jobs cannot be started and client commands will not be able to run successfully.

It is crucial that the qmaster daemon is running and able to respond to requests from the execution agents or the client commands quickly. It the qmaster is down, stalled or drastically slowed down then the Univa Grid Engine will not work or will exhibit poor performance. Conversely, poor performance is usually a sign that qmaster performance is encumbered.

It is therefore most important that the host on which the qmaster is running is suitably powerful. This has generally four aspects:

- The qmaster host needs to have enough processing power. State-of-the-art CPUs with four or more cores are commonly sufficient even for large installations. For small installations (a few hundred execution nodes, a few thousand jobs in the system at any point in time) a less powerful host may suffice.
- The qmaster host needs to have sufficient memory. This is highly dependent on the number of jobs waiting to be dispatched at any one time in the systems. For large jobs counts (in the hundred thousands of jobs) 16 Gbytes are advisable.
- The qmaster hosts needs to have fast access to sufficient file space. Since the qmaster has to process all status changes immediately (and there can be many per second in a large and busy cluster), limitations in file access speed can greatly impede the performance of Univa Grid Engine. The best possible file access path should be provided to the qmaster, even more so if the installation is large with many execution hosts and jobs.
- The qmaster needs good network connectivity to all other components. While the overall performance will not be influenced much by a slow link to a particular execution or client machine or even the total failure of network connectivity to such a machine, it is still important that the qmaster can interact quickly with other components. The amount of data being commonly transferred is relatively small, however, so low latency is much more important than high bandwidth. The qmaster requires good connectivity but generally does not require an ultra high speed network (like InfiniBand).

Setting up the qmaster on a suitable machine with the above considerations in mind is key to avoid performance problems.

The following subsection will discuss how to ensure that a failure of the qmaster machine will not halt the operation of a Univa Grid Engine cluster.

SGE_SHADOWD and the Shadow Master Hosts Univa Grid Engine plays a crucial role in a workload processing environment. If Univa Grid Engine were to fail, servers would then become idle thereby rendering expensive equipment useless. And within Univa Grid Engine, the qmaster is a single point of failure. Jobs can be submitted, dispatched, started, monitored and controlled only if qmaster is running. Thus making sure that qmaster is always running is important for a production-grade Univa Grid Engine system.

There are, in essence, two ways to achieve this:

- Integrating the qmaster machine and the qmaster daemon into a high availability (HA) solution. This will result in a fail-over qmaster machine and some qmaster failure detection conditions which are monitored by the HA solution. This should also include the file system being used by qmaster for persisting out status data (see *Installation Guide -> Selecting a File System for Persistency Spooling of Status Data* for information on status information spooling) in the HA configuration so it fails over together with qmaster.
- If the file service used for spooling is available or is considered reasonably reliable, then use the shadow master facility. It is provided by the *shadow master daemon* sge_shadowd. The shadow daemon runs on a different host than qmaster and monitors whether qmaster is alive. If the shadow daemon detects qmaster's failure then it invokes a new qmaster which takes over the cluster operation based on the status information in the spooling data. There can be more than one shadow daemon on different shadow master hosts. In such a scenario one of the shadow masters will take over the qmaster role (see *Administrator's Guide -> Special Activities -> Ensuring High Availability* for more information).

The Scheduler Thread The qmaster also provides the decision making function in an Univa Grid Engine cluster. These are for instance

- matching of resources with job requirements
- planning of resource assignment schedules
- job priority management
- user dependent resource entitlement control
- resource reservations
- job placement decisions

This functionality is clearly separated from the rest of qmaster's tasks (configuration management, status monitoring, reporting, etc) in the scheduler thread. The operation of the scheduler thread is normally opaque to the user and even the Univa Grid Engine administrator. The scheduler thread interacts with the qmaster by receiving cluster status and configuration updates and sending back job dispatching decision. There are, however a few administrative commands which allow the administrator to analyze the scheduler thread's decision making. An example is the `qconf -sss` ("show scheduler status") command.

2.1.2 SGE_EXECD and the Execution Hosts

The execution hosts are the nodes in a Univa Grid Engine cluster on which workloads (jobs, interactive sessions, etc) are being executed by Univa Grid Engine. The component which identifies a node as execution host and which allows Univa Grid Engine to utilize it is the *execution daemon* (sge_execd). A host may be declared as an execution host in a Univa Grid Engine cluster but it cannot act as such unless a sge_execd is running on it. The execution daemon provides the following functionalities:

- It communicates with the qmaster in both directions:
  - It reports the resource utilization status (CPU load, memory usage, etc) as well as status and resource usage of running jobs and finished jobs on a specific node

  - It receives tasks to execute any type of workload which Univa Grid Engine support from qmaster. There are also configuration options which will result in qmaster sending requests to re-prioritize running jobs from time to time.
- The sge_execd hands off the starting of workloads and the low level process control to a component called *shepherd* (sge_shepherd). There is one sge_shepherd per job. So one sge_execd utilizes and controls several sge_shepherds if a node runs multiple jobs at a time simultaneously.

Execution hosts do not solely have to be dedicated to Univa Grid Engine. It is possible to also run other workloads (e.g. work started by users manually) on those nodes. The sge_execd monitors all loads, regardless of whether initiated by Univa Grid Engine or otherwise which the qmaster and sge_schedd can take into account. In high throughput clusters or use cases with parallel jobs it is however advisable to designate execution hosts. Other workload will otherwise interfere and affect the efficiency of the cluster.

By the same token is it also possible to use the host on which the qmaster runs as an execution host. But also this is only advisable for small installations, e.g. for tests. The qmaster is a crucial component in a cluster and should mostly run on a designated machine. And again, the activity of the qmaster would also interfere with the workloads on that host if it were used as an execution host also. An exception is probably if the jobs to be executed on the qmaster host are administrative Univa Grid Engine tasks.

Finally, execution hosts can also be submission and administrative hosts (see below). This is necessary even if jobs are executing corresponding Univa Grid Engine client commands.

2.1.3 Univa Grid Engine Client Commands and Submission/Administration Hosts

Users and administrators (see *Introduction -> Concepts and Components -> Types of Users and User Lists* for information on different user type) interact with Univa Grid Engine via a set of client commands. There are client commands which are restricted for administrative tasks, i.e. which can only be executed by users in an administrative role, and commands which are utilized by users as well as administrators. Hosts need to be declared to have permission to run Univa Grid Engine user or administrative commands, otherwise the qmaster will reject a request from a client command running on those nodes.

A host having the permission to run client commands for the regular user such as submitting, monitoring or controlling jobs is called a submission host. A host allowed to run administrative commands, such as modifying the cluster configuration is called an administrative host. An administrative host is automatically also a submission host.

Below is a short breakdown of the client commands being available.

Command Line Clients There is a set of client commands to be run from the command line. There are commands available targeting the end user and other commands being mainly used by administrators. Here are a few examples:

- `qsub/qrsh/qsh/qlogin` - these are all commands which submit workload (batch, interactive, parallel) into a Univa Grid Engine cluster

- `qstat/qhost` - commands to retrieve information about the status of jobs, hosts or the cluster and its components
- `qalter/qmod` - commands to modify jobs, either their characteristics or their status (e.g. suspending a job)
- `qdel` - a command to delete jobs (terminate running job or cancel submitted but not yet executing jobs)
- `qacct` - a command to generate resource usage reports for finished jobs and accounting reports for users or user groups
- `qconf` - the main administrative command line interface allowing display and modification of the Univa Grid Engine configuration

Graphical User Interfaces There is also a graphical user interface called `qmon`. Qmon provides access to the same functions as the command line clients described above. So it fulfills a mix of administrative and end-user targeting roles.

Qmon will be replaced by new graphical user interface components. Therefore usage of it is deprecated and the qmon functionality is not described as part of this documentation set. Refer to the online help within qmon for explanations on how to use the various dialogs.

2.1.4 The Accounting and Reporting Database

Univa Grid Engine collects ample data about jobs while in execution and after completion. This data is recorded for later review by the user or administrator and for creating cluster usage statistics or for accounting reports for users and user groups. There are two processes by which such data is recorded and there are two access methods to the resulting data.

The first step is to record data of finished jobs in the accounting file. That file can be accessed with the `qacct` command line client to run simple reports or administrators may choose to access and parse the file directly.

A more comprehensive set of data is collected into the *accounting and reporting database*. This does not only include the resource usage of finished jobs but also resource utilization snapshots of running jobs and other cluster statistics. That is written out by qmaster into another file and then fed into a relational database (Oracle, MySQL or PostgreSQL). From there it is possible to use SQL analysis methods. One specifically tuned for the Univa Grid Engine accounting an reporting data is the Univa product UniSight. It provides a straight forward means to generate reports from Univa Grid Engine accounting data and for exporting such reports into various formats (such as *comma separated values*, also known as CSV).

2.2 What is a Cell?

Sometimes it is necessary for sites to completely separate the management of different workloads. Such a case could be to operate a pre-staging test cluster in which new configurations or new types of jobs are being tested before they get moved over into a completely separate productions cluster. For such and similar cases, Univa Grid Engine provides a means to use the same installation repository (i.e. the same file space where binaries and other installation package components reside) while keeping the configurations and operations separated.

This is accomplished by the *Cell* concept. A Univa Grid Engine cell simply refers to the name of the root of the configuration data directory hierarchy. Univa Grid Engine commands know which cell they belong to through the environment variable SGE_CELL. It tells the qmaster that it is to manage the corresponding cell and read its configuration when qmaster is started and it will also tell client commands which qmaster they are to respond to.

2.3 Types of Workloads being Managed by Grid Engine

2.3.1 Batch Jobs

A batch job is a shell script which lists UNIX commands and application calls. Those calls are executed in a sequential manner on the execution host as if the call were invoked by the submitting user locally.

Example of a batch job:

```
#!/bin/sh

# redirect the output-file of the batch job
#$ -o /tmp/output.$JOB_ID

# Call UNIX command 'hostname'
hostname
```

For more information on how to write shell scripts and submit batch jobs consult the man-page submit(1).

2.3.2 Parallel Jobs

Univa Grid Engine supports the possibility to handle and submit jobs for parallel and distributed processing. Applications which uses message-passing environments like Parallel Virtual Machine (PVM) or Message Passing Interface (MPI) are also supported like shared memory parallel programs on multiple slots across execution hosts for distributed memory parallel jobs.

2.3.3 Interactive Jobs

Univa Grid Engine provides the facility to execute interactive applications or batch jobs which supports terminal I/O (standard output, standard input, standard error). Interactive jobs, unlike batch jobs, supplies the possibility to interact with the submitted job and it transfers the output directly to the user terminal.

2.3.4 Array Jobs

An array job divides a batch job (shell script) into a certain number of tasks. Those tasks are distributed by the Univa Grid Engine in the cluster. Each of these tasks occupies a slot on an execution host. This is useful, for example, when there is a large amount of data which can be divided into sub-tasks and should be processed with the same application or script.

```
#!/bin/sh

# redirect the output-file of the batch job
#$ -o /tmp/array.$JOB_ID
# This job is divided in 10 tasks.
#$ -t 1-10

# The data_processor application is started ten times with consecutively numbered
#     input files which contains the divided input data.
/tmp/data_processor -i /tmp/array_input.$SGE_TASK_ID
```

2.3.5 Checkpointing Jobs

Jobs which support Checkpointing store the completed current application state which enables the Univa Grid Engine to restore and restart this job from the last checkpoint in case the job was halted or aborted. This means that work which is already done is not lost. This technique is used to provide more fault tolerance and to increase the flexibility of the Univa Grid Engine system accordingly. (e.g. job migration and load balancing mechanisms).

2.3.6 Immediate Jobs

Typically, jobs are not started immediately but rather are queued into the pending queue. They are started as soon as the scheduler dispatches them to an appropriate execution host. Immediate jobs, in contrast, are dispatched straight away to an execution host. If this is not possible, as in the case when there is no free slot available, the job will fail.

For more information how to start a job immediately consult the man-page submit(1).

2.4 How Workload Gets Queued in Univa Grid Engine

2.5 Univa Grid Engine Queues and Cluster Queues

2.5.1 Understanding Queue Subordination

2.6 Expressing Capabilities and Capacities

2.6.1 What is a *Complex*?

Complexes reflect resource attributes which can be requested for a job by the user via the **-l** option of e.g. qsub and qrsh and resources that reflect the load of e.g. a host. A complex also defines how Univa Grid Engine should handle and interpret these resource attributes in its scheduling and dispatching of the pending jobs. There are many of predefined complex entries like *qname* or *hostname*. Consult the man-page **complex(5)** for a list of all predefined complexes.

A complex resource attribute consists of the following attributes:

Attribute	Description
name	The name of the complex object like *hostname*
shortcut	A shortcut for the complex object which can be used in the *-l* parameter call like *h* for *hostname*
type	The type of the complex object. The type defines which values are allowed. ***Available types of complex resource attributes*** **INT** - With *INT* only raw integers are allowed as values. **DOUBLE** - Only floating point values in double precision are allowed. **TIME** - Only a valid time specifier is allowed. Consult *queue_conf(5)* for a format description. **MEMORY** - Only memory size specifiers are allowed. Consult *queue_conf(5)* for a format description. **BOOL** - Only the strings **TRUE** or **FALSE** are allowed. **STRING** - All strings are allowed. The strings are used for wildcard regular boolean expression matching. **CSTRING** - Like *STRING* except that the comparison is case insensitive. **RESTRING** - Like *STRING* but will be deprecated in future. **HOST** - Like *STRING* but only valid hostnames are allowed. **RSMAP** - Like a list of strings (ids) where the amount of ids can be requested like an *INT*. Chosen ids are attached to a job.
relop	The relation operator defines how Univa Grid Engine has to compare the by the user requested value to the corresponding value accounted by Univa Grid Engine. If the result of the comparison is false, the job will not be scheduled. Valid operators are: ==, <, >, <=, >= and **EXCL**. In case of *STRING-alike* type only == is allowed. **EXCL** is only allowed for **BOOL** See *Administrator's Guide -> Special Activities -> Setting Up Nodes for Exclusive Use* for detailed description how to use the **EXCL** operator.
requestable	If *requestable* is set to *yes*, a user is able to request this complex via the *-l* parameter of e.g. qsub. Accordingly, if set to *no*, a user is not able to request this complex. If set to *forced*, a user has to request this complex or the job is rejected. See *complex(5)* for how to enable resource request enforcement.

Attribute	Description
consumable	*consumable* can be set either to *yes*, *no*, 'job_ or *host. yes* and *job* are only valid as parameter if the type of the complex is *numeric* (e.g. *INT*). If set to *yes* or *job* the consumption of the corresponding complex is handled by the bookkeeping of Univa Grid Engine and accordingly a job will not be scheduled if not enough of the requested resources are available. If set to *yes* Univa Grid Engine debits the requested consumable per used slot and if set to *job* Univa Grid Engine will debit per job. *host* (available since 8.1.3) is only allowed to be set for a *RSMAP* complex. The requested value is decremented once on each host the job runs. The amount does not depend on the granted amount of slots.
default	Only meaningful if the complex is a consumable. The value set as default be consumed by every running job. Can be overwritten by the *-l* parameter.
urgency	With the *urgency* it is possible to influence the priority of job per resource base. Consult *complex(5)* and *SGE_priority(5)* for more information.

Example:

```
#qconf -sc
#name          shortcut   type        relop requestable consumable default  urgency
#----------------------------------------------------------------------------------
arch           a          RESTRING    ==    YES         NO         NONE     0
calendar       c          RESTRING    ==    YES         NO         NONE     0
cpu            cpu        DOUBLE      >=    YES         NO         0        0
...
```

See *Administrator's Guide -> Special Activities -> Using Consumables* for more information and how to use them.

2.6.2 What is a *Quota*?

With a *resource quota* it is possible to set limits for the consumption of resources of any job requests. For example, this is useful in preventing single users to fill or overload the whole cluster. Those resource quotas are defined in **resource quota sets** or simply **RQS**.

See Administrator's Guide -> Special Activities -> Using Resource Quota Sets_ how to use such resource quota sets.

2.6.3 What is a *Load Sensor*?

A *Load Sensor* is an executable (script or binary) that periodically reports the load of one or more predefined resources to the execution daemon of the executing host.

Before the Load Sensor is run, any reported resource must be defined as a complex (see *Introduction -> Concepts and Components -> What is a Complex?*). The Load Sensor will be started automatically during start-up of the execution daemon or, if added later to an already-running execution daemon, after a few load-report intervals. A Load Sensor must fulfill a predefined set of rules, e.g. how it reports the load to the execution daemon. See *SGE*execd(8)_ for those rules. More examples are available in the *$SGE*ROOT/util/resources/loadsensors_ directory.

It is possible to configure Load Sensors globally or per-host. For either, use qconf and change the *load*sensor_ line to contain the path of the Load Sensor executable. Configuring globally means that each execution daemon (on every execution host) will run its own instance of the sensor and report the load. Alternatively, the Load Sensor may be configured to run on a per-host basis, where only the execution daemon on that particular host will run the specified Load Sensor.

Configure globally with:

```
# qconf -mconf
```

Configure per-host with:

```
# qconf -mconf <hostname>
```

Example of a load-sensor script which reports the number of running processes on a host:

```
#!/bin/sh

myhost=`uname -n`

while [ 1 ]; do
      # wait for input
      read input
      result=$?
      if [ $result != 0 ]; then
           exit 1
      fi
      if [ "$input" = quit ]; then
           exit 0
      fi
      #send number of running processes
      processes=`ps -elf  | wc -l`
      echo begin
      echo "$myhost:processes:$processes"
      echo end
done

exit 0
```

2.6.4 What is a *Resource Request*?

It is possible to request special resources when submitting a job e.g. the job needs a host which has at least 512 MB main memory. Those resources need to be predefined as complex with the attribute *requestable* (see *Introduction -> Concepts and Components -> What is a Complex?). A resource request can be stated at time of submission via the* -l_ operator of e.g. *qsub* or *qrsh* or directly within the job-script.

Univa Grid Engine even provides the possibility to request resources on an *if available*-basis. If the request cannot be fulfilled, proceed to run the job on a host/queue which does not provide this resource. These are known as **soft-requests**.

Example: Submit a job which has to run on a host with at least 512 MB main memory. At best, Univa Grid Engine should dispatch the job to a host which has also at lest 256 MB free main memory.

```
# qsub -l mt=512MB -soft -l mf=256MB job_script.sh
```

2.7 Reservations

2.7.1 Automatic Resource Reservation

A job is submitted each time as either an explicit or implicit resource request. The Univa Grid Engine scheduler-component will **reserve** all needed resources automatically for pending jobs and prioritize them as lower priority or overlapping resource requests.

2.7.2 Advance Reservations

Advance reservation is a resource reservation completely independent of a particular job and can be requested by a user or administrator and gets created by the Univa Grid Engine system. An advance reservation causes the requested resources to be blocked for other jobs that are submitted subsequently.

See also *User Guide -> Reservations* and _Administrator's Guide -> Special Activities -> Enabling Reservations .

2.8 Determining the Scheduling Order

Univa Grid Engine determines a rank order for the jobs which are waiting to be scheduled (i.e. assigned to resources for execution). Conceptually, the Univa Grid Engine scheduler will look into jobs in order of priority and try to find suitable resources for them. Given that resources (such as CPU, memory or other resources) are available, the scheduler will then reserve those resources for the job before considering the next job. Thus jobs with higher priority have greater chances to get the required resources and will likely be started sooner.

One of the main tasks of a Univa Grid Engine administrator is therefore to control the rank order of the jobs. While there are ways to directly set priorities for jobs, controlling the rank order is mainly performed by defining priority related policies which will automatically influence the order of jobs. Job priorities will change dynamically based on altered conditions. They also will change if the administrator modifies an underlying policy.

If an administrator chooses to not define the policies which will assign priorities for jobs then the order in which jobs are scheduled will be determined by submission order. The first jobs being submitted will then be at the top of the pending job list, i.e. a first-in-first-out (FIFO) scheduling order will be used.

Here is now an overview on the key influence factors, i.e. the main policies, which control the priority assignments for jobs.

2.8.1 What is *Priority*?

The usage of the term *priority* in the above introduction is generic. *Priority* is used to be a qualifier for the rank order in which jobs are waiting to be scheduled. There is, however, a more specific definition of *priority* in Univa Grid Engine. It refers to the *POSIX priority* which is either implicitly or explicitly assigned to a job. The *POSIX priority* is simply a number in the range between -1023 (lowest priority) and 1024 (highest priority). It can be assigned to jobs for instance with the `qsub -p <prio>` or `qalter -p <prio>` command options. If no explicit assignment is being made then the default value of `0` is assigned implicitly.

The *POSIX priority* is primarily a means for users to rank the jobs submitted. If a user has submitted two jobs for example and wants to make sure one receives higher priority independent of the order of submission, then the user could leave the priority of the more important job at the default of `0` and lower the priority of the other job to, let's say, `-10`. This "trick" of lowering the priority for the less important job is used because regular users (as opposed to Univa Grid Engine administrators) can only set priority values of 0 and lower. This is to prevent every user requesting `1024` for their own jobs. However, with a default priority of `0`, users may only lower the priorities of their own jobs.

It is, therefore, in good practice to change that default value for the *POSIX priority*, e.g. to `-100`. See the *SGE_request(5)* manual page for information on how to accomplish this via a system-wide default request file and embedding a `-p -100` setting into it.

Another use case for the *POSIX priority* is to supply the administrator with the means to manually control the rank order of individual jobs. The administrator can set the priority to any value in the POSIX range so if a particular job is very important and needs to be pushed to the top of the waiting list then assigning as priority of `1024` is one way to achieve that.

Note
Assigning priorities manually can quickly become complicated, and it's also quite inflexible. For this reason, the *POSIX priority* is commonly only used as a means for exceptional overrides.

2.8.2 What is a *Ticket*?

Univa Grid Engine provides a collection of policies controlling the importance of jobs which all are based upon the notion of a *ticket*. A *ticket* can be viewed as a currency - more "tickets" equals more "rights" for a specific job. These "rights" are commonly referred to as *entitlements* in Univa Grid Engine. See below for an explanation of *entitlement*.

Tickets are automatically assigned and modified (at least for the most part) and that they not only influence the scheduling order of a job but also determines the resource access which a running job receives , in particularly if that job is competing for the same resource(s) as other jobs. An example can be the portion of a CPU which a job gets if two jobs are running simultaneously.

The ticket value assigned to a job gets is derived from several sources. These are the aforementioned ticket-based policies. A job may not be the most important job in each of those policies but the combined ticket number derived from all policies can well make it the most important job in the system. There is a weighting functionality which controls how much influence each of the policies can exercise and it is, of course, possible to use policies selectively, i.e. not all or for only a part of the jobs.

The following are the policies being based upon the concept of tickets:

The Fair-Share Policy This ensures that individual users, groups of users or projects get *fair* and equitable access to resources in the cluster. A group of users, e.g. a department, may contribute to 30% of the funding of a cluster. Then that user group should also be entitled to consume a **share** of 30% of the cluster resources, at least on average and balanced over a certain period of time. So for this policy it is important to compare **entitlements** with actual past **usage**.

The Functional Share Policy It is also a means to ensure equitable resource access is being granted to user groups. It differs from the fair-share policy in that resource entitlements are enforced independently of past usage. So there is no percentage of average resource usage to be achieved over time but rather a portion of the resources which is owned by a user group at any time unless the users don't utilize those resources (in which case the available resources are split among other user groups according to their entitlements).

The Deadline Policy This policy provides a steady increase in tickets for jobs being submitted with a deadline. The idea is that jobs nearing deadline will receive preferential treatment by Univa Grid Engine, i.e. they are more likely to receive the required resources, thus be started sooner and receive more of the resources available.

2.8.3 What is an *Urgency*?

2.9 Calendar Schedules

2.10 Types of Users and User Lists

There are a number of different user roles in Univa Grid Engine. Part of this discussion can be found in the section about administrative and submission hosts There are also different means to group users and thus manage their rights or entitlements. A more complete overview regarding user roles and groups of users can be found below.

2.10.1 Administrators

An administrator account has full rights in a Univa Grid Engine cluster. There can be multiple administrators. The `root` account on the qmaster machine has administrative rights by default. All other administrative accounts need to be given that role explicitly by an already registered Univa Grid Engine administrator. Administrators can completely reconfigure a cluster or manipulate and inspect all jobs.

2.10.2 Operators

Operators also can execute administrative tasks but only those which keep the cluster operational. They can suspend or resume jobs, for example, but are unable to modify the configuration. Accounts need to receive operator privileges by an Univa Grid Engine administrator.

2.10.3 Users

Users need to be given permission to access a Univa Grid Engine cluster, again by administrators. Users then can submit workloads into Univa Grid Engine, control their own jobs (e.g. modify, suspend, resume or delete them) and monitor the status of the cluster and of jobs (of all jobs). Regular users cannot modify the cluster configuration and manipulate the jobs of other users.

2.10.4 User Groups

Users can be grouped together by the administrator for easier configuration. Such user groups can then be used for instance to define user access lists, departments or projects (see below).

2.10.5 User Access Lists

Lists of users can be given permission or denied to utilize certain parts of the Univa Grid Engine system. There are positive and negative lists, i.e. lists which contain those with access and lists which contain those who are denied access. It's usually only necessary to use only one list, depending on what requires less effort to define.

Some of the functions which are controlled by access lists are accessible to hosts or classes of workloads.

2.10.6 Departments

Departments are groups of users that conceptually belong to the same organizational unit. Departments are specifically used in the context of Univa Grid Engine policies which control resource entitlements across such organizational units.

2.10.7 Projects

Projects are similar to departments (see above), and are meant for finer grained resource entitlement control and less so along organizational boundaries.

2.11 Cluster Status Data Spooling Methods

This section describes characteristics and implications of the different status or data receiving methods being supported.

2.11.1 Classic Spooling

Backing Up and Restoring the Classic Spooling Data *Might want to move this subsection to Tasks in Admin Guide*

2.11.2 BerkeleyDB Spooling

Backing Up and Restoring the BerkeleyDB Spooling Data