NASA CONTRACTOR REPORT 177433

The Network Queueing System

B.K. Kingsbury

CONTRACT NAS2-11786
December 1986

NASA

NASA CONTRACTOR REPORT 177433

The Network Queueing System

B. K. Kingsbury
Sterling Software
Palo Alto, CA

Prepared for
Ames Research Center
under Contract NAS2-11786
December 1986

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

The
Network Queueing System

*Brent A. Kingsbury**

Sterling Software
1121 San Antonio Road, Palo Alto 94303

ABSTRACT

This paper describes the implementation of a networked, UNIX based queueing system developed for a government contract with the National Aeronautics and Space Administration (NASA). The system discussed supports both batch and device requests, and provides the facilities of remote queueing, request routing, remote status, queue access controls, batch request resource quota limits, and remote output return.

1. Origins

The invention of the *Network Queueing System* (NQS) was driven by the need for a good UNIX batch and device queueing facility capable of supporting such requests in a networked environment of UNIX machines. More specifically, NQS was developed as part of an effort aimed at tying together a diverse assortment of UNIX based machines into a useful computational complex for the National Aeronautics and Space Administration (NASA).

Today, this computational complex is officially known as the *Numerical Aerodynamic Simulator Processing System Network*, otherwise known as the NPSN. The assorted machines in this network are of varying manufacture, and (as of the time of this writing) include Digital Equipment Corporation VAXes, Silicon Graphics Irises, large Amdahl 5840 mainframes, and a Cray Research Incorporated CRAY-2. Each of the machines in the network runs its own vendor-supplied version of the UNIX operating system, with appropriate kernel and user-space extensions as necessary.

The presence of UNIX on all of these machines has made possible the creation of a common user interface, so that despite the obvious hardware differences, users can freely move among the different machines of the NPSN without being confronted with entirely different software environments. As part of this common user interface, NQS has been implemented as a collection of *user-space* programs providing the required batch and device queueing capabilities for each machine in the network.

2. Design Goals

NQS was architected and written with the following design goals in mind:

- Provide for the full support of both *batch* and *device* requests. A *batch request* is defined as a shell script containing commands not requiring the direct services of some physical device (other than the CPU resource), that can be executed independently of any user intervention by the invocation of an appropriate command interpreter (e.g. /bin/csh, /bin/sh). In contrast, a *device request* is defined as a set of independent instructions requiring the direct services of a specific device for execution (e.g. a line printer request).
- Support all of the resource quotas enforceable by the underlying UNIX kernel implementation that are relevant to any particular batch request, and its corresponding batch queue.

---

* The author is presently an employee of Cray Research Incorporated.

- Support the remote queueing and routing of batch and device requests throughout the network of machines running NQS. This means that some mechanism must exist to reliably transport batch and device requests between distinct machines, even if one or both of the machines involved crash repeatedly during the transaction.
- Modularize all of the request scheduling algorithms so that the NQS request schedulers can be easily modified on an installation by installation basis, if necessary.
- Support queue access restrictions whereby the right to submit a batch or device request to a particular queue can be controlled, in the form of a user and group access list for any queue.
- Support networked output return, whereby the *stdout* and *stderr* files of any batch request can be returned to a possibly remote machine.
- Allow for the mapping of accounts across machine boundaries. Thus, the account: winston on the machine called: Amelia might be mapped to the account: chandra on the machine called: Orville.
- Provide a friendly mechanism whereby the NQS configuration on any particular machine can be modified without having to resort to the editing of obscure configuration files.
- Support status operations across the network so that a user on one machine can obtain information relevant to NQS on another machine, without requiring the user to log in on the target remote machine.
- Provide a design for the future implementation of file staging, whereby several files or file hierarchies can be *staged* in or out of the machine that eventually executes a particular batch request. For files being *staged-in*, this implies that a *copy* of the file must be constructed on the execution machine, prior to the execution of the batch request. Such files must then be deleted upon the completion of the batch request. For files being *staged-out*, this implies the actual movement of the file from the *execution* machine, to the eventual destination machine.

3. Implementation Strategies

Before dashing off to implement NQS completely from scratch, a long look was taken at an already existing UNIX queueing system known as the *Multiple Device Queueing System* (MDQS), as developed at the U.S. Army Ballistic Research Laboratory. [1]

At one point, it was even decided that NQS could be implemented as an enhanced version of MDQS, borrowing heavily from the original MDQS source code. Theoretically at least, this strategy was supposed to reduce the work and risk involved in building a networked queueing system that would satisfy NASA's needs. This thinking lasted long enough for an early design document to be written detailing the modifications to be made under such a plan.

The plan however was later abandoned, when it was recognized that the new code required for the proposed extensions exceeded the size of the already existing MDQS code. Rather than heap unwieldy extensions upon a frame never designed for such weight, NQS was built completely from scratch. This new strategy allowed for the construction of a new framework from which to hang new ideas, along with many of the concepts included in MDQS. NQS is therefore something old, and something new.

4. The NQS Landscape

This section of the paper describes the general design and concepts of NQS. It must be understood that NQS continues to be developed. This paper discusses only the current state of affairs, with occasional pointers referencing future areas of improvement.

4.1 The Queue and Request Model

In order to provide support for the two request types of *batch* and *device*, NQS implements two distinctly different queue types, with the respective type names of *batch* and *device*. Only *batch*

queues are allowed to accept and execute *batch requests*. Similarly, *device queues* are only allowed to accept and execute *device requests*.

In addition to the first two queue types, a third queue type known as a *pipe queue* exists to transport requests to other batch, device, or pipe queues at possibly remote machine destinations. Readers familiar with MDQS will note that the implementation of three distinctly different queue types differs substantially from the MDQS philosophy of having only one queue type.

4.1.1 Batch Queues

The first queue type implemented in NQS is called a *batch queue*. As stated earlier, NQS batch queues are specifically implemented to run only *batch requests*.

4.1.1.1 Batch Queue Quota Limits

It is useful to be able to place limits on the amounts of different resources that a batch request can consume during execution. Towards that end, NQS batch queues have an associated set of resource quota limits, that all other NQS queue types lack.

For a batch request to be queued in a particular batch queue, any resource quota limits defined by the request must be *less than or equal to* the corresponding limit as defined for the target batch queue. If a batch request fails to specify a particular resource limit value for which a limit is enforceable by the underlying UNIX implementation, then the queued batch request inherits the corresponding limit as defined for the target batch queue.

If a resource limit associated with a batch queue is later lowered by a system administrator, then all requests residing in the queue with a quota limit *greater* than the new corresponding quota limit, are given a *grandfather* clause (and the adjusting system administrator is notified accordingly). This example illustrates the important principal enforced in NQS that the set of limits under which a batch request is to run, are determined *and frozen* at the time that the batch request is first queued in its destination batch queue.

4.1.1.2 Spawning a Batch Request

The actual execution of a batch request is a somewhat complicated affair. First, a batch request may require that the output files of *stderr* and *stdout* be spooled, to a possibly remote machine destination. In order to do this safely, a temporary version of the output files is created in a protected location known to NQS.

Second, any additional environment variables optionally exported with the request from the originating (and possibly remote) host, are placed in the environment set for the shell that is about to be *exec*ed.

Third, based on any request shell specifications and the shell strategy policy at the local host, the proper shell (e.g. /bin/csh, /bin/ksh, /bin/sh, etc.) is chosen (see the *Batch Request Shell Strategies* section below). The chosen shell will be spawned as a *login* shell, virtually indistinguishable from the shell that the request owner would have gotten had they logged directly into the execution machine.

Fourth, all of the resource limits as supported by the underlying UNIX operating system implementation are applied to the new shell process, as determined for the request at the time it was first queued in the batch queue.

After the resource limits have been applied, the proper shell is *exec*ed, and the shell script that defines the batch request is actually executed. Upon completion, the spooled output files of *stderr* and *stdout* are returned to their possibly remote machine destinations.

4.1.1.3 Batch Queue Run Limits

To prevent the local host from being swamped with running batch requests, some mechanism must exist to prevent too many batch requests from running at any single given time. Currently, this mechanism is quite simple, and is implemented by the presence of two batch request *run*

limits.

The first batch request run limit is global in nature, and places a ceiling on the maximum number of batch requests allowed to execute simultaneously on the local host.

The second batch request run limit is applied at the queue level, and places a ceiling on the maximum number of batch requests allowed to execute simultaneously *in the containing batch queue*.

When a batch request completes execution, the entire set of batch queues is traversed in order of decreasing batch queue priority. For each batch queue in the order traversed, any eligible batch requests are spawned until either the queue run limit is reached, or the global batch request run limit is reached. If upon discovering that no more requests can be spawned for the batch queue under scrutiny, and the total number of running batch requests is still less than the global batch request run limit, then the next lower priority batch queue is examined applying the same algorithm, until all of the batch queues have been examined.

So far, this simple run limit scheme has sufficed as the only tool to control the running batch request execution load. Since batch requests can vary widely in their consumption of resources, additional more sophisticated control mechanisms limiting the number of simultaneously executing batch requests may be required in the future.

4.1.2 Device Queues

Device queues represent the second queue type implemented in NQS. Unlike their sibling *batch queues*, device queues do not have a set of associated resource quota limits. Device queues do however have a set of associated *devices*, which batch queues do not have.

4.1.2.1 Devices

For each *device queue*, there exists a set of one or more devices to which requests entering the device queue can be sent for execution. Each such device in turn has an associated server, which constitutes the program that is always spawned to handle a request that is given to the device for execution.

Any imaginable *queue-to-device mapping* can be configured. In general, N device queues can be configured to "feed" M devices. The only restriction placed on the value of N and M, is the obvious one that their respective values be greater than or equal to zero (note that it is possible for a device queue to exist without *any* devices in its device set, though such a queue is useless). It is even possible to have multiple device queues feeding the same device.

4.1.2.2 Spawning a Device Request

When an NQS device completes the task of handling a device request or is found to be idle after a device request has been recently queued, all of the device queues that "feed" the device are scanned to determine if they have a queued request that can be handled by the device. Like MDQS, an NQS device request can specify that a particular device *forms* type be used to execute the request. For a queued device request to be deemed eligible for execution by a particular device, any forms specified by the request must match the forms defined for the device. If the request does not specify a forms type, then it is assumed that the request can be satisfied by any device in the mapping set of the queue containing the request.

If two or more queues are found to contain a request that can be executed by the newly idled device, then the first available request from the device queue with the numerically higher *queue priority* is chosen. If two or more such queues have the same *queue priority*, then the queues are serviced in the classic "round-robin" fashion.

4.1.2.3 Device Queue Run Limits

Like a batch queue, some mechanism must exist to keep the number of simultaneously running device requests from swamping the local host. Unlike a batch queue however, device queues do

not have an associated run limit. Device queues are instead throttled by their associated devices, which can be disabled as necessary by a system administrator.

4.1.3 Pipe Queues

Pipe queues represent the third queue type implemented in NQS, and are responsible for routing and delivering requests to other (possibly remote) queue destinations. Pipe queues derive their name from their conceptual similarity to a *pipeline*, transporting requests to other queue destinations.

4.1.3.1 Pipe Queues and Request Transport

Differing from both batch and device queues, pipe queues do not have any associated quota limits or devices. Pipe queues do however have a set of associated *destinations* to which they route and deliver requests. Pipe queues also differ from their sibling batch and device queues, in that they can accept *both* batch and device requests.

With each pipe queue, there is an associated server that is spawned to handle each request released from the queue for routing and delivery. Ironically, the spawned instance of a pipe queue server is called a *pipe client*, due to the use of the word *server* in the context of a *client/server* network connection.

Thus, when a pipe queue request requires routing and delivery to some destination of the pipe queue, the associated pipe queue server is spawned as a *pipe client*, which must then route and deliver the request to a destination. For each attempted remote destination, this requires the creation of a *network server* process on the remote host acting as an agent on behalf of the pipe queue request. The choice of the term *pipe client* allows us to use the standard *client/server* vocabulary when discussing the queueing and delivery of a pipe queue request to a remote host.

4.1.3.2 Spawning a Pipe Request

When a *pipe client* is spawned to route and deliver a request, it is given complete freedom to choose any destinations from the destination set configured for the pipe queue, as possible destinations for the request. If a selected destination does not accept the request, then the pipe client is free to try another destination for the request.

It is quite possible for a request to be rejected by all but one of the possible destinations defined for a pipe queue. It is not necessary to find many destinations willing to accept the request. Only one accepting destination need exist for the pipe queue request to be handled successfully.

It is also possible for every single destination of a pipe queue to reject the request for reasons which are deemed permanent in nature (e.g. all of the destination queues reside on remote machines where the request owner does not have access to an account). In such situations, the request is deleted, and mail is sent to the request owner informing him or her of the demise of their request.

Requests can be rejected by a destination for a plethora of reasons, including remote host failures, queue type disagreements with the request type, lack of request owner account authorization at the remote queue destination, insufficient queue space, or any one of a hundred other reasons including the simple problem of the destination queue being disabled (unable to accept any new requests).

Some of the reasons for a destination rejection denote retriable events (the effort to queue the request at the destination may succeed if tried later). Examples of this kind of failure include the destination queue being disabled (the system administrators at the destination may enable it some time), and machine failures (the destination machine is crashed, but might be rebooted in the future).

Other destination rejection reasons are more permanent such as the lack of proper account authorization at the remote destination, or request and destination type disagreement (the request is a device request, and the destination is a batch queue for instance).

Due to the tremendous number of ways in which a request can be rejected by a queue destination, there is an equally tremendous amount of logic incorporated into NQS that attempts to deal with the situation. Some failures require that queue destinations be disabled for some finite amount of time after which the destination is considered retriable. All failures of the retriable variety require that the pipe queue request be requeued and delayed for some amount of time, after which an attempt is made to reroute the request.

Even the successful case of a request being tentatively accepted by a queue destination is fraught with complexity, since one or both machines involved in the transaction may crash at any time.

In summary, pipe queues are both powerful and complex. Since the pipe client configured with each pipe queue is allowed to choose which destinations to try from the destination set, it is possible to implement a crude but effective request class mechanism. The pipe client can examine the request, and then choose an appropriate destination queue that is more appropriate for the request. Thus, "large" batch requests queued in a pipe queue can be delivered to batch queues which may run only at night, while "small" batch requests can be delivered to fast batch queues, which run with a UNIX *nice* execution value that gives high compute priority, while keeping a small upper limit on CPU time and maximum file size for the request.

When a pipe queue is used as request class mechanism, it is wise to define the target destination queues with the attribute of *pipeonly*, which prevents any requests from being queued in such queues unless the requests are queued *from another pipe queue*. In this way, the request class policies implemented by the pipe queue and associated server (pipe client) can be strictly enforced.

Pipe queues also help to ameliorate the unreliability of the surrounding network and machines. Even if the proper destination machine is down or unreachable, the pipe queue mechanism can requeue the request and deliver it later, when the destination machine and connecting network are restored to operation.

4.1.3.3 Pipe Queue Run Limits

To prevent pipe queues from flooding the host system with an overly large number of simultaneously running pipe client processes, a mechanism identical to that implemented for batch queues is employed.

4.1.4 Request States

In the previous sections, we have described the general request and queue type concepts implemented in NQS. This section descends the staircase of detail, focusing on the different states that a request can go through all the way from its initial creation, to its ultimate execution.

A request residing within an NQS queue can be in one of several states. First of all, the request may actually be *running*. This request state exists for requests residing in batch and device queues, and implies that the request is presently being executed. The analogous request state for requests residing within a pipe queue is termed *routing*, since the request is not actually running, but is rather being routed and delivered to another queue destination.

The second (and most common) request state, is what is termed the *queued* state. A request in the queued state is completely ready to enter the *running* or *routing* states.

The third request state describes the condition of where a request is waiting for some finite time interval to pass, after which it will enter one of the states of queued, running, or routing. This request state is known as the *waiting* state.

The fourth request state is known as the *arriving* state. All requests in the arriving state are in the process of being queued from another (possibly remote) pipe queue. When completely received they will enter one of the other states of *waiting*, *queued*, *running*, or *routing*.

There are also three additional request states that are not implemented in the current version of NQS. The first such state is known as the *holding* state, and describes the condition of where an operator, user, or both have applied a *hold* to the given request. Such a request is frozen, and

cannot exit the hold state unless all holds applied by an operator or user have been released.

The second and third unimplemented request states concern the batch request states of *staging-in*, and *staging-out*. These states will not be implemented, unless the demand for the facility of file staging increases, since it is already possible to use the remote file copy commands in the shell script that constitutes a batch request, to copy the requisite files to and from the execution machine for the request. The advantage of implementing file staging is that NQS can use a transaction mechanism to prevent the execution of a batch request, until all of the input files have been staged-in to the local host. In this way, crashes of remote machines cannot cause a batch request to fail. Output files could be similarly staged.

4.2 More Landscaping

The previous major section described the queue and request model implemented in NQS. This section of the paper describes the implementation of queue access controls, batch request quota limits, batch request shell strategies, request transaction states, the networking implementation, account mapping across machine boundaries, NQS configuration control, status operations, and the possible future implementation of file staging.

4.2.1 Queue Access Controls

In any reasonable queueing system, it is necessary to provide for the configuration of queue access restrictions. Without such restrictions, there would be no way to prevent every user of the machine from submitting their requests to the fastest queue with the highest priority and resource limits on the machine. Thus, NQS supports queue access restrictions.

For each queue, access may be either *unrestricted* or *restricted*. If access is *unrestricted*, any request may enter the queue. If access is *restricted*, then a request can only enter the queue if the requester's login *user-id* or login *group-id* is defined in the access set for the target queue.

All such access permissions are always defined relative to user and group definitions present on the local host. The restriction that all user and group references be relative to the local host is not a problem, since request ownership mapping is performed whenever a request is transported across a machine boundary (see the *Account Mapping* section below).

Lastly, an additional queue access parameter known as *pipeonly* can be defined for any queue. The presence of this queue access attribute prevents requests from being directly placed within the queue by one of the user commands used to submit an NQS request. Queues with the *pipeonly* attribute can only accept requests queued via another pipe queue. As outlined in the summary of the *Spawning a Pipe Request* section, this attribute makes is possible to implement a simple request execution class facility.

4.2.2 Batch Request Quota Limits

As mentioned previously, NQS supports an extensive set of batch request resource quota limits. However, NQS cannot enforce a batch request resource quota limit unless the underlying UNIX implementation also supports the enforcement of the same limit. Thus, the resource limit enforcement functions of NQS have been implemented using an appropriate set of *#ifdefs*, allowing the system maintainers to configure the resource limit functions as appropriate.

It must be understood that NQS does not define the interface through which errant batch requests will be informed of their attempts to consume more of a given resource than is allocated to them. Upon exceeding some limit types, some UNIX implementations send a signal to the offending process. Other implementations may simply cause the errant system call to fail, with *errno* being set as appropriate.

If a batch request specifies the enforcement of a quota limit that is not enforceable at the execution machine, then the limit is simply ignored, and the request is run anyway. It is also possible to specify that no limit be given to the usage of a particular resource for both a batch request and batch queue.

Lastly, the NQS implementation of batch request resource limits allows each batch request to specify a *warning limit* value for UNIX kernels that allow processes to be warned when they are getting close to exceeding some hard quota limit. Once again as for hard quota limits, the actual enforcement mechanism of warning limits is up to the supporting UNIX kernel.

The full set of batch request resource quota limits recognized by NQS falls into two principal categories. The first category concerns only those limits applicable to *each* process of the process family comprising the running request. This category of limits is known collectively as the *per-process* limit set.

The second category concerns only those limits applicable to the entire request. That is, the consumption of the limited resource as consumed by *all* processes comprising the running batch request must never exceed the given *per-request* limit.

The complete set of batch request quota limits supported by NQS is listed below. Each limit is shown with its corresponding *Qsub*(1) command syntax (*Qsub*(1) is the command used to submit an NQS batch request). The use of the "(P)" and "(R)" description in the limit definition indicates the *per-process* or *per-request* nature of the limit:

-lc *limit*	- (P) corefile size limit.
-ld *limit* [, *warn*]	- (P) data segment size limit.
-lf *limit* [, *warn*]	- (P) file size limit.
-lF *limit* [, *warn*]	- (R) file space limit.
-lm *limit* [, *warn*]	- (P) memory size limit.
-lM *limit* [, *warn*]	- (R) memory space limit.
-ln *limit*	- (P) nice execution priority limit.
-ls *limit*	- (P) stack segment size limit.
-lt *limit* [, *warn*]	- (P) CPU time limit.
-lT *limit* [, *warn*]	- (R) CPU time limit.
-lv *limit* [, *warn*]	- (P) temporary file size limit.
-lV *limit* [, *warn*]	- (R) temporary file space limit.
-lw *limit*	- (P) working set limit.

The present implementation also includes provisions for the additional limits of:

-l6 *limit*	- (R) tape drive device limit.
-lP *limit*	- (R) number of processors limit.
-lq *limit* [, *warn*]	- (P) Quick device file size limit.
-lQ *limit* [, *warn*]	- (R) Quick device file space limit.

These last limits are not presently supported, but are instead reserved for future use. The last two future limits of *-lq*, and *-lQ* are reserved for defining limits on the amount of fast (quick) file storage to be allocated to a process of the running request, and to the entire running request. An example of a fast file storage resource can be found in the solid state disk (SSD) product that Cray Research Incorporated supports with their CRAY-XMP series of computers.

4.2.3 Batch Request Shell Strategy

The execution of a batch request requires the creation of a shell process to interpret the shell script which defines the batch request. On many UNIX systems, there is more than one shell available (e.g. /bin/csh, /bin/ksh, /bin/sh). To deal with this problem, NQS allows a shell pathname to be specified when a batch request is first submitted.

If no particular shell is specified for the execution of the request, then NQS must have some other means of deciding which shell to use when spawning the request. The solution to this dilemma has been to equip NQS with a *batch request shell strategy* , which can be configured as necessary by the local system administrators.

The batch request shell strategy as configured on a particular system, determines the shell to be used when executing a batch request on the local host that fails to identify any specific shell for

its execution. Three such shell strategies can be configured for NQS, and they are known by the names of

fixed,
free, and
login.

A shell strategy of *fixed* causes the request to be run by the *fixed shell*, the pathname of which is configured by the system administrator. Thus, a particular NQS installation may be configured with a *fixed* shell strategy where the default shell used to execute all batch requests is defined as the Bourne shell.

A shell strategy of *free* simply causes the user's login shell (as defined in the password file), to be *exec*ed). This shell is in turn given a pathname to the batch request shell script, and it is the user's login shell that actually decides which shell should be used to interpret the script. The *free* shell strategy therefore runs the batch request script *exactly* as would an interactive invocation of the script, and is the default NQS shell strategy.

The third shell strategy of *login* simply causes the user's login shell (as defined in the password file), to be the default shell used to interpret the batch request shell script.

The strategies of *fixed* and *login* exist for host systems that are short on available free processes. In these two strategies, a single shell is *exec*ed, and that same shell is the shell that executes all of the commands in the batch request script (barring shell *exec* operations in any user startup files: .profile, .login, .cshrc).

In every case however, the shell that is chosen to execute the batch request is always spawned as a login shell, with all of the environment variables and settings that the request owner would have gotten, had they logged directly into the machine.

The shell strategy as configured for any particular host, can always be determined by the NQS *qlimit* command.

4.2.4 Transactions

The accurate recording of request state information is a sometimes complicated affair within NQS. The need to support some reliable mechanism for the recording of request state is particularly critical when an NQS request is in the process of being routed and delivered to a remote queue destination. It is also necessary to support some reliable mechanism for detecting interrupted executions of batch and device requests upon system restart, so that they can be restarted or aborted depending upon the user's wishes.

To do this, NQS uses the UNIX file system to record request state information. On the surface, this use of the UNIX file system to store request state information seems trivial. It's not.

The UNIX file system buffer cache implementation of "lazy write I/O" makes the situation almost intolerable, since the update of request state information must occur *synchronously*, for many of the request state transitions. That is, there are several instances where the state of a particular request must be accurately recorded on the physical disk medium *prior* to continuing further with the transaction, otherwise reliable transaction recovery is impossible.

The need for synchronous state updates becomes absolutely critical when an NQS pipe client process is routing and delivering a request to a remote queue destination on another machine. The algorithm used to remotely queue an NQS request must allow for both machines involved in the transaction to crash, without leaving things in an unrecoverable state.

The algorithm to do this is implemented using a well known technique called the *two-phase commit protocol*. While the algorithm is quite interesting, space restrictions prohibit a full explanation of the technique here, and the reader is referred to the text: *Nested Transactions: An Approach to Reliable Distributed Computing* by Moss. [2]

What will be described here however, is the unusual mechanism implemented in the present version of NQS to get around the UNIX file system buffer cache.

While AT&T system V release 2 UNIX supposedly supported an undocumented flag in the *open*(2) system call forcing synchronous write operations for the opened file descriptor, not all UNIX implementations running on the various machines of the NPSN supported this feature. However, an examination of the UNIX source code as supplied by all of the different vendors showed that the *link*(2) system call was synchronous, to the extent that the target file inode had either been written to disk, or was scheduled to be written to disk upon return from the system call.

Therefore, since the amount of transaction state information for each request is quite small, NQS does something unbelievably strange. It uses the modification time field of protected and preallocated files to store transaction state information for each request.

The update of transaction state information in this manner is performed by setting the modification time of the appropriately preallocated file (never created or deleted once NQS is installed), making a link to the updated inode to force its writing to disk, followed by an unlink to remove the temporary link used to force the I/O operation. While the desired synchronous transaction state update is accomplished using a mechanism that is not very fast or efficient, it does have at least the virtue of being relatively portable.

All of the code involved in setting and reading transaction state for a request is isolated in a very small number of NQS source modules. When a synchronous I/O mechanism becomes supported as a general UNIX standard, then the implementation of NQS will be changed to take advantage of it, discarding the atavistic technique described here.

4.2.5 Networking Implementation

At present, all NQS network conversations are performed using the *Berkeley socket mechanism*, as ported into the respective vendor kernels or emulated by other means. The only connection type used by NQS is that of a *stream connection*, in which NQS assumes that the requisite bytes will be reliably transmitted to and from the server in the order in which they were written, by the underlying network software of the respective host systems. Any conversion to the use of the *streams* mechanism as developed by AT&T should be extremely straightforward.

In general, all NQS database information is always stored in the form most appropriate for the local host. If it becomes necessary to communicate information to another remote NQS host, then the information is converted into a network format understood by all NQS machines.

All network conversations performed by NQS are always done using the classic *client/server* model, in which a client process creates a connection to the remote machine where a server process is created to act on behalf of the client process.

When this initial connection is created, some introductory information is exchanged between the two processes. Regardless of the transaction to be conducted, the format of the introduction is always the same, in which certain key "personality" information is transmitted by the client process to the remote server. Included as part of this introductory dialogue, are the the client's identity in the form of its real user-id and corresponding user name at the client host, and the timezone in effect at the client's machine.

The parameters of real user-id and user name are both passed to the server process, so that the server can map the identity of the client to the appropriate account at the remote server machine. Although one of these two parameters is sufficient, both are passed so that the client mapping at the server machine can be performed by *either* user-id or user name, depending upon the implementation at the remote host.

The timezone for the client is also passed across so that future implementations of NQS when performing remote status operations, will properly display event times using the timezone of the client.

Lastly, the initial dialogue is the obvious place in which attempts can be made by malevolent users to try to gain unauthorized entry to a remote machine. At present, the only mechanism to prevent this, is the difficulty in faking the NQS protocols, and the requirement that all networking connections be made from privileged ports that can only be gotten by privileged root processes.

4.2.6 Account Mapping

When a network connection is established between an NQS client process and a remote NQS server process, an account mapping must be performed so that the network server at the remote machine can take on the proper identity attributes. This mapping is performed for all network conversations. In particular, the transport of a batch or device request requires that the ownership of the request be adjusted as appropriate, since the user-id of the request owner is not necessarily the same on all machines.

This mapping can be performed either by mapping the client's host and user-id, or client's host and user name to the proper account. In both cases though, the mapping must be done by the remote server machine if there is to be any semblance of security.

The choice of whether to map user-id or user name values was the subject of intense debate. In the beginning, the mapping was to have been made by mapping user-ids. Near the very end of the project, it was mandated that the mapping be performed by user name, and not user-id.

The present implementation of NQS has therefore adopted the defensive position that the server machine should make the decision as to which algorithm to use when performing an account mapping. Since both the user-id and user name of the client process are available to the server process (see the *Networking Implementation* section), the server can use either one when performing the account mapping.

Beyond the problem of user-id versus user name mapping, an additional problem is posed by the need to determine the identity of the client's host, irrespective of the network interface upon which a connection is made. In the environment of the NPSN, there are often at least two different principal paths by which a machine can be reached. The example paths typically include the interfaces of ethernet and hyperchannel, and lead to the existence of entries in the UNIX */etc/hosts* file where the names of *amelia-hy* and *amelia-ec* denote the two different paths of hyperchannel and ethernet to the same machine known locally as *amelia*.

NQS however requires that it be able to tell without ambiguity that connections coming from *amelia-hy* and *amelia-ec* denote connections coming from the *same machine*, even though the entries in the */etc/hosts* file are separate.

To do this, it was necessary to create the notion of a *machine-id*, a number that uniquely identifies a client machine, irrespective of the path used to conduct the network conversation. Thus, an additional mapping mechanism was created to map different client host addresses to a single unique machine-id.

Like the user-id versus user name mapping controversy, this decision was also caught in a maelstrom of controversy. When the dust finally settled, the *machine-id* concept was still present in the NQS implementation. Unfortunately, the storm of controversy swept away the tools which were going to be used to administer the machine-id mappings. Thus, the present implementation provides a rudimentary program called *nmapmgr* which can be used to painfully create the requisite machine-id mappings.

Someone receiving NQS source code for the first time would do well to either implement their own machine-id mapping mechanism, or polish the present mechanism.

4.2.7 Configuration Control

All of the setup and configuration of NQS is accomplished through the use of a single configuration program known as the *qmgr* utility. This program establishes a connection to the

local NQS daemon, and transmits message packets to perform the various configuration commands implemented in NQS. This program is quite user friendly, and provides an on-line help facility.

The use of an intelligent configuration program to setup and modify NQS on the local machine provides many benefits, one of which is the benefit of consistency. One cannot for example, add a queue-to-device mapping for a non-existent device or queue.

When given a particular command such as adding a device to the queue-to-device mapping set for some queue, the *qmgr* utility builds a message update packet which is then sent to the local NQS daemon for processing. The local NQS daemon then successfully performs the update or returns an error code, which the *qmgr* program diagnoses. In either case, the final outcome of the command is always displayed to the user system administrator.

4.2.8 Status Operations

All of the obvious status operations are supported by NQS, including device, request, queue, and limit queries. The latter status operation is used to determine the set of batch request resource limits supported by NQS on the local machine.

These status functions are supported by the respective NQS commands: *qdev*, *qstat*, and *qlimit*, with *qstat* providing information about previously queued requests and their containing queues.

Due to time constraints, the only status function which has been networked is the *qstat* command. As time becomes available, this situation will hopefully be corrected.

4.2.9 File Staging

Although file staging is not presently implemented by NQS, future versions of NQS may implement such a facility. A thorough examination of the NQS source code will reveal that provisions have been made for this eventuality in both the request transaction state mechanism, and the batch request data structures.

5. Conclusion

NQS is only another effort aimed at providing a more complete queueing system for a collection of UNIX machines operating in a networked environment.

As mentioned in the *Implementation Strategies* section, NQS was designed and written after a careful examination of a previous UNIX queueing system known as MDQS. It is hoped that others will now build on NQS, as NQS has been built from ideas in MDQS.

REFERENCES

1. Kingston, Douglas P. III, *A Tour Through the Multi-Device Queueing System*, revised for MDQS 2.0, Ballistic Research Laboratory, Army Armament Research And Development Command (AARADCOM). September 12, 1983.

2. Moss, J. Elliot B., *Nested Transactions: An Approach to Reliable Distributed Computing*, Cambridge, Massachusetts: The MIT Press, 1985.

Appendix:

NQS Manual Pages

NAME

qdel – delete or signal NQS request(s).

SYNOPSIS

qdel [–k] [–signo] [–u username] request-id ...

DESCRIPTION

Qdel deletes all queued NQS requests whose respective *request-id* is listed on the command line. Additionally, if the flag -k is specified, then the default signal of **SIGKILL** (-9) is sent to any running request whose *request-id* is listed on the command line. This will cause the receiving request to exit and be deleted. If the flag –signo is present, then the specified signal is sent instead of the **SIGKILL** signal to any running request whose *request-id* is listed on the command line. In the absence of the -k and –signo flags, *qdel* will **not** delete a *running* NQS request.

To delete or signal an NQS request, the invoking user *must* be the owner; namely the submitter of the request. The only exception to this rule occurs when the invoking user is the *superuser*, or has NQS operator privileges as defined in the NQS manager database. Under these conditions, the invoker may specify the –u username flag which allows the invoker to delete or signal requests owned by the user whose account name is *username*. When this form of the command is used, **all** *request-ids* listed on the command line are presumed to refer to requests owned by the specified user.

An NQS request is always uniquely identified by its *request-id*, no matter where it is in the network of the machines comprising the NPSN. A *request-id* is always of the form: *seqno* or *seqno.hostname* where *hostname* identifies the machine from whence the request was originally submitted, and *seqno* identifies the sequence number assigned to the request on the originating host. If the *hostname* portion of a *request-id* is omitted, then the local host is always assumed.

The *request-id* of any NQS request is displayed when the request is first submitted (unless the *silent* mode of operation for the given NQS command was specified). The user can also obtain the *request-id* of any request through the use of the *qstat*(1) command.

CAVEATS

When an NQS request is signalled by the methods discussed above, the proper signal is sent to *all* processes comprising the NQS *request* that are in the same *process group*. Whenever an NQS request is spawned, a new *process group* is established for all processes in the request. However, should one or more processes of the request successfully execute a *setpgrp*() system call, then such processes will **not** receive any signals sent by the *qdel*(1) command. This can lead to "rogue" request processes which must be killed by other means such as the *kill*(1) command. For the UNIX implementations that support the ability to "lock" a process, and all of its progeny into a *process-group*, NQS will exploit this capability to prevent processes from "escaping" in this manner.

SEE ALSO

qdev(1), qlimit(1), qpr(1), qstat(1), qsub(1),
kill(2), setpgrp(2), signal(2) in the *NPSN UNIX System Programmer Reference Manual.*
qmgr(1M) in the *NPSN UNIX System Administrator Reference Manual.*

NPSN HISTORY

Origin: Sterling Software Incorporated

August 1985 – Brent Kingsbury, Sterling Software
Original release.

May 1986
Second release.

NAME

qdev – display status of NQS devices

SYNOPSIS

qdev [device-name] [device-name@host-name ...]

DESCRIPTION

Qdev displays the status of devices known to the Network Queueing System (NQS).

If no devices are specified, then the current state of each NQS device on the local host is displayed. Otherwise, the response is limited to the devices specified. Devices may be specified either as *device-name* or *device-name@host-name*. In the absence of a *host-name* specifier, the local host is assumed.

A *device header* with several headings is displayed for each of the selected devices. The first heading in a device header appears as **Device:**, and is followed by the name of the device formatted as *device-name@host-name*. The second heading of **Fullname:** is followed by the full path name of the special file associated with the device. The third heading of **Server:** is followed by the command line which will be used to *execve(2)* the device server. The fourth heading of **Forms:** is followed by the forms configured for the device.

The final heading of **Status:** prefaces a display of the general device state. The general state of a device is defined by two principal properties of the device.

The first property concerns whether or not the device is willing to continue accepting queued requests. If it is, the device is said to be **ENABLED**. If the device is unwilling to continue accepting queued requests, and is idle, its state is **DISABLED**. A third state of **ENABLED/CLOSED** is used to describe a device that is unwilling to continue accepting queued requests, but is not yet idle.

The second principal property of a device concerns whether or not the device is busy. There are three cases. If the device is busy, it is said to be **ACTIVE**. If the device is idle and not known to be out of service, it is said to be **INACTIVE**. Finally, if the device is idle and known to be out of service, it is said to be **FAILED**. **FAILED** covers both hardware and software failures.

If a device is busy, information about the active request follows the device header. The *request-name*, *request-id*, and the name of the user who submitted the request are all displayed.

SEE ALSO

qdel(1), qlimit(1), qpr(1), qstat(1), and qsub(1)
in the *NPSN UNIX System Programmer Reference Manual.*
qmgr(1M) in the *NPSN UNIX System Administrator Reference Manual.*

NPSN HISTORY

Origin: Sterling Software Incorporated

May 1986 – Robert Sandstrom, Sterling Software
Original release.

NAME

qlimit – show supported batch limits, and shell strategy for the named host(s).

SYNOPSIS

qlimit [host-name ...]

DESCRIPTION

Qlimit displays the set of batch request resource limit types that *can* be directly enforced on the implied local host or named hosts, and also the *batch request shell strategy* defined for the implied local host or named hosts.

If no *host-names* are given, then the information displayed is only relevant to the local host. Otherwise, the supported batch request limits, and *batch request shell strategy* for each of the named hosts is displayed.

NQS supports many batch request resource limit types that can be applied to an NQS batch request. However, not all UNIX implementations are capable of supporting the rather extensive set of limit types that NQS provides.

The set of limits applied to a batch request, is always restricted to the set of limits that can be directly supported by the underlying UNIX implementation. If a batch request specifies a limit that cannot be enforced by the underlying UNIX implementation, then the limit is simply ignored, and the batch request will operate as though there were no limit (other than the obvious physical maximums), placed upon that resource type.

When an attempt is made to queue a batch request, each *limit-value* specified by the request (that can also be supported by the local UNIX implementation), is compared against the corresponding *limit-value* as configured for the destination batch queue. If the corresponding batch queue *limit-value* for all batch request *limit-values is defined as unlimited*, or is *greater than* or *equal to* the corresponding batch request *limit-value*, then the request can be successfully queued, provided that no other anomalous conditions occur. For request *infinity limit-values*, the corresponding queue *limit-value* must also be configured as infinity.

These resource limit checks are performed irrespective of the batch request arrival mechanism, either by a direct use of the *qsub*(1) command, or by the indirect placement of a batch request into a batch queue via a *pipe* queue. It is impossible for a batch request to be queued in an NQS batch queue if *any* of these resource limit checks fail.

Finally, if a request fails to specify a *limit-value* for a resource limit type that is supported on the execution machine, then the corresponding *limit-value* as configured for the destination queue, becomes the *limit-value* for the unspecified request limit.

Upon the successful queueing of a request in a batch queue, the set of limits under which the request will execute is frozen, and will not be modified by subsequent *qmgr*(1M) commands that alter the limits of the containing batch queue.

As mentioned above, this command also displays the *shell strategy* as configured for the implied local host, or named hosts. In the absence of a *shell specification* for a batch request, NQS must choose which shell should be used to execute that batch request. NQS supports three different algorithms, or *strategies* to solve this problem that can be configured for each system by a system administrator, depending on the needs of the user's involved, and upon system performance criterion.

The three possible shell strategies are called:

fixed,
free, and
login.

These shell strategies respectively cause the configured *fixed* shell to be exec'd to interpret all batch requests, cause the user's login shell as defined in the password file to be exec'd which in turn chooses and spawns the appropriate shell for running the batch shell script, or cause only the user's login shell to be exec'd to interpret the script.

A shell strategy of *fixed* means that the same shell as chosen by the system administrator, will be used to execute all batch requests.

A shell strategy of *free* will run the batch request script *exactly* as would an interactive invocation of the script, and is the default NQS shell strategy.

The strategies of *fixed*, and *login* exist for host systems that are short on available free processes. In these two strategies, a single shell is exec'd, and that same shell is the shell that executes all of the commands in the batch request shell script.

When a shell strategy of *fixed* has been configured for a particular NQS system, then the "fixed" shell that will be used to run all batch requests at that host is displayed.

SEE ALSO

qdel(1), qdev(1), qpr(1), qstat(1), and qsub(1) in the *NPSN UNIX System Programmer Reference Manual.*
qmgr(1M) in the *NPSN UNIX System Administrator Reference Manual.*

NPSN HISTORY

Origin: Sterling Software Incorporated

May 1986 – Brent Kingsbury, Sterling Software
Original release.

NAME

qmgr – NQS queue manager program

SYNOPSIS

qmgr

DESCRIPTION

Qmgr is a program used by the System Administrator or System Operator to control NQS requests, queues, devices, and the general NQS configuration at the local machine.

Definitions

An NQS *request* is a request by a user or user program to perform a function that requires a delay in servicing (e.g., after a certain time). Examples of such functions are the the scheduling of a shared serial-access resource (e.g., a printer), and the scheduling of batch job requests. A *device queue* holds requests for resources such as printers and Computer Output Microfilm (COM) units. A *batch queue* holds requests for scheduled, perhaps delayed, processing by various subsystems in the NPSN. A *pipe queue* is a queue which can pass queued requests on to other *pipe queues*, *batch queues*, or *device queues*. An NQS *device* is a site at which a shared serial-access resource such as a printer is offered. A *daemon* is a process which is designed to run continuously, providing some service when needed. (See the **QUEUE TYPES** section below for more information concerning queues.) Lastly, an NQS *manager* identifies a person who is capable of changing any NQS characteristic on the local machine. An NQS *operator* identifies a person who can execute only the *operator commands* as a proper subset of all the commands provided by the *qmgr*(1m) utility.

Commands

The following paragraphs describe the syntax of each *Qmgr*(1m) command. All command keywords are recognized regardless of upper or lower case usage. Keyword characters shown in uppercase indicate the *smallest possible abbreviation of the keyword* for the particular command being described.

ABort Queue *queue* [*seconds*]

All requests in the named *queue* that are currently running are aborted as follows. A SIGTERM signal is sent to each process of each request presently running in the named *queue*. After the specified number of *seconds* of real time have elapsed, a SIGKILL signal is sent to all remaining processes for each request running in the named *queue*. If a *seconds* value is not specified, then the delay is sixty seconds. All requests aborted by this command are deleted, and all output files associated with the requests are returned to the appropriate destination.

NQS *operator* privileges are required to use this command.

ADd Queues = (*queue* [, *queue* ...]) *complex*

Add the specified *queue(s)* to the batch queue complex named *complex*.

Full NQS *manager* privileges are required to use this command.

ADd DEStination = *destination queue*
ADd DEStination = (*destination* [, *destination* ...]) *queue*

The specified *destination(s)* are added as valid destinations for a pipe queue named *queue*.

Full NQS *manager* privileges are required to use this command.

ADd DEVice = *device queue*
ADd DEVice = (*device* [, *device* ...]) *queue*
The specified *device(s)* are added as resources to service requests from *queue*. The *device(s)* must exist (see **Create DEVICE** below).

Full NQS *manager* privileges are required to use this command.

ADd Forms *form-name* ...
The specified *form-name(s)* are added to the list of valid forms.

Full NQS *manager* privileges are required to use this command.

ADd Groups = *group queue*
ADd Groups = (*group* [, *group* ...]) *queue*
The specified *group(s)* are added to the access list for *queue*. There are two ways to specify a group:

group name
[*group id*]

Full NQS *manager* privileges are required to use this command.

ADd Managers *manager* ...
The specified *manager(s)* are added to the list of authorized NQS managers with privileges as specified. A *manager* specification consists of an account name specification, followed by a colon, followed by either the letter *m* or the letter *o*. There are four ways to specify an account name:

local_account_name
[*local_user_id*]
[*remote_user_id*]@*remote_machine_name*
[*remote_user_id*]@[*remote_machine_mid*]

If the account name specification is followed by *:m*, then the account is designated as an NQS *manager* account, capable of using all **qmgr** commands. If the account name specification is followed by *:o*, then the account is designated as an NQS *operator* account, capable of only using those commands appropriate for an NQS *operator*.

Full NQS *manager* privileges are required to use this command.

ADd Users = *user queue*
ADd Users = (*user* [, *user* ...]) *queue*
The specified *user(s)* are added to the access list for *queue*. There are two ways to specify a user:

user name
[*user id*]

Full NQS *manager* privileges are required to use this command.

Create Batch_queue *queue* **PRiority** = *n* [**PIpeonly**]
[**Run_limit** = *n*]

Define a batch queue named *queue* with inter-queue priority *n* (0..63). If **PIpeonly** is specified, then requests may enter this *queue* only if their source is a pipe queue. The specification of a **Run_limit** sets a ceiling on the maximum number of requests allowed to run in the batch queue at any given time. The default run-limit is one. (See the **QUEUE TYPES** section below for more information.)

Full NQS *manager* privileges are required to use this command.

Create Complex = (*queue* [, *queue* ...]) *complex*
Create a queue *complex* consisting of the specified set of batch *queues*. NQS provides for the grouping of a set of batch queues into a queue complex which can have an associated **Run_limit**.

Full NQS *manager* privileges are required to use this command.

Create DEVICE *device* **FOrms** = *forms* **FUllname** = *filename*
Server = (*server*)
Define a *device* with the specified *forms* and associate it with a *server*. This is done by specifying an absolute path name to the program binary (*server*) and any arguments required by the program. *Filename* is the absolute path name of the device (special file) and is typically /**dev**/*device*.

Full NQS *manager* privileges are required to use this command.

Create DEVICE_queue *queue* **PRiority** = *n* [**Device** = *device*]
[**Device** = (*device* [, *device* ...])]
[**PIpeonly**]
Define a device queue named *queue* with inter-queue priority *n* (0..63). If **PIpeonly** is specified, then requests may enter this *queue* only if their source is a pipe queue. After **Device** appears a list of one or more *devices* that may service this *queue*. (See the **QUEUE TYPES** section below for more information.)

Full NQS *manager* privileges are required to use this command.

Create Pipe_queue *queue* **PRiority** = *n* **Server** = (*server*)
[**Destination** = *destination*]
[**Destination** = (*destination* [, *destination* ...])]
[**PIpeonly**] [**Run_limit** = *n*]
Define a pipe queue named *queue* with inter-queue priority *n* (0..63) and associate it with a *server*. This is done by specifying an absolute path name to the program binary (*server*) and any arguments required by the program. After **Destination** appears a list of one or more *destination* queues that requests from this pipe *queue* may be sent to. If **PIpeonly** is specified, then requests may enter this *queue* only if their source is a pipe queue. **Run_limit** sets a ceiling on the maximum number of requests allowed to run in the pipe queue at any given time. The default run-limit is one. (See the **QUEUE TYPES** section below for more information.)

Full NQS *manager* privileges are required to use this command.

DElete Complex *complex*
Delete a queue *complex*.

Full NQS *manager* privileges are required to use this command.

DElete DEStination = *destination queue*
DElete DEStination = (*destination* [, *destination* ...]) *queue*
Delete the mappings from the pipe queue *queue* to the *destination* queues. All requests from the named *queue* being transferred to a deleted *destination* complete normally. If all *destinations* for a pipe *queue* are deleted in this manner, then the pipe *queue* is effectively stopped.

Full NQS *manager* privileges are required to use this command.

DElete DEVice *device*
Delete the specified *device*. A device must be disabled to delete it from the device set (see **DIsable Device** below).

Full NQS *manager* privileges are required to use this command.

DElete DEVice = *device queue*
DElete DEVice = (*device* [, *device* ...]) *queue*
Delete the mappings from the device queue *queue* to the *device(s)*. All requests from the named device *queue* running on any of the named *devices* are allowed to complete normally. If ALL queue-to-device mappings for the named device *queue* are removed by this command, then the *queue* is effectively stopped.

Full NQS *manager* privileges are required to use this command.

DElete Forms *form-name ...*
The specified *form-name(s)* are deleted from the list of valid forms.

Full NQS *manager* privileges are required to use this command.

DElete Groups = *group queue*
DElete Groups = (*group* [, *group* ...]) *queue*
The specified *group(s)* are deleted from the access list for *queue*. There are two ways to specify a group:

group name
[*group id*]

Full NQS *manager* privileges are required to use this command.

DElete Managers *manager ...*
The specified *manager(s)* are deleted from the list of authorized NQS managers. A *manager* specification consists of an account name specification, followed by a colon, followed by either the letter *m* or the letter *o*. There are four ways to specify an account name:

local_account_name
[*local_user_id*]
[*remote_user_id*]@*remote_machine_name*
[*remote_user_id*]@[*remote_machine_mid*]

If the account name specification is followed by *:m*, it is understood that the account is currently permitted to use all **qmgr** commands. If the account name specification is followed by *:o*, it is understood that the account is currently permitted to use only those commands appropriate for an operator to use. The *root* account always has full privileges.

Full NQS *manager* privileges are required to use this command.

DElete Queue *queue*
The *queue* is deleted. To delete a *queue*, no requests may be present in the *queue* and the *queue* MUST be disabled (see **DIsable Queue** below). Any queue-to-device mappings are updated accordingly.

Full NQS *manager* privileges are required to use this command.

DElete Request *requestid ...*
Delete the request(s) named by the *requestid(s)*. This command can delete both running and non-running requests. If a request is running, then all processes of the request are sent a SIGKILL signal.

NQS *operator* privileges are required to use this command.

DElete Users = *user queue*
DElete Users = (*user* [, *user ...*]) *queue*
The specified *user(s)* are deleted from the access list for *queue*. There are two ways to specify a user:

user name
[*user id*]

Full NQS *manager* privileges are required to use this command.

DIsable Device *device*
The current request will complete. After that, the *device* is prevented from handling any more requests until it is enabled (see **ENable Device** below). If the disabled *device* was the last enabled device in a queue-to-device mapping, then the device queue is effectively stopped.

NQS *operator* privileges are required to use this command.

DIsable Queue *queue*
Prevent any more requests from being placed in this *queue*.

NQS *operator* privileges are required to use this command.

ENable Device *device*
If the *device* is already enabled, then this is a no-op. Otherwise, the *device* becomes available to handle requests.

NQS *operator* privileges are required to use this command.

ENable Queue *queue*

If the *queue* is already enabled, then this is a no-op. Otherwise, the *queue* is enabled to accept new requests.

NQS *operator* privileges are required to use this command.

EXit

Exit from the NQS manager subsystem.

Help [*command*]

Get help information. **Help** without an argument displays information about what commands are available. **Help** with an argument displays information about that command. The command may be partially specified as long as it is unique. A more complete help request yields more detailed information.

The **Help** *command* provides information that is often more extensive than the command descriptions in this manual page! Use it.

Lock Local_daemon

Lock the NQS local daemon into memory. See **plock(2)**.

NQS *operator* privileges are required to use this command.

MODify Request [**Nice_limit** = *nice*] [**RTime_limit** = *Tlimit*] [**RMemory_limit** = *Mlimit*] *requestid*

Modify parameters for the request specified by *requestid*. *Nice* is the initial nice value for the request. *Tlimit* is a per request CPU time limit. *Mlimit* is a per request memory limit. For the syntax of these limits, see the **LIMITS** section below.

NQS *operator* privileges are required to use this command.

MOVe Queue *queue1 queue2*

Move all requests currently in *queue1* to *queue2*.

NQS *operator* privileges are required to use this command.

MOVe Request *requestid ... queue*

Move the request(s) named by the *requestid(s)* to the named *queue*.

NQS *operator* privileges are required to use this command.

Purge Queue *queue*

All queued requests are purged (dropped) from the *queue* and are irretrievably lost. Running requests in the *queue* are allowed to complete.

NQS *operator* privileges are required to use this command.

Remove Queue = (*queue* [, *queue* ...]) *complex*

Remove the specified *queue(s)* from the batch queue complex named *complex*.

Full NQS *manager* privileges are required to use this command.

SEt COMplex Run_limit = *run-limit complex*

Change the *run-limit* of an NQS queue *complex*. The *run-limit* determines the maximum number of requests that will be allowed to run in the queue *complex* at any given time.

NQS *operator* privileges are required to use this command.

SEt CORefile_limit = (*limit*) *queue*

Set a per-process maximum core file size *limit* for a batch *queue* against which the per-process maximum core file size limit for a request may be compared. If the local host does not support per-process core file size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum core file size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process core file size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the **LIMITS** section below.

Full NQS *manager* privileges are required to use this command.

SEt DAta_limit = (*limit*) *queue*

Set a per-process maximum data segment size *limit* for a batch *queue* against which the per-process maximum data segment size limit for a request may be compared. If the local host does not support per-process data segment size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum data segment size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process maximum data segment size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the **LIMITS** section below.

Full NQS *manager* privileges are required to use this command.

SEt DEBug *level*

Set the debug *level*. The following values are valid:

0	No debug
1	Minimum debug
2	Maximum debug

Full NQS *manager* privileges are required to use this command.

SEt DEFault Batch_request Priority *priority*

Set the default intra-batch-queue *priority*. This is NOT the UNIX execution time priority. This is the priority used if the user does not specify an intra-queue priority parameter on the **qsub(1)** command.

Full NQS *manager* privileges are required to use this command.

SEt DEFault Batch_request Queue *queue*

Set the default batch *queue*. This is the queue used if the user does not specify a queue parameter on the **qsub(1)** command.

Full NQS *manager* privileges are required to use this command.

SEt DEFault DEStination_retry Time *retry_time*
Set the default number of hours that can elapse during which time a pipe queue destination can be unreachable, before being marked as completely failed.

Full NQS *manager* privileges are required to use this command.

SEt DEFault DEStination_retry Wait *interval*
Set the default number of minutes to wait before retrying a pipe queue destination that was unreachable at the time of the last attempt.

Full NQS *manager* privileges are required to use this command.

SEt DEFault DEVice_request Priority *priority*
Set the default intra-device-queue *priority*. This is the priority used if the user does not specify an intra-queue priority parameter on the **qpr(1)** command.

Full NQS *manager* privileges are required to use this command.

SEt DEFault Print_request Forms *form-name*
Set the default print forms to *form-name*. This is the forms used if the user does not specify a forms parameter on the **qpr(1)** command.

Full NQS *manager* privileges are required to use this command.

SEt DEFault Print_request Queue *queue*
Set the default print *queue*. This is the queue used if the user does not specify a queue parameter on the **qpr(1)** command.

Full NQS *manager* privileges are required to use this command.

SEt DEStination = *destination queue*
SEt DEStination = (*destination* [, *destination* ...]) *queue*
Associate one or more *destination* queues with a particular pipe *queue*.

Full NQS *manager* privileges are required to use this command.

SEt DEVICE = *device queue*
SEt DEVICE = (*device* [, *device* ...]) *queue*
Associate one or more *devices* with a particular *queue*.

Full NQS *manager* privileges are required to use this command.

SEt DEVICE_server = (*server*) *device*
Associate a *server* with a *device*. *Server* should consist of the absolute path name to the program binary followed by any arguments required by the program.

Full NQS *manager* privileges are required to use this command.

SEt Forms *form-name ...*
Specify the valid *form-name(s)*. Other valid forms may be added to this list (see **ADd**

Forms above).

Full NQS *manager* privileges are required to use this command.

SEt Forms = *form-name device*
Set the *form-name* for a *device*.

Full NQS *manager* privileges are required to use this command.

SEt LIfetime *lifetime*
Set pipe-queue request *lifetime* in hours.

Full NQS *manager* privileges are required to use this command.

SEt LOg_file *filename*
Specify the name of the log file for NQS messages.

Full NQS *manager* privileges are required to use this command.

SEt MAIl *userid*
Specify the *userid* used to send NQS mail.

Full NQS *manager* privileges are required to use this command.

SEt MANagers *manager ...*
The list of authorized NQS managers is set to the specified *manager(s)*. A *manager* specification consists of an account name specification, followed by a colon, followed by either the letter *m* or the letter *o*. There are four ways to specify an account name:

local_account_name
[*local_user_id*]
[*remote_user_id*]@*remote_machine_name*
[*remote_user_id*]@[*remote_machine_mid*]

If the account name specification is followed by *:m*, then the account is designated as an NQS *manager* account, capable of using all **qmgr** commands. If the account name specification is followed by *:o*, then the account is designated as an NQS *operator* account, capable of only using those commands appropriate for an NQS *operator*. The *root* account always has full privileges. Also see **ADd Manager** above.

Full NQS *manager* privileges are required to use this command.

SEt MAXimum Copies *copies*
Set the maximum number of print *copies*.

Full NQS *manager* privileges are required to use this command.

SEt MAXimum Open_retries *retries*
Specify the maximum number of *retries* for a failed device open.

Full NQS *manager* privileges are required to use this command.

SEt MAXimum Print_size *size*
Specify the maximum *size* of an NQS print file in bytes.

Full NQS *manager* privileges are required to use this command.

SEt NEtwork Client = (*client* **)**
Specify the network client to be used. *Client* should consist of the absolute path name of the client followed by any arguments required by the client.

Full NQS *manager* privileges are required to use this command.

SEt NEtwork Daemon = (*daemon* **)**
Specify the network daemon to be used. *Daemon* should consist of the absolute path name of the daemon followed by any arguments required by the daemon.

Full NQS *manager* privileges are required to use this command.

SEt NEtwork Server = (*server* **)**
Specify the network server to be used. *Server* should consist of the absolute path name of the server followed by any arguments required by the server.

Full NQS *manager* privileges are required to use this command.

SEt NIce_value_limit = *nice-value queue*
Set the UNIX *nice-value* limit for a batch *queue*, against which the nice-value for a request may be compared. If a request already in the queue has asked for treatment more favorable than the new *nice-value*, then it will be given a grandfather clause. A request specifying a nice-value may only enter a batch queue if the queue's nice value is numerically less than (more willing to allow access to the CPU) or equal to the request's nice value. *Nice-value* is an integer preceded by an optional negative sign.

Full NQS *manager* privileges are required to use this command.

SEt NO_Access *queue*
Specify that no one will be allowed to place requests in *queue*. Root is an exception; requests submitted by root are always allowed into a queue, even if root is not explicitly given access.

Full NQS *manager* privileges are required to use this command.

SEt NO_Default Batch_request Queue
Indicate that there is to be no default batch request queue.

Full NQS *manager* privileges are required to use this command.

SEt NO_Default Print_request Forms
Indicate that there is to be no default print request forms.

Full NQS *manager* privileges are required to use this command.

SEt NO_Default Print_request Queue
Indicate that there is to be no default print request queue.

Full NQS *manager* privileges are required to use this command.

SEt NO_Network_daemon

Indicate that there is to be no network daemon.

Full NQS *manager* privileges are required to use this command.

SEt Open_wait *interval*

Specify the number of seconds to wait between failed device opens.

Full NQS *manager* privileges are required to use this command.

SEt PER_Process Cpu_limit = (*limit*) *queue*

Set a per-process maximum CPU time *limit* for a batch *queue* against which the per-process maximum CPU time limit for a request may be compared. If the local host does not support per-process CPU time limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum CPU time limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process maximum CPU time limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the **LIMITS** section below.

Full NQS *manager* privileges are required to use this command.

SEt PER_Process Memory_limit = (*limit*) *queue*

Set a per-process maximum memory size *limit* for a batch *queue* against which the per-process maximum memory size limit for a request may be compared. If the local host does not support per-process memory size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum memory size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process maximum memory size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the **LIMITS** section below.

Full NQS *manager* privileges are required to use this command.

SEt PER_Process Permfile_limit = (*limit*) *queue*

Set a per-process maximum permanent file size *limit* for a batch *queue* against which the per-process maximum permanent file size limit for a request may be compared. If the local host does not support per-process permanent file size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum permanent file size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process maximum permanent file size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the **LIMITS** section below.

Full NQS *manager* privileges are required to use this command.

SEt PER_Process Tempfile_limit = (*limit*) *queue*

Set a per-process maximum temporary file size *limit* for a batch *queue* against which the per-process maximum temporary file size limit for a request may be compared. If the local host does not support per-process temporary file size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum temporary file size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process maximum temporary file size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the **LIMITS** section below.

Full NQS *manager* privileges are required to use this command.

SEt PER_Request Cpu_limit = (*limit*) *queue*

Set a per-request maximum CPU time *limit* for a batch *queue* against which the per-request maximum CPU time limit for a request may be compared. If the local host does not support per-request CPU time limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-request maximum CPU time limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-request maximum CPU time limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the **LIMITS** section below.

Full NQS *manager* privileges are required to use this command.

SEt PER_Request Memory_limit = (*limit*) *queue*

Set a per-request maximum memory size *limit* for a batch *queue* against which the per-request maximum memory size limit for a request may be compared. If the local host does not support per-request memory size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-request maximum memory size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-request maximum memory size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the **LIMITS** section below.

Full NQS *manager* privileges are required to use this command.

SEt PER_Request Permfile_limit = (*limit*) *queue*

Set a per-request maximum permanent file space *limit* for a batch *queue* against which the per-request maximum permanent file space limit for a request may be compared. If the local host does not support per-request permanent file space limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-request maximum permanent file space limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-request maximum permanent file space limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the **LIMITS** section below.

Full NQS *manager* privileges are required to use this command.

SEt PER_Request Tempfile_limit = (*limit*) *queue*

Set a per-request maximum temporary file space *limit* for a batch *queue* against which the per-request maximum temporary file space limit for a request may be compared. If the local host does not support per-request temporary file space limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-request maximum temporary file space limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-request maximum temporary file space limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the **LIMITS** section below.

Full NQS *manager* privileges are required to use this command.

SEt PIpe_client = (*client*) *queue*

Associate a *pipe client* with a pipe *queue*. *Client* should consist of the absolute path name to the program binary followed by any arguments required by the program.

Full NQS *manager* privileges are required to use this command.

SEt PRiority = *priority queue*

Specify the inter-queue *priority* of a *queue*.

Full NQS *manager* privileges are required to use this command.

SEt Run_limit = *run-limit queue*

Change the *run-limit* of an NQS batch or pipe queue. The *run-limit* determines the maximum number of requests that will be allowed to run in the queue at any given time.

NQS *operator* privileges are required to use this command.

SEt SHell_strategy FIxed = (*shell*)

Specify that *shell* should be used to execute all batch requests. *Shell* must be the absolute path name of a command interpreter.

Full NQS *manager* privileges are required to use this command.

SEt SHell_strategy FRee

Specify that the **free** shell strategy should be used to execute all batch requests. The **free** shell strategy aims at duplicating the shell choice that would have been made if the batch request script had been executed interactively. Under this strategy, the user's *login shell* is allowed to determine the shell to be used to execute the batch request. The user's *login shell* is the shell named within the user's entry in the password file (see **passwd(4)**).

Full NQS *manager* privileges are required to use this command.

SEt SHell_strategy Login

Specify that the **login** shell strategy should be used to execute all batch requests. Under the **login** shell strategy, the user's *login shell* is used to execute the batch request. The *login shell* is the shell named in the password file (see **passwd(4)**).

Full NQS *manager* privileges are required to use this command.

SEt STack_limit = (*limit*) *queue*

Set a per-process maximum stack segment size *limit* for a batch *queue* against which the per-process maximum stack segment size limit for a request may be compared. If the local host does not support per-process stack segment size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum stack segment size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process maximum stack segment size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the **LIMITS** section below.

Full NQS *manager* privileges are required to use this command.

SEt Unrestricted_access *queue*

Specify that no requests will be turned away from *queue* on the grounds of queue access restrictions.

Full NQS *manager* privileges are required to use this command.

SEt Working_set_limit = (*limit*) *queue*

Set a per-process maximum working set size *limit* for a batch *queue* against which the per-process maximum working set size limit for a request may be compared. If the local host does not support per-process working set size limits, then this command will report an error. Otherwise, every batch queue on the local host will have a per-process maximum working set size limit associated with it at all times. If a request already in the queue has asked for more than the new limit, then it will be given a grandfather clause. A request specifying a per-process maximum working set size limit may only enter a batch queue if the queue's limit is greater than or equal to the request's limit. For the syntax of *limit*, see the **LIMITS** section below.

Full NQS *manager* privileges are required to use this command.

SHOw All

Display the standard amount of information about *devices*, *forms*, *limits supported*, *managers*, *parameters*, and *queues*. See below.

SHOw Device [*device-name*]

Display the status of all NQS devices on this host. If a *device-name* is specified, output will be limited to that device.

SHOw Forms

Display the list of valid forms.

SHOw Limits_supported

Display the list of NQS resource limit types which are meaningful on this machine. If a limit type is meaningful on a machine, then the corresponding **qmgr(1M)** commands will allow the association of a limit of that type with any batch queue on that machine. Note that users may request resource limits that are NOT meaningful on the machine where **qsub(1)** is invoked. If the the request is to be executed on a remote machine where the limit is meaningful, then NQS will honor it. Otherwise the unsupported limit is simply ignored.

SHOw LOng Queue [*queue-name* [*user-name*]]
Display in long format the status of all NQS queues on this host. If a *queue-name* is specified, output will be limited to that queue. If a *user-name* is specified, output will downplay any requests not belonging to that user.

SHOw Managers
Display the list of authorized NQS managers.

SHOw Parameters
Display the general NQS parameters.

SHOw Queue [*queue-name* [*user-name*]]
Display the status of all NQS queues on this host. If a *queue-name* is specified, output will be limited to that queue. If a *user-name* is specified, output will downplay any requests not belonging to that user.

SHUtdown [*seconds*]
Shutdown NQS on the local host. A SIGTERM signal is sent to each process of each request presently running. After the specified number of *seconds* of real time have elapsed, a SIGKILL signal is sent to all remaining processes for each request. If a *seconds* value is not specified, then the delay is sixty seconds. Unlike **ABort Queue**, **SHUtdown** requeues all of the requests it kills, provided that the initial SIGTERM signal is caught or ignored by the running request.

NQS *operator* privileges are required to use this command.

STArt Queue *queue*
If the *queue* is already started, then nothing happens. Otherwise, the *queue* is started and requests in the *queue* are eligible for selection.

NQS *operator* privileges are required to use this command.

STOp Queue *queue*
Any requests in the *queue* that are currently running are allowed to complete. All other requests are "frozen" in the *queue*. New requests can still be submitted to the *queue*, but will be "frozen" like the other requests in the *queue*.

NQS *operator* privileges are required to use this command.

Unlock Local_daemon
Remove a lock that has been keeping the NQS local daemon in memory. See **plock(2)**.

NQS *operator* privileges are required to use this command.

QUEUE TYPES

NQS supports four different queue types, that serve to provide four very different functions. These four queue types are known as *batch*, *device*, *pipe*, and *network*.

The queue type of *batch* can only be used to execute NQS *batch requests*. Only NQS *batch requests* created by the *qsub*(1) command can be placed in a *batch queue*.

The queue type of *device* can only be used to execute NQS *device requests*. Only NQS *device requests* created by the *qpr*(1) command can be placed in a *device queue*.

Queues of type: *pipe*, are used to send NQS requests to other *pipe* queues, or to request destination queues of type *batch* or *device*, as appropriate for the request type. In general, *pipe queues* in combination with *network queues*, act as the mechanism that NQS uses to to transport both *batch* and *device* requests to distant queues on other remote machines. It is also perfectly legal for a *pipe queue* to transport requests to queues on the *same* machine.

When a *pipe queue* is defined, it is given a *destination set*, which defines the set of possible destination queues for requests entered in that *pipe queue*. In this manner, it is possible for a *batch* or *device* request to pass through many pipe queues on its way to its ultimate destination, which must eventually be a queue of type *batch* or *device* (matching the request type).

Each *pipe queue* has an associated *server*. For each request handled by a *pipe queue*, the associated server is spawned which must select a queue destination for the request being handled, based on the characteristics of the request, and upon the characteristics of each queue in the *destination set* defined for the pipe queue.

Since a different server can be configured for each pipe queue, and *batch* and *device* queues can be endowed with the *pipeonly* attribute that will only admit requests queued via another *pipe queue*, it is possible for respective NQS installations to use *pipe queues* as a *request class* mechanism, placing requests that ask for different resource allocations in different queues, each of which can have different associated limits and priorities.

It is also completely possible for a *pipe client* (pipe queue server) when handling a request, to discover that no *destination queue* will accept the request, for various reasons which can include insufficient resource limits to execute the request, or a lack of a corresponding account or privilege for queueing at a remote queue. In such circumstances, the request will be deleted, and the user will be notified by mail (see *mail*(1)).

The queue type of *network* as alluded to earlier, is implicitly used by *pipe* queues to pass NQS requests between machines, and is also used to handle queued file transfer operations.

QUEUE ACCESS

NQS supports queue access restrictions. For each queue of queue type other than *network*, access may be either *unrestricted* or *restricted*. If access is *unrestricted*, any request may enter the queue. If access is *restricted*, a request can only enter the queue if the requester or the requester's login group has been given access. Requests submitted by root are an exception; they are always queued, even if root has not explicitly been given access.

LIMITS

NQS supports many batch request resource limit types that can be applied to an NQS batch queue. The configurability of these limits allows an NQS manager to set batch queue-specific resource limits which all batch requests in the queue must adhere to.

The syntax of a *limit* in commands of the form SEt Some_limit = (*limit*) *queue* is quite flexible.

For *finite* CPU time limits, the acceptable syntax is as follows:

[[hours :] minutes :] seconds [.milliseconds]

Whitespace can appear anywhere between the principal tokens, with the exception that no whitespace can appear around the decimal point.

Example time *limit-values are:*

1234 : 58 : 21.29	– 1234 hrs 58 mins 21.290 secs
12345	– 12345 seconds
121.1	– 121.100 seconds
59:01	– 59 minutes and 1 second

For all other *finite* limits (with the exclusion of the *nice-value*), the acceptable syntax is:

.fraction [units]

or

integer [.fraction] [units]

where the *integer* and *fraction* tokens represent strings of up to eight decimal digits, denoting the obvious values. In both cases, the *units* of allocation may also be specified as one of the case insensitive strings:

b	– bytes
w	– words
kb	– kilobytes (2^10 bytes)
kw	– kilowords (2^10 words)
mb	– megabytes (2^20 bytes)
mw	– megawords (2^20 words)
gb	– gigabytes (2^30 bytes)
gw	– gigawords (2^30 words)

In the absence of any *units* specification, the units of *bytes* are assumed.

For all limit types with the exception of the *nice-value*, it is possible to state that no limit should be applied. This is done by specifying a *limit* of "unlimited", or any initial substring thereof.

The complications caused by *batch request* resource limits first show up when queueing a *batch request* in a *batch queue*. This operation is described in the following paragraphs.

If a batch request specifies a limit that cannot be enforced by the underlying UNIX implementation, then the limit is simply ignored, and the batch request will operate as though there were no limit (other than the obvious physical maximums), placed upon that resource type. (See the *qlimit*(1) command to find out what limits are supported by a given machine.)

For each remaining *finite* limit that can be supported by the underlying UNIX implementation that is **not** a CPU *time-limit*, or UNIX *nice-value*, the *limit-value* is internally converted to the units of *bytes* or *words*, whichever is more appropriate for the underlying machine architecture.

As an example, a *per-process memory size limit value* of 321 megabytes would be interpreted as 321 x 2^20 bytes, provided that the underlying machine architecture was capable of directly addressing single bytes. Thus the original limit *coefficient* of 321 would become 321 x 2^20. On a machine that was only capable of addressing words, the appropriate conversion of 321 x 2^20

bytes / #of-bytes-per-word would be performed.

If the result of such a conversion would cause overflow when the coefficient was represented as a *signed-long integer* on the supporting hardware, then the coefficient is replaced with the coefficient of: of 2^N-1 where N is equal to the number of bits of precision in a signed long integer. For typical 32-bit machines, this *default extreme limit* would therefore be $2^{31}-1$ bytes. For word addressable machines in the supercomputer class supporting 64-bit long integers, the *default extreme limit* would be $2^{63}-1$ words.

Lastly, some implementations of UNIX reserve coefficients of the form: 2^N-1 as synonomous with infinity, meaning no limit is to be applied. For such UNIX implementations, NQS further decrements the *default extreme limit* so as to not imply infinity.

The identical internal conversion process as described in the preceding paragraphs is also performed for all *finite limit-values* specified with a particular batch request.

After each applicable request *limit* has been converted as described above, the resulting *limit* is then compared against the corresponding *limit* as configured for the destination batch queue. If the corresponding batch queue *limit* for all batch request *limits is defined as unlimited*, or is *greater than* or *equal to* the corresponding batch request *limit*, then the request can be successfully queued, provided that no other anomalous conditions occur. For requests that ask for a *limit* of infinity, the corresponding queue *limit* must also be configured as infinity.

These resource limit checks are performed irrespective of the batch request arrival mechanism, either by a direct use of the *qsub*(1) command, or by the indirect placement of a batch request into a batch queue via a *pipe* queue. It is impossible for a batch request to be queued in an NQS batch queue if *any* of these resource limit checks fail.

Finally, if a request fails to specify a *limit* for a resource limit type that is supported on the execution machine, then the corresponding *limit* as configured for the destination queue becomes the *limit* for the request.

Upon the successful queueing of a request in a batch queue, the set of limits under which the request will execute is frozen, and will not be modified by subsequent *qmgr*(1M) commands that alter the limits of the containing batch queue.

SEE ALSO

passwd(4), plock(2), qdel(1), qdev(1), qlimit(1), qpr(1), qstat(1), and qsub(1)
in the *NPSN UNIX User Reference Manual.*

NPSN HISTORY

Origin: Sterling Software Incorporated

August 1985 – Brent Kingsbury, Sterling Software
Original release.

May 1986
Second release.

NAME

qpr – submit a hardcopy print request to NQS

SYNOPSIS

qpr [**–a** date-time] [**–f** form-name] [**–mb**] [**–me**]
[**–mu** user-name] [**–n** number-of-copies] [**–p** priority]
[**–q** queue-name] [**–r** request-name] [**–z**] [files]

DESCRIPTION

Qpr places the named files in a *Network Queueing System* (NQS) queue to be printed by a device such as a line printer or laser printer. If no files are specified, *qpr* will read from the standard input.

In the absence of the **–z** flag, *qpr* will print a *request-id* on the standard output, upon successful queueing of a request. This *request-id* can be compared with what is reported by *qdev*(1) and *qstat*(1) to find out what happened to a request, and given as an argument to *qdel*(1) to delete a request. A *request-id* is always of the form: *seqno.hostname* where *seqno* refers to the sequence number assigned to the request by NQS, and *hostname* refers to the name of originating local machine. This identifier is used throughout NQS to uniquely identify the request, no matter where it is in the network.

The following options to *qpr* may appear in any order and may be intermixed with file names.

–a *date-time*

Submit at the specified date and/or time. In the absence of this flag, *qpr* will submit the request immediately.

If a *date-time* specification is comprised of two or more tokens separated by whitespace characters, then the *date-time* specification must be placed within double quotes as in: **–a** *"July, 4, 2026 12:31-EDT"*, or otherwise escaped such that the shell will interpret the entire *date-time* specification as a single lexical token.

The syntax accepted for the *date-time* parameter is relatively flexible. Unspecified date and time values default to an appropriate value (e.g. if no date is specified, then the current month, day, and year are assumed).

A date can be specified as a month and day (current year assumed). The year can also be explicitly specified. It is also possible to specify the date as a weekday name (e.g. "Tues"), or as one of the strings "today" or "tomorrow". Weekday names and month names can be abbreviated by any three character (or longer) prefix of the actual name. An optional period can follow an abbreviated month or day name.

Time of day specifications can be given using a twenty-four hour clock, or "am" and "pm" specifications may be used alternatively. In the absence of a meridian specification, a twenty-four hour clock is assumed.

It should be noted that the time of day specification is interpreted using the precise meridian definitions whereby "12am" refers to the twenty-four hour clock time of 0:00:00, "12m" refers to noon, and "12-pm" refers to 24:00:00. Alternatively, the phrases "midnight" and "noon" are accepted as time of day specifications, where "midnight" refers to the time of 24:00:00.

A timezone may also appear at any point in the *date-time* specification. Thus, it is legal to say: "April 1, 1987 13:01-PDT". In the absence of a timezone specification, the local timezone is assumed, with daylight savings time being inferred when appropriate, based on the date specified.

All alphabetic comparisons are performed in a case insensitive fashion such that both "WeD" and "weD" refer to the day of Wednesday.

Some valid *date-time* examples are:

01-Jan-1986 12am, PDT
Tuesday, 23:00:00
11pm tues.
tomorrow 23-MST

−f *form-name*

Limit the set of acceptable devices to those devices which are loaded with the forms: *form-name*. In the absence of this flag, *qpr* will submit the request only to a device that is loaded with the *default* forms. If there is no *default* forms defined, the request will be submitted to the appropriate output device without regard to the forms configured for the device.

In any case, only those devices associated with the chosen queue will be considered.

−mb Send mail to the user on the originating machine when the request begins execution. If the **−mu** flag is also present, then mail is sent to the user specified for the **−mu** flag instead of to the invoking user.

−me Send mail to the invoker on the originating machine when the request has ended execution. If the **−mu** flag is also present, then mail is sent to the user specified for the **−mu** flag instead of to the invoking user.

−mu *user-name*

Specify that any mail concerning the request should be delivered to the user *user-name*. *User-name* may be formatted either as *user* (containing no '@' characters), or as *user@machine*. In the absence of this flag, any mail concerning the request will be sent to the invoker on the originating machine.

−n *number-of-copies*

Print *number-of-copies* copies. The default is one.

−p *priority*

Assign an intra-queue priority to this request. The specified *priority* must be an integer, and must be in the range [0..63], inclusive. A value of 63 defines the highest *intra-queue* request priority, while a value of 0 defines the lowest. This priority does not determine the execution priority of the request. This priority is only used to determine the relative ordering of requests within a queue.

When a request is added to a queue, it is placed at a specific position within the queue such that it appears ahead of all existing requests whose priority is less than the priority of the new request. Similarly, all requests with a higher priority will remain ahead of the new request when the queueing process is complete. When the priority of the new request is equal to the priority of an existing request, the existing request takes precedence over the new request.

If no *intra-queue* priority is chosen by the user, then NQS assigns a default value.

−q *queue-name*

Specify the queue to which the device request is to be submitted. If no **−q** *queue-name* specification is given, then the user's environment variable set is searched for the variable: **QPR_QUEUE**. If this environment variable is found, then the character string value for **QPR_QUEUE** is presumed to name the queue to which the request should be submitted. If the **QPR_QUEUE** environment variable is not found, then the request will be submitted to the default device request queue, *if* defined by the local system administrator. Otherwise, the request cannot be queued, and an appropriate error message is displayed to this effect.

-r *request-name*

Assign a name to this request. In the absence of an explict **-r** *request-name* specification, the *request-name* defaults to the name of the first print file (leading path name removed) specified on the command line. If no print files were specified, then the default *request-name* assigned to the request is **STDIN**.

In all cases, if the *request-name* is found to begin with a digit, then the character 'R' is pre-pended to prevent a *request-name* from beginning with a digit. All *request-names* are truncated to a maximum length of 15 characters.

Be sure not to confuse *request-name* with *request-id*.

-s Submit the request silently. If the request is submitted successfully, nothing will be written to stdout or stderr.

QUEUE ACCESS

NQS supports queue access restrictions. For each queue of queue type other than *network*, access may be either *unrestricted* or *restricted*. If access is *unrestricted*, any request may enter the queue. If access is *restricted*, a request can only enter the queue if the requester or the requester's login group has been given access to that queue (see *qmgr*(1M)). Requests submitted by root are an exception; they are always queued, even if root has not explicitly been given access.

Use *qstat*(1) to determine who has access to a particular queue.

SEE ALSO

mail(1), qdel(1), qdev(1), qlimit(1), qstat(1), and qsub(1)
in the *NPSN UNIX System Programmer Reference Manual.*
qmgr(1M) in the *NPSN UNIX System Administrator Reference Manual.*

NPSN HISTORY

Origin: Sterling Software Incorporated

May 1986 — Robert Sandstrom, Sterling Software
Original release.

NAME

qstat – display status of NQS queue(s)

SYNOPSIS

qstat [**–a**] [**–l**] [**–m**] [**–u** user-name] [**–x**]
[queue-name ...] [queue-name@host-name ...]

DESCRIPTION

Qstat displays the status of Network Queueing System (NQS) queues.

If no queues are specified, then the current state of each NQS queue on the local host is displayed. Otherwise, information is displayed for the specified queues only. Queues may be specified either as *queue-name* or *queue-name@host-name*. In the absence of a *host-name* specifier, the local host is assumed.

For each selected queue, *qstat* displays a *queue header* (information about the queue itself) followed by information about requests in the queue. Ordinarily, *qstat* shows only those requests belonging to the invoker. The following flags are available:

-a Shows all requests.

-l Requests are shown in a long format.

-m Requests are shown in a medium-length format.

-u *user-name*
Shows only those requests belonging to *user-name*.

-x The queue header is shown in an extended format.

The *queue header* always includes the queue-name, queue type, queue status (see below), an indication of whether or not the queue is *pipeonly* (accepts requests from pipe queues only), and the number of requests in the queue. An extended queue header goes on to display the priority and run limit of a queue, as well as the access restrictions, cumulative use statistics, server and destinations (if a pipe queue), queue to device mappings (if a device queue), and resource limits (if a batch queue).

By default, *qstat* displays the following information about a request: the *request-name*, the *request-id*, the owner, the relative request priority, and the current request state (see below). For running requests, the process group is also shown, as soon as this information becomes available to the local NQS daemon.

Qstat -m shows the following additional information: If the request was submitted with the constraint that it not run before a certain time and date, then the constraining time and date will also be displayed.

Qstat -l shows the time at which the request was created, an indication of whether or not mail will be sent, where mail will be sent, and the username on the originating machine. If a batch queue is being examined, resource limits, planned disposition of stderr and stdout, any advice concerning the command interpreter, and the umask value are shown. If a device queue is being examined, the requested forms are shown.

It must be understood that the relative ordering of requests within a queue does not always determine the order in which the requests will be run. The NQS request scheduler is allowed to make exceptions to the request ordering for the sake of efficient machine resource usage. However, requests appearing near the beginning of the queue have higher priority than requests appearing later, and will usually be run before requests appearing later on in the queue.

QUEUE STATE

The general state of a queue is defined by two principal properties of the queue.

The first property determines whether or not requests can be submitted to the queue. If they can, then the queue is said to be *enabled*. Otherwise the queue is said to be *disabled*. One of the words CLOSED, ENABLED, or DISABLED will appear in the queue status field to indicate the respective queue states of: enabled (with no local NQS daemon), enabled (and local NQS daemon is present), and disabled. Requests can only be submitted to the queue if the queue is enabled, and the local NQS daemon is present.

The second principal property of a queue determines if requests which are ready to run, but are not now presently running, will be allowed to run upon the completion of any currently running requests, and whether any requests are presently running in the queue.

If queued requests not already running are blocked from running, and no requests are presently executing in the queue, then the queue is said to be *stopped*. If the same situation exists with the difference that at least one request is running, then the queue is said to be *stopping*, where the requests presently executing will be allowed to complete execution, but no new requests will be spawned.

If queued requests ready to run are only prevented from doing so by the NQS request scheduler, and one or more requests are presently running in the queue, then the queue is said to be *running*. If the same circumstances prevail with the exception that no requests are presently running in the queue, then the queue is said to be *inactive*. Finally, if the NQS daemon for the local host upon which the queue resides is not running, but the queue would otherwise be in the state of *running* or *inactive*, then the queue is said to be *shutdown*. The queue states describing the second principal property of a queue are therefore respectively displayed as STOPPED, STOPPING, RUNNING, INACTIVE, and SHUTDOWN.

REQUEST STATE

The state of a request may be *arriving*, *holding*, *waiting*, *queued*, *staging*, *routing*, *running*, *departing*, or *exiting*. A request is said to be *arriving* if it is being enqueued from a remote host. *Holding* indicates that the request is presently prevented from entering any other state (including the *running* state), because a *hold* has been placed on the request. A request is said to be *waiting* if it was submitted with the constraint that it not run before a certain date and time, and that date and time have not yet arrived. *Queued* requests are eligible to proceed (by *routing* or *running*). When a request reaches the head of a pipe queue and receives service there, it is *routing*. A request is *departing* from the time the pipe queue turns to other work until the request has arrived intact at its destination. *Staging* denotes a *batch* request that has not yet begun execution, but for which input files are being brought on to the execution machine. A *running* request has reached its final destination queue, and is actually executing. Finally, *exiting* describes a batch request that has completed execution, and will exit from the system after the required output files have been returned (to possibly remote machines).

Imagine a batch request originating on a workstation, destined for the batch queue of a computation engine, to be run immediately. That request would first go through the states *queued*, *routing*, and *departing* in a local pipe queue. Then it would disappear from the pipe queue. From the point of view of a queue on the computation engine, the request would first be *arriving*, then *queued*, *staging* (if required by the batch request), *running*, and finally *exiting*. Upon completion of the *exiting* phase of execution, the batch request would disappear from the batch queue.

CAVEATS

NQS is not finished, and continues to undergo development. Some of the request states shown above may or may not be supported in your version of NQS.

SEE ALSO

qdel(1), qdev(1), qlimit(1), qpr(1), and qsub(1)
in the *NPSN UNIX System Programmer Reference Manual.*
qmgr(1M) in the *NPSN UNIX System Administrator Reference Manual.*

NAME

qsub – submit an NQS batch request.

SYNOPSIS

qsub [flags] [script-file]

DESCRIPTION

Qsub submits a batch request to the Network Queueing System (NQS).

If no *script-file* is specified, then the set of commands to be executed as a batch request is taken directly from the standard input file (*stdin*). In all cases however, the *script file* is spooled, so that later changes will **not** affect previously queued batch requests.

All of the flags that can be specified on the command line can also be specified within the first comment block inside the batch request *script file* as *embedded default flags*. Such flags appearing in the batch request *script file* set default characteristics for the batch request. If the same flag is specified on the command line, then the command line flag (and any associated value) takes precedence over the *embedded* flag. See the section entitled: **LONG DESCRIPTION** for more information on *embedded default flags*.

What follows is a terse definition of the flags implemented by the *Qsub* command (see the section: **LONG DESCRIPTION** for the complete definition and syntax used for each of these flags).

–a – run request after stated time
–e – direct stderr output to stated destination
–eo – direct stderr output to the stdout destination
–ke – keep stderr output on the execution machine
–ko – keep stdout output on the execution machine
–lc – establish per-process corefile size limit
–ld – establish per-process data-segment size limits
–lf – establish per-process permanent-file size limits
–lF – establish per-request permanent-file space limits
–lm – establish per-process memory size limits
–lM – establish per-request memory space limits
–ln – establish per-process nice execution value limit
–ls – establish per-process stack-segment size limits
–lt – establish per-process CPU time limits
–lT – establish per-request CPU time limits
–lv – establish per-process temporary-file size limits
–lV – establish per-request temporary-file space limits
–lw – establish per-process working set limit
–mb – send mail when the request begins execution
–me – send mail when the request ends execution
–mu – send mail for the request to the stated user
–nr – declare that batch request is not restartable
–o – direct stdout output to the stated destination
–p – specify intra-queue request priority
–q – queue request in the stated queue
–r – assign stated request name to the request
–re – remotely access the stderr output file
–ro – remotely access the stdout output file
–s – specify shell to interpret the batch request script
–x – export all environment variables with request
–z – submit the request silently

LONG DESCRIPTION

As described above, it is possible to specify *default* flags within the batch request *script file* that configure the default behavior of the batch request. The algorithm used to scan for such *embedded default flags* within an NQS batch request script file is as follows:

1. Read the first line of the *script file*.
2. If the current line contains only whitespace characters, or the first non-whitespace character of the line is ":", then goto step 7.
3. If the first non-whitespace character of the current line is not a "#" character, then goto step 8.
4. If the second non-whitespace character in the current line is *not* the "@" character, or the character immediately following the second non-whitespace character in the current line is *not* a "$", then goto step 7.
5. If no "-" is present as the character *immediately* following the "@$" sequence, then goto step 8.
6. Process the *embedded* flag, stopping the parsing process upon reaching the end of the line, or upon reaching the first unquoted "#" character.
7. Read the next *script file* line. Goto step 2.
8. End. No more *embedded* flags will be recognized.

Here is an example of the use of *embedded* flags within the *script file*.

```
#
# Batch request script example:
#
# @$-a "11:30pm EDT" -lt "21:10, 20:00"
#          # Run request after 11:30 EDT by default,
#          # and set a maximum per-process CPU time
#          # limit of 21 minutes and ten seconds.
#          # Send a warning signal when any process
#          # of the running batch request consumes
#          # more than 20 minutes of CPU time.
# @$-lT 1:45:00
#          # Set a maximum per-request CPU time limit
#          # of one hour, and 45 minutes. (The
#          # implementation of CPU time limits is
#          # completely dependent upon the UNIX
#          # implementation at the execution
#          # machine.)
# @$-mb -me   # Send mail at beginning and end of
#          # request execution.
# @$-q batch1 # Queue request to queue: batch1 by
#          # default.
# @$         # No more embedded flags.
#
make all
```

The following paragraphs give the detailed descriptions of the *flags* supported by the *Qsub* command.

−a *date-time*

Do not run the batch request before the specified date and/or time. If a *date-time* specification is comprised of two or more tokens separated by whitespace characters, then the *date-time* specification must be placed within double quotes as in: −a *"July, 4, 2026 12:31-EDT"*, or otherwise escaped such that *Qsub* and the shell will interpret the entire *date-time* specification as a single character string. This restriction also applies when an embedded default −a flag with accompanying *date-time* specification appears within the batch request *script file*.

The syntax accepted for the *date-time* parameter is relatively flexible. Unspecified date and time values default to an appropriate value (e.g. if no date is specified, then the current month, day, and year are assumed).

A date may be specified as a month and day (current year assumed), or the year can also be explicitly specified. It is also possible to specify the date as a weekday name (e.g. "Tues"), or as one of the strings: "today", or "tomorrow". Weekday names and month names can be abbreviated by any three character (or longer) prefix of the actual name. An optional period can follow an abbreviated month or day name.

Time of day specifications can be given using a twenty-four hour clock, or "am" and "pm" specifications may be used alternatively. In the absence of a meridian specification, a twenty-four hour clock is assumed.

It should be noted that the time of day specification is interpreted using the precise meridian definitions whereby "12am" refers to the twenty-four hour clock time of 0:00:00, "12m" refers to noon, and "12-pm" refers to 24:00:00. Alternatively, the phrases "midnight" and "noon" are accepted as time of day specifications, where "midnight" refers to the time of 24:00:00.

A timezone may also appear at any point in the *date-time* specification. Thus, it is legal to say: "April 1, 1987 13:01-PDT". In the absence of a timezone specification, the local timezone is assumed, with daylight savings time being inferred when appropriate, based on the date specified.

All alphabetic comparisons are performed in a case insensitive fashion such that both "WeD" and "weD" refer to the day of Wednesday.

Some valid *date-time* examples are:

01-Jan-1986 12am, PDT
Tuesday, 23:00:00
11pm tues.
tomorrow 23-MST

−e *[machine:][[/]path/] stderr-filename*

Direct output generated by the batch request which is sent to the *stderr* file to the named *[machine:][[/]path/] stderr-filename*.

The brackets "[" and "]" enclose optional portions of the *stderr* destination *machine*, *path*, and *stderr-filename*.

If no explicit *machine* destination is specified, then the destination machine defaults to the machine that originated the batch request, or to the machine that will eventually run the request, depending on the respective absence, or presence of the −ke flag.

If no *machine* destination is specified, and the path/filename does not begin with a "/", then the current working directory is prepended to create a fully qualified path

name, provided that the −ke (keep stderr) flag is also absent. In all other cases, any partial path/filename is interpreted relative to the user's home directory on the *stderr* destination machine.

This flag cannot be specified when the −eo flag option is also present.

If the −eo and −e *[machine:][[/]path/] stderr-filename* flag options are not present, then all *stderr* output for the batch request is sent to the file whose name consists of the first seven characters of the *request-name* followed by the characters: ".e", followed by the request sequence number portion of the *request-id* discussed below. In the absence of the −ke flag, this default *stderr* output file will be placed on the machine that originated the batch request in the current working directory, as defined when the batch request was first submitted. Otherwise, the file will be placed in the user's home directory on the execution machine.

−eo Direct all output that would normally be sent to the *stderr* file to the *stdout* file for the batch request. This flag cannot be specified when the −e *[machine:][[/]path/] stderr-filename* flag option is also present.

−ke In the absence of an explicit *machine* destination for the *stderr* file produced by a batch request, the *machine* destination chosen for the *stderr* output file is the machine that originated the batch request. In some cases however, this behavior may be undesirable, and so the −ke flag can be specified which instructs NQS to leave any *stderr* output file produced by the request on the machine that actually *executed* the batch request.

This flag is meaningless if the −eo flag is specified, and cannot be specified if an explicit *machine* destination is given for the *stderr* parameter of the −e flag.

−ko In the absence of an explicit *machine* destination for the *stdout* file produced by a batch request, the *machine* destination chosen for the *stdout* output file is the machine that originated the batch request. In some cases however, this behavior may be undesirable, and so the −ko flag can be specified which instructs NQS to leave any *stdout* output file produced by the request on the machine that actually *executed* the batch request.

This flag cannot be specified if an explicit *machine* destination is given for the *stdout* parameter of the −o flag.

−lc *per-process corefile size limit*

Set a *per-process* maximum *core file size limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to exit creating a core file whose size would exceed the maximum *per-process core file size limit* for the request, then the core file image of the aborting process will be reduced to the necessary size by an algorithm dependent upon the underlying UNIX implementation.

Not all UNIX implementations support *per-process corefile size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process corefile size limit*.

−ld *per-process data-segment size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *data-segment size limit* for all processes that constitute the running batch request. If any process comprising the

running request exceeds the maximum *per-process data-segment size-limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process data-segment warning size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default −ld flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-process data-segment size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process data-segment size limit*.

−lf *per-process permanent-file size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *permanent-file size limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to write to a permanent file such that the file size would increase beyond the maximum *per-process permanent-file size limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process warning permanent-file size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default −lf flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-process permanent-file size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

At the time of this writing, the author was unaware of any UNIX implementation that made a distinction at the kernel level, between *permanent*, and *temporary* files. While it is certainly possible to construct a *pseudo-temporary* file by first creating it, and then unlinking its pathname, the disk space allocated for such a file will be allocated from the same pool of disk space that all other UNIX files are allocated from. Furthermore, such a file will be subject to the same quota enforcement mechanisms, if any are provided by the underlying UNIX implementation, that all other UNIX files are created under.

For all UNIX implementations that do not support a distinction between *permanent*, and *temporary* files at the kernel level, this limit is interpreted as a *per-process file size limit*, with the word *permanent* removed from the definition.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process permanent-file size limit*.

−lF *per-request permanent-file space limit [, warn-limit]*

Set a *per-request* maximum and an optional warning cumulative *permanent-file space limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to write to a permanent file such that the file space consumed by all permanent files opened for writing by all of the processes in the batch request, would increase beyond the maximum *per-request permanent-file space limit* for the request, then all of the processes in the request will be terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-request warning permanent-file space limits*. When such a warning limit is exceeded, a signal is delivered to one or more of the processes comprising the running request, depending upon the underlying UNIX implementation.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **−lF** flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-request permanent-file space limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

At the time of this writing, the author was unaware of any UNIX implementation that made a distinction at the kernel level, between *permanent*, and *temporary* files. While it is certainly possible to construct a *pseudo-temporary* file by first creating it, and then unlinking its pathname, the disk space allocated for such a file will be allocated from the same pool of disk space that all other UNIX files are allocated from. Furthermore, such a file will be subject to the same quota enforcement mechanisms, if any are provided by the underlying UNIX implementation, that all other UNIX files are created under.

For all UNIX implementations that do not support a distinction between *permanent*, and *temporary* files at the kernel level, this limit is interpreted as a *per-request file space limit*, with the word *permanent* removed from the definition.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-request permanent-file space limit*.

−lm *per-process memory size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *memory size limit* for all processes that constitute the running batch request. If any process comprising the running request exceeds the maximum *per-process memory size limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process warning memory size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default −lm flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-process memory size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process memory size limit*.

−lM *per-request memory space limit [, warn-limit]*

Set a *per-request* maximum and an optional warning cumulative *memory space limit* for all processes that constitute the running batch request. If the sum of all memory consumed by all of the processes comprising the running request exceeds the maximum *per-request memory space limit* for the request, then all of the processes in the request will be terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-request warning memory size limits*. When such a warning limit is exceeded, a signal is delivered to one or more of the processes comprising the running request, depending upon the underlying UNIX implementation.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default −lM flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-request memory space limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-request memory space limit*.

−ln *per-process nice value limit*

Set a *per-process nice value* for all processes comprising the running batch request.

At present, all UNIX implementations support the use of an integer called the *nice* value, which determines the *execution-time* priority of a process relative to all other processes in the system. By letting the user set a limit on the *nice* value for all processes comprising the running request, a user can cause a request to consume less (or more) of the CPU resource presented by the execution machine.

This is particularly useful when a user wishes to execute a CPU intensive batch request on a machine running interactive processes. By setting a low *execution-time priority*, a user can make a long running batch request give way to more interactive processes during the daytime, while the coming of the nighttime hours with typically smaller process loads will allow such a request to gain more and more of the CPU resource. In this way, long running batch requests can be polite to their more transient, interactive neighbor processes.

The only quirk associated with this flag results from the peculiar choice of *nice* values, implemented by the standard UNIX implementations. In general, increasingly *negative* nice values cause the relative execution priority of a process to *increase*, while increasingly *positive* nice values causes the relative priority to *decrease*! Thus, a *nice value* limit specification of: "-ln -10" is greedier than a *nice value* limit specification of: "-ln 0".

Since varying UNIX implementations often support a different finite range of *nice values*, NQS allows the specification of *nice values* that can eventually turn out to be outside the limits for the UNIX implementation running at the *execution* machine. In such cases, NQS will simply bind the specified *nice value* limit to within the necessary range as appropriate.

Lastly, any *nice value* specified by the use of this flag must be acceptable to the batch queue in which the request is ultimately placed (see the section entitled **LIMITS** for more information).

—ls *per-process stack-segment size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *stack-segment size limit* for all processes that constitute the running batch request. If any process comprising the running request exceeds the maximum *per-process stack-segment size limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process warning stack-segment size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default —ls flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-process stack-segment size limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process stack-segment size limit*.

—lt *per-process CPU time limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *CPU time limit* for all processes that constitute the running batch request. If any process comprising the running request exceeds the maximum *per-process CPU time limit* for the request, then that process is terminated by a signal chosen by the underlying UNIX

implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process CPU warning time limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **−lt** flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-process CPU time limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process CPU time limit*.

−lT *per-request CPU time limit [, warn-limit]*

Set a *per-request* maximum and an optional warning cumulative *CPU time limit* for all of the processes that constitute the running batch request. If the sum of the CPU times consumed by all of the processes in the batch request exceeds the maximum *per-request CPU time limit* for the request, then all of the processes in the request will be terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-request CPU warning time limits*. When such a warning limit is exceeded, a signal is delivered to one or more of the processes comprising the running request, depending upon the underlying UNIX implementation.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default **−lT** flag with its associated limit value(s) appears within the batch request *script file*.

Not all UNIX implementations support *per-request CPU time limits*. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-request CPU time limit*.

−lv *per-process temporary file size limit [, warn-limit]*

Set a *per-process* maximum and an optional warning *temporary (volatile) file size limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to write to a *temporary* file such that the file size would increase beyond the maximum *per-process temporary-file size limit* for the request, then that process is terminated by a signal chosen by the underlying

UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-process warning temporary-file size limits*. When a warning limit is exceeded, a signal as determined by the underlying UNIX implementation is delivered to the offending process.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default −lv flag with its associated limit value(s) appears within the batch request *script file*.

At the time of this writing, no UNIX operating system known to the author supported a distinction at the kernel level between *permanent* and *temporary files*. Certainly, a *pseudo-temporary* file can be constructed by creating it, and then unlinking its pathname. However, the file space allocated for such a file will be allocated from the same pool of disk space that all other UNIX files are allocated from.

Until a mechanism is implemented in the kernel that knows about *permanent* and *temporary* files, this limit cannot be supported in the sense most useful for batch requests, namely the strict enforcement of disk quotas for *permanent* versus *temporary* files.

Until such a time, this limit will simply be ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process temporary-file size limit*.

−lV *per-request temporary file space limit [, warn-limit]*

Set a *per-request* maximum and an optional warning cumulative *temporary (volatile) file space limit* for all processes that constitute the running batch request. If any process comprising the running request attempts to write to a *temporary* file such that the file space consumed by all *temporary* files opened for writing by all of the processes in the batch request would increase beyond the maximum *per-request temporary-file space limit* for the request, then all of the processes in the request will be terminated by a signal chosen by the underlying UNIX implementation.

The ability to specify an optional warning limit exists for those UNIX operating systems that support *per-request warning temporary-file space limits*. When such a warning limit is exceeded, a signal is delivered to one or more of the processes comprising the running request, depending upon the underlying UNIX implementation.

If a maximum limit (and optional warning limit) specification is comprised of two or more tokens separated by whitespace characters, then the specification must be enclosed within double quotes, or otherwise escaped such that *Qsub* and the shell will interpret the entire specification as a single character string token. This caveat also applies when an embedded default −lV flag with its associated limit value(s) appears within the batch request *script file*.

At the time of this writing, no UNIX operating system known to the author supported a distinction at the kernel level between *permanent* and *temporary files*. Certainly, a *pseudo-temporary* file can be constructed by creating it, and then unlinking its pathname. However, the file space allocated for such a file will be allocated from the same pool of disk space that all other UNIX files are allocated from.

Until a mechanism is implemented in the kernel that knows about *permanent* and *temporary* files, this limit cannot be supported in the sense most useful for batch requests, namely the strict enforcement of disk quotas for *permanent* versus *temporary* files.

Until such a time, this limit will simply be ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *temporary-file space limit*.

−lw *per-process working set size limit*

Set a *per-process* maximum *working set size limit* for all processes that constitute the running batch request.

Not all UNIX implementations support *per-process working set size limits*, and such a limit only makes sense in the context of a paged virtual memory machine. If a batch request specifies this limit, and the machine upon which the batch request is eventually run does not support the enforcement of this limit, then the limit is simply ignored.

See the section entitled **LIMITS** for more information on the implementation of batch request limits, and for a description of the precise syntax of a *per-process working set size limit*.

−mb Send mail to the user on the originating machine when the request begins execution. If the **−mu** flag is also present, then mail is sent to the user specified for the **−mu** flag instead of to the invoking user.

−me Send mail to the user on the originating machine when the request has ended execution. If the **−mu** flag is also present, then mail is sent to the user specified for the **−mu** flag instead of to the invoking user.

−mu *user-name*

Specify that any mail concerning the request should be delivered to the user *user-name*. *User-name* may be formatted either as *user* (containing no '@' characters), or as *user@machine*. In the absence of this flag, any mail concerning the request will be sent to the invoker on the originating machine.

−nr Declare that the request is non-restartable. If this flag is specified, then the request will not be restarted by NQS upon system boot if the request was running at the time of an NQS shutdown or system crash.

By default, NQS assumes that all requests are restartable, with the caveat that it is the responsibility of the user to ensure that the request will execute correctly if restarted, by the use of appropriate programming techniques.

Requests that are not running are always preserved across host crashes and NQS shutdowns for later requeueing, with or without this flag.

When NQS is shutdown via an operator command to the *qmgr*(1M) NQS control program, a **SIGTERM** signal is sent to all processes comprising all running NQS requests on the local host, and all queued NQS requests are barred from beginning execution. After a finite number of seconds have elapsed (typically sixty, but this value can be overridden by the operator), all remaining processes comprising all remaining running NQS requests are killed by the signal: **SIGKILL**.

For an NQS request to be properly restarted after an NQS shutdown, the **−nr** flag must not be specified, and the spawned batch request shell must ignore **SIGTERM** signals (which is done by default). The spawned batch request shell must also not

exit before the final **SIGKILL** arrives. Since the batch request shell is simply spawning commands and programs, waiting for their completion, this implies that the commands and programs being executed by the batch request shell must also be immune to **SIGTERM** signals, saving state as appropriate before being killed by the final **SIGKILL** signal.

See the **CAVEATS** section below for more discussion concerning the restartability of NQS batch requests.

−o *[machine:][[/]path/] stdout-filename*

Direct output generated by the batch request which is sent to the *stdout* file to the named *[machine:][[/]path/] stdout-filename*.

The brackets "[" and "]" enclose optional portions of the *stdout* destination *machine*, *path*, and *stdout-filename*.

If no explicit *machine* destination is specified, then the destination machine defaults to the machine that originated the batch request, or to the machine that will eventually run the request, depending on the respective absence, or presence of the **−ko** flag.

If no *machine* destination is specified, and the path/filename does not begin with a "/", then the current working directory is prepended to create a fully qualified path name, provided that the **−ko** (keep stdout) flag is also absent. In all other cases, any partial path/filename is interpreted relative to the user's home directory on the *stdout* destination machine.

If no **−o** *[machine:][[/]path/] stdout-filename* flag is specified, then all *stdout* output for the batch request is sent to the file whose name consists of the first seven characters of the *request-name* followed by the characters: ".o", followed by the request sequence number portion of the *request-id* discussed below. In the absence of the **−ko** flag, this default *stdout* output file will be placed on the machine that originated the batch request in the current working directory, as defined when the batch request was first submitted. Otherwise, the file will be placed in the user's home directory on the execution machine.

−p *priority* Explicitly assign an *intra-queue* priority to the request. The specified *priority* must be an integer, and must be in the range [0..63], inclusive. A value of 63 defines the highest *intra-queue* request priority, while a value of 0 defines the lowest. This priority does **not** determine the execution priority of the request. This priority is only used to determine the relative ordering of requests within a queue.

When a request is added to a queue, it is placed at a specific position within the queue such that it appears ahead of all existing requests whose priority is less than the priority of the new request. Similarly, all requests with a higher priority will remain ahead of the new request when the queueing process is complete. When the priority of the new request is equal to the priority of an existing request, the existing request takes precedence over the new request.

If no *intra-queue* priority is chosen by the user, then NQS assigns a default value.

−q *queue-name*

Specify the queue to which the batch request is to be submitted. If no **−q** *queue-name* specification is given, then the user's environment variable set is searched for the variable: **QSUB_QUEUE**. If this environment variable is found, then the character string value for **QSUB_QUEUE** is presumed to name the queue to which the request should be submitted. If the **QSUB_QUEUE** environment variable is not found, then the request will be submitted to the default batch request queue, *if*

defined by the local system administrator. Otherwise, the request cannot be queued, and an appropriate error message is displayed to this effect.

−r *request-name*

Assign the specified *request-name* to the request. In the absence of an explict −r *request-name* specification, the *request-name* defaults to the name of the *script file* (leading path name removed) given on the command line. If no *script file* was given, then the default *request-name* assigned to the request is **STDIN**.

In all cases, if the *request-name* is found to begin with a digit, then the character 'R' is prepended to prevent a *request-name* from beginning with a digit. All *request-names* are truncated to a maximum length of 15 characters.

−re

By default, all output generated by a batch request sent to the *stderr* file is temporarily into a file residing in a protected directory on the machine that executes the request. When the batch request completes execution, this file is then spooled to its final destination, possibly on a remote machine.

This default spooling of the *stderr* output file is done to reduce the network traffic costs incurred by the submitter (owner) of a batch request which is to return its *stderr* output to a remote machine upon completion. In some cases, this behavior is not desired. If it is necessary to override this behavior, then the −re flag can be specified which says that *stderr* output produced by the request is to be *immediately* written to the final destination file, as output is generated, no matter what the networking cost.

Circumstances may not allow a given NQS implementation to support this flag, in which case it will be ignored, and the *stderr* output file will simply be spooled as it ordinarily would without this flag.

−ro

By default, all output generated by a batch request sent to the *stdout* file is temporarily spooled into a file residing in a protected directory on the machine that executes the request. When the batch request completes execution, this file is then spooled to its final destination, possibly on a remote machine.

This default spooling of the *stdout* output file is done to reduce the network traffic costs incurred by the submitter (owner) of a batch request which is to return its *stdout* output to a remote machine upon completion. In some cases, this behavior is not desired. If it is necessary to override this behavior, then the −ro flag can be specified which says that *stdout* output produced by the request is to be *immediately* written to the final destination file, as output is generated, no matter what the networking cost.

Circumstances may not allow a given NQS implementation to support this flag, in which case it will be ignored, and the *stdout* output file will simply be spooled as it ordinarily would without this flag.

−s *shell-name*

Specify the absolute path name of the shell which will be used to interpret the batch request script. This flag unconditionally overrides any *shell strategy* configured on the execution machine for selecting which shell to spawn in order to interpret the batch request script.

In the absence of this flag, the NQS system at the execution machine will use one of three (3) distinct *shell choice strategies* for the execution of the batch request. Any one of the three strategies can be configured by a system administrator for each NQS machine.

The three shell strategies are called:

fixed,
free, and
login.

These shell strategies respectively cause the configured *fixed* shell to be exec'd to interpret all batch requests, cause the user's login shell as defined in the password file to be exec'd which in turn chooses and spawns the appropriate shell for interpreting the batch request script, or cause only the user's login shell to be exec'd to interpret the script.

A shell strategy of *fixed* means that the same shell (as configured by the system administrator), will be used to execute all batch requests.

A shell strategy of *free* will run the batch request script *exactly* as would an interactive invocation of the script, and is the default NQS shell strategy.

The strategies of *fixed* and *login* exist for host systems that are short on available free processes. In these two strategies, a single shell is exec'd, and that same shell is the shell that executes all of the commands in the batch request script.

The *shell strategy* configured for a particular NQS system can be determined by the *qlimit*(1) command.

−x Export all environment variables. When a batch request is submitted, the current values of the environment variables: **HOME**, **SHELL**, **PATH**, **LOGNAME** (not all systems), **USER** (not all systems), **MAIL**, and **TZ** are saved for later recreation when the batch request is spawned, as the respective environment variables: **QSUB_HOME**, **QSUB_SHELL**, **QSUB_PATH**, **QSUB_LOGNAME**, **QSUB_USER**, **QSUB_MAIL**, and **QSUB_TZ**. Unless the **−x** flag is specified, no other environment variables will be exported from the originating host for the batch request. If the **−x** flag option is specified, then all remaining environment variables whose names do not conflict with the automatically exported variables, are also exported with the request. These additional environment variables will be recreated under the same name when the batch request is spawned.

−z Submit the batch request silently. If the request is submitted successfully, then no messages are displayed indicating this fact. Error messages will, however, always be displayed.

If the batch request is successfully submitted and the **−z** flag has not been specified, the *request-id* of the request is displayed to the user. A *request-id* is always of the form: *seqno.hostname* where *seqno* refers to the sequence number assigned to the request by NQS, and *hostname* refers to the name of originating local machine. This identifier is used throughout NQS to uniquely identify the request, no matter where it is in the network.

The following events take place in the following order when an NQS *batch* request is spawned:

The process that will become the head of the *process group* for all processes comprising the batch request is created by NQS.

Resource limits are enforced.

The real and effective group-id of the process is set to the group-id as defined in the local password file for the request owner.

The real and effective user-id of the process is set to the real user-id of the batch request owner.

The user file creation mask is set to the value that the user had on the originating machine when the batch request was first submitted.

It the user explicitly specified a shell by use of the **−s** flag (discussed above), then that user-specified shell is chosen as the shell that will be used to execute the batch request script. Otherwise, a shell is chosen based upon the *shell strategy* as configured for the local NQS system (see the earlier discussion of the **−s** flag for a description of the possible *shell strategies* that can be configured for an NQS system).

The environment variables of **HOME, SHELL, PATH, LOGNAME** (not all systems), **USER** (not all systems), and **MAIL** are set from the user's password file entry, as though the user had logged directly into the execution machine.

The environment string: **ENVIRONMENT=BATCH** is added to the environment so that shell scripts (and the user's **.profile** (*Bourne shell*) or **.cshrc** and **.login** (*C-shell*) scripts), can test for batch request execution when appropriate, and not (for example) perform any setting of terminal characteristics, since a batch request is not connected to an input terminal.

The environment variables of **QSUB_WORKDIR, QSUB_HOST, QSUB_REQNAME,** and **QSUB_REQID** are added to the environment. These environment variables equate to the obvious respective strings of the working directory at the time that the request was submitted, the name of the originating host, the name of the request, and the request *request-id*.

All of the remaining environment variables saved for recreation when the batch request is spawned are added at this point to the environment. When a batch request is initially submitted, the current values of the environment variables: **HOME, SHELL, PATH, LOGNAME** (not all systems), **USER** (not all systems), **MAIL,** and **TZ** are saved for later recreation when the batch request is spawned. When recreated however, these variables are added to the environment under the respective names: **QSUB_HOME, QSUB_SHELL, QSUB_PATH, QSUB_LOGNAME, QSUB_USER, QSUB_MAIL,** and **QSUB_TZ,** to avoid the obvious conflict with the local version of these environment variables. Additionally, all environment variables exported from the originating host by the **−x** option are added to the environment at this time.

The current working directory is then set to the user's home directory on the execution machine, and the chosen shell is exec'd to execute the batch request script with the environment as constructed in the steps outlined above.

In all cases, the chosen shell is exec'd as though it were the *login* shell. If the *Bourne* shell is chosen to execute the script, then the **.profile** file is read. If the *C-shell* is chosen, then the **.cshrc** and **.login** scripts are read.

If the user did not specify a specific shell for the batch request, then NQS chooses which shell should be used to execute the shell script, based on the *shell strategy* as configured by the system administrator (see the earlier discussion of the **−s** flag).

In such a case, a *free* shell strategy instructs NQS to execute the login shell for the user (as configured in the password file). The login shell is in turn instructed to examine the shell script file, and fork another shell *of the appropriate type* to interpret the shell script, behaving *exactly* as an interactive invocation of the script.

Otherwise no additional shell is spawned, and the chosen *fixed* or *login* shell sequentially executes the commands contained in the shell script file until completion of the batch request.

QUEUE TYPES

NQS supports four different queue types that serve to provide four very different functions. These four queue types are known as *batch*, *device*, *pipe*, and *network*.

The queue type of *batch* can only be used to execute NQS *batch requests*. Only NQS *batch requests* created by the *qsub*(1) command can be placed in a *batch queue*.

The queue type of *device* can only be used to execute NQS *device requests*. Only NQS *device requests* created by the *qpr*(1) command can be placed in a *device queue*.

Queues of type *pipe* are used to send NQS requests to other *pipe* queues, or to request destination queues of type *batch* or *device*, as appropriate for the request type. In general, *pipe queues*, in combination with *network queues*, act as the mechanism that NQS uses to transport both *batch* and *device* requests to distant queues on other remote machines. It is also perfectly legal for a *pipe queue* to transport requests to queues on the *same* machine.

When a *pipe queue* is defined, it is given a *destination set* which defines the set of possible destination queues for requests entered in that *pipe queue*. In this manner, it is possible for a *batch* or *device* request to pass through many pipe queues on its way to its ultimate destination, which must eventually be a queue of type *batch* or *device* (matching the request type).

Each *pipe queue* has an associated *server*. For each request handled by a *pipe queue*, the associated server is spawned which must select a queue destination for the request being handled, based on the characteristics of the request, and upon the characteristics of each queue in the *destination set* defined for the pipe queue.

Since a different server can be configured for each pipe queue, and *batch* and *device* queues can be endowed with the *pipeonly* attribute that will only admit requests queued via another *pipe queue*, it is possible for respective NQS installations to use *pipe queues* as a *request class* mechanism, placing requests that ask for different resource allocations in different queues, each of which can have different associated limits and priorities.

It is also completely possible for a *pipe queue server*, when handling a request, to discover that no *destination queue* will accept the request, for various reasons which can include insufficient resource limits to execute the request, or a lack of a corresponding account or privilege for queueing at a remote queue. In such circumstances, the request will be deleted, and the user will be notified by mail (see *mail*(1)).

The queue type of *network*, as alluded to earlier, is implicitly used by *pipe* queues to pass NQS requests between machines, and is also used to handle queued file transfer operations.

QUEUE ACCESS

NQS supports queue access restrictions. For each queue of queue type other than *network*, access may be either *unrestricted* or *restricted*. If access is *unrestricted*, any request may enter the queue. If access is *restricted*, a request can only enter the queue if the requester or the requester's login group has been given access to that queue (see *qmgr*(1M)). Requests submitted by root are an exception; they are always queued, even if root has not explicitly been given access.

Use *qstat*(1) to determine who has access to a particular queue.

LIMITS

NQS supports many batch request resource limit types that can be applied to an NQS batch request. The existence of configurable resource limits allows an NQS user to set resource limits within which his or her request must execute. In many instances, smaller limit values can result in a more favorable scheduling policy for a batch request.

The syntax used to specify a *limit-value* for one of the *limit-flags* (*−llimit-letter-type*), is quite flexible, and describes values for two general limit categories. These two general categories respectively deal with time related limits, and those limits are not time related.

For *finite* CPU time limits, the *limit-value* is expressed in the reasonably obvious format:

[[hours :] minutes :] seconds [.milliseconds]

Whitespace can appear anywhere between the principal tokens, with the exception that no whitespace can appear around the decimal point.

Example time *limit-values are:*

1234 : 58 : 21.29	– 1234 hrs 58 mins 21.290 secs
12345	– 12345 seconds
121.1	– 121.100 seconds
59:01	– 59 minutes and 1 second

For all other *finite* limits (with the exclusion of the *nice limit-value* –l*n*), the acceptable syntax is:

.fraction [units]

or

integer [.fraction] [units]

where the *integer* and *fraction* tokens represent strings of up to eight decimal digits, denoting the obvious values. In both cases, the *units* of allocation may also be specified as one of the case insensitive strings:

b	– bytes
w	– words
kb	– kilobytes (2^10 bytes)
kw	– kilowords (2^10 words)
mb	– megabytes (2^20 bytes)
mw	– megawords (2^20 words)
gb	– gigabytes (2^30 bytes)
gw	– gigawords (2^30 words)

In the absence of any *units* specification, the units of *bytes* are assumed.

For all limit types with the exception of the *nice limit-value* (–l*n*), it is possible to state that no limit should be applied. This is done by specifying a *limit-value* of "unlimited", or any initial substring thereof. Whenever an *infinite limit-value* is specified for a particular resource type, then the batch request operates as though no explicit limits have been placed upon the corresponding resource, other than by the limitations of the physical hardware involved.

The complications caused by *batch request* resource limits first show up when queueing a *batch request* in a *batch queue*. This operation is described in the following paragraphs.

If a batch request specifies a limit that cannot be enforced by the underlying UNIX implementation, then the limit is simply ignored, and the batch request will operate as though there were no limit (other than the obvious physical maximums), placed upon that resource type. (See the *qlimit*(1) command to find out what limits are supported by a given machine.)

For each remaining *finite* limit that can be supported by the underlying UNIX implementation that is **not** a CPU *time-limit* or UNIX *execution-time nice-value-limit*, the *limit-value* is internally converted to the units of *bytes* or *words*, whichever is more appropriate for the underlying machine architecture.

As an example, a *per-process memory size limit value* of 321 megabytes would be interpreted as 321 x 2^20 bytes, provided that the underlying machine architecture was capable of directly addressing single bytes. Thus the original limit *coefficient* of 321 would become 321 x 2^20. On a machine that was only capable of addressing words, the appropriate conversion of 321 x 2^20 *bytes / #of-bytes-per-word* would be performed.

If the result of such a conversion would cause overflow when the coefficient was represented as a *signed-long integer* on the supporting hardware, then the coefficient is replaced with the coefficient of: of 2^N-1 where N is equal to the number of bits of precision in a signed long integer. For typical 32-bit machines, this *default extreme limit* would therefore be $2^{31}-1$ bytes. For word addressable machines in the supercomputer class supporting 64-bit long integers, the *default extreme limit* would be $2^{63}-1$ words.

Lastly, some implementations of UNIX reserve coefficients of the form: 2^N-1 as synonymous with infinity, meaning no limit is to be applied. For such UNIX implementations, NQS further decrements the *default extreme limit* so as not to imply infinity.

The identical internal conversion process as described in the preceding paragraphs is also performed for each *finite limit-value* configured for a particular batch queue using the *qmgr*(1M) program.

After all of the applicable *limit-values* have been converted as described above, each such resulting *limit-value* is then compared against the corresponding *limit-value* as configured for the destination batch queue. If, for every type of limit, the batch queue *limit-value* is *greater than* or *equal to* the corresponding batch request *limit-value*, then the request can be successfully queued, provided that no other anomalous conditions occur. For request *infinity limit-values*, the corresponding queue *limit-value* must also be configured as infinity.

These resource limit checks are performed irrespective of the batch request arrival mechanism, either by a direct use of the *qsub(1)* command, or by the indirect placement of a batch request into a batch queue via a *pipe* queue. It is impossible for a batch request to be queued in an NQS batch queue if *any* of these resource limit checks fail.

Finally, if a request fails to specify a *limit-value* for a resource limit type that is supported on the execution machine, then the corresponding *limit-value* configured for the destination queue becomes the *limit-value* for the unspecified request limit.

Upon the successful queueing of a request in a batch queue, the set of limits under which the request will execute is frozen, and will not be modified by subsequent *qmgr*(1M) commands that alter the limits of the containing batch queue.

CAVEATS

When an NQS batch request is spawned, a new *process-group* is established such that all processes of the request exist in the same *process-group*. If the *qdel*(1) command is used to send a signal to an NQS batch request, the signal is sent to all processes of the request in the created *process-group*. However, should one or more processes of the request choose to successfully execute a *setpgrp*(2) system call, then such processes will not receive any signals sent by the *qdel*(1) command. This can lead to "rogue" requests whose constituent processes must be killed by other means such as the *kill*(1) command. However, NQS takes advantage of any UNIX implementations that provide a mechanism of "locking" a process, and all of its subsequent children in a particular *process-group*. For such UNIX implementations, this problem does not occur.

It is extremely wise for all processes of an NQS request to catch any SIGTERM signals. By default, the receipt of a SIGTERM signal causes the receiving process to die. NQS sends a SIGTERM signal to all processes in the established *process-group* for a batch request as a notification that the request should be prepared to be killed, either because of an *abort queue* command issued by an operator using the *qmgr*(1M) program, or because it is necessary to shutdown NQS and all running requests as part of a general shutdown procedure of the local host.

It must be understood that the spawned *shell* ignores SIGTERM signals. If the current immediate child of the *shell* does not ignore or catch SIGTERM signals, then it will be killed by the receipt of such, and the shell will go on to execute the next command from the script (if there is one). In any case, the shell will not be killed by the SIGTERM signal, though the executing command will have been killed.

After receiving a SIGTERM signal delivered from NQS, a process of a batch request typically has sixty seconds to get its "house in order" before receiving a SIGKILL signal (though the sixty second duration can be changed by the operator).

All batch requests terminated because of an operator *NQS shutdown request* that did not specify the **−nr** flag are considered restartable by NQS, and are requeued (provided that the batch request shell process is still present at the time of the SIGKILL signal broadcast as discussed above), so that when NQS is rebooted, such batch requests will be respawned to continue execution. It is however, up to the user to make the request restartable by the appropriate programming techniques. NQS simply spawns the request again as though it were being spawned for the first time.

Upon completion of a batch request, a mail message can be sent to the submitter (see the discussion of the **−me** flag above). In many instances, the completion code of the spawned *Bourne* or *C-Shell* will be displayed. This is merely the value returned by the shell through the *exit*(2) system call.

Lastly, there is no good way to echo commands executed by unmodified versions of the *Bourne* and *C* shells. While the *C-shell* can be spawned in such a fashion as to echo the commands it executes, it is often very difficult to tell an echoed command from genuine output produced by the batch request, because no "magic" character such as a '$' is displayed in front of the echoed command. The *Bourne* shell does not support any echo option whatsoever.

Thus, one of the better ways to write the shell script for a batch request is to place appropriate lines in the shell script of the form:

```
echo "explanatory-message"
```

where the echoed message should be a meaningful message chosen by the user.

LIMITATIONS AND IMPLEMENTATION NOTES

Network queues have not yet been implemented.

In the present implementation, it is **not** possible to see the *stderr* or *stdout* files produced by the batch request while the request is **running**, unless the **-re** and **-ro** flags have been respectively specified.

Lastly, the strange "**@$**" syntax used to introduce *embedded argument* flags was chosen because it rarely conflicts with anything else present in a shell script file. NQS users with better minds will (rightly) suggest improved alternatives to this convention.

SEE ALSO

mail(1), qdel(1), qdev(1), qlimit(1), qpr(1), qstat(1),
kill(2), setpgrp(2), signal(2) in the *NPSN UNIX System Programmer Reference Manual.*
qmgr(1M) in the *NPSN UNIX System Administrator Reference Manual.*

NPSN HISTORY

Origin: Sterling Software Incorporated

August 1985 – Brent Kingsbury, Sterling Software
Original release.

May 1986

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-177433		

4. Title and Subtitle	5. Report Date
The Network Queueing System	December 1986
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Brent K. Kingsbury	

9. Performing Organization Name and Address	10. Work Unit No.
Sterling Software Incorporated 1121 San Antonio Road Palo Alto, CA 94303	T-3472
	11. Contract or Grant No. NAS2-11786
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
National Aeronautics and Space Administration Washington, DC 20546	14. Sponsoring Agency Code 536-01-11

15. Supplementary Notes

Point of contact: Karl Rowley, M/S 258-5, NASA Ames Research Center
Moffett Field, CA 94035 (415) 694-4417, FTS 464-4417

16. Abstract

This paper describes the implementation of a networked, UNIX based queueing system developed for a government contract with the National Aeronautics and Space Administration (NASA). The system discussed supports both batch and device requests, and provides the facilities of remote queueing, request routing, remote status, queue access controls, batch request resource quota limits, and remote output return.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Batch queueing software, System software	Unclassified-Unlimited Subject catergory 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	64	

*For sale by the National Technical Information Service, Springfield, Virginia 22161