|  |
| --- |
| NCBI |
| NetSchedule Server |
| High-Level Description |
|  |
| **Sergey Satskiy** |
| **2/8/2012** |

|  |
| --- |
| Document version: 1.33 |

Changes history

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | What changes |
| 1.0 | Feb 8, 2012 | Sergey Satskiy | Initial revision |
| 1.1 | Mar 8, 2012 | Sergey Satskiy | Updated after initial review |
| 1.2 | Mar 14, 2012 | David McElhany | Reviewed |
| 1.3 | Mar 23, 2012 | Sergey Satskiy | markdel\_batch\_size settings parameter added |
| 1.4 | May 21, 2012 | Sergey Satskiy | exclusive new affinity flag description; clearing worker node preferred affinities; wnode\_timeout parameter description added |
| 1.5 | May 30, 2012 | Sergey Satskiy | pending\_timeout parameter description added |
| 1.6 | June 13, 2012 | Sergey Satskiy | max\_pending\_wait\_timeout parameter described |
| 1.7 | August 23, 2012 | Sergey Satskiy | NS 4.14.0 related changes in the configuration parameters |
| 1.8 | November 30, 2012 | Sergey Satskiy | Job state diagram updated for NS 4.16.1 and up |
| 1.9 | December 14, 2012 | Sergey Satskiy | Adding ‘notif\_handicap’ queue parameter description for NS 4.16.3 |
| 1.10 | April 22, 2013 | Sergey Satskiy | New default value for run\_timeout\_precision for NS 4.16.8 |
| 1.11 | September 11, 2013 | Sergey Satskiy | Added description of the scramble\_job\_keys queue parameter for NS 4.16.10 |
| 1.12 | November 25, 2013 | Sergey Satskiy | Added stat\_interval server parameter for NS 4.16.11 |
| 1.13 | December 31, 2013 | Sergey Satskiy | netcache\_api\_section obsolete in NS 4.17.0  Added linked\_section\_PPP queue parameters for NS 4.17.0  Added service\_to\_queue section description for NS 4.17.0 |
| 1.14 | January 8, 2014 | Sergey Satskiy | Fix: various timeout became floating point values |
| 1.15 | March 21, 2014 | Sergey Satskiy | Adding queue pause/resume feature description for NS 4.17.0 |
| 1.16 | March 31, 2014 | Sergey Satskiy | Adding [server]/max\_client\_data parameter for NS 4.17.0 |
| 1.17 | April 15, 2014 | Sergey Satskiy | Adding description of the recently added features: transient client data, service to queue, queue linked sections. |
| 1.18 | May 27, 2014 | Sergey Satskiy | Adding read\_timeout for a queue. Changing run\_timeout description. The changes are for NS 4.17.2 |
| 1.19 | May 30, 2014 | Sergey Satskiy | Updating the transition diagram |
| 1.20 | June 3, 2014 | Sergey Satskiy | Updating the notifications description – NS 4.17.2 supports READ notifications |
| 1.21 | June 12, 2014 | Sergey Satskiy | Updating the transition diagram for NS 4.19.0 |
| 1.22 | August 4, 2014 | Sergey Satskiy | Adding the client registry garbage collector parameters description for NS 4.20.0 |
| 1.23 | August 7, 2014 | Sergey Satskiy | Adding read\_failed\_retries parameter for NS 4.20.0  Adding read\_blacklist\_time parameter for NS 4.20.0 |
| 1.24 | September 10, 2014 | Sergey Satskiy | Adding reader\_timeout parameter for NS 4.20.0 |
| 1.25 | September 15, 2014 | Sergey Satskiy | Adding reader\_host parameter for NS 4.20.0 |
| 1.26 | September 16, 2014 | Sergey Satskiy | Marking run\_time\_precision parameter obsolete for NS 4.20.0 |
| 1.27 | September 16, 2014 | Sergey Satskiy | Introducing max\_pending\_read\_wait\_timeout parameter for NS 4.20.0 |
| 1.28 | November 24, 2014 | Sergey Satskiy | Introducing [error\_simulator] section for debug purposes in debug mode |
| 1.29 | November 26, 2014 | Sergey Satskiy | Introducing [error\_simulator]/ reply\_with\_garbage and [error\_simulator]/ garbage\_data parameters |
| 1.30 | August 4, 2015 | Sergey Satskiy | Obsolete configuration file parameter ‘force\_storage\_version’ for NS 4.23.0 |
| 1.31 | August 17, 2015 | Sergey Satskiy | Introducing [server]/reserve\_dump\_space parameter for NS 4.23.0 |
| 1.32 | February 29, 2016 | Sergey Satskiy | New group and scope registries garbage collector settings for NS 4.25.0 |
| 1.33 | May 16, 2016 | Sergey Satskiy | Adding job scopes description; adding affinity prioritization flag description. |

Table of Contents

[NetSchedule Server 5](#_Toc451183492)

[Overview 5](#_Toc451183493)

[Queues 6](#_Toc451183494)

[Communication Protocol 6](#_Toc451183495)

[Files Architecture 7](#_Toc451183496)

[Basic Scenario 7](#_Toc451183497)

[Complete Job State Diagram 8](#_Toc451183498)

[Affinities 11](#_Toc451183499)

[Notifications 13](#_Toc451183500)

[Job State Changes 13](#_Toc451183501)

[Job Availability 13](#_Toc451183502)

[Job Groups 14](#_Toc451183503)

[Blacklists 15](#_Toc451183504)

[Clients 15](#_Toc451183505)

[Garbage Collection 16](#_Toc451183506)

[Job Security Token 16](#_Toc451183507)

[Queue Pausing and Resuming 17](#_Toc451183508)

[Service to Queue 17](#_Toc451183509)

[Arbitrary Queue Properties 17](#_Toc451183510)

[Job Scopes 18](#_Toc451183511)

[Monitoring and Maintenance 18](#_Toc451183512)

[Alerts 18](#_Toc451183513)

[GRID Dashboard 18](#_Toc451183514)

[Commands 19](#_Toc451183515)

[AppLog 20](#_Toc451183516)

[grid\_cli Utility 21](#_Toc451183517)

[Python Module 21](#_Toc451183518)

[Command Line Arguments 21](#_Toc451183519)

[Configuration Parameters 21](#_Toc451183520)

[[server] section 22](#_Toc451183521)

[[log] section 25](#_Toc451183522)

[[bdb] section 26](#_Toc451183523)

[[service\_to\_queue] section 26](#_Toc451183524)

[[qclass\_YYY] section 27](#_Toc451183525)

[[queue\_ZZZ] section 27](#_Toc451183526)

[[error\_emulator] section 31](#_Toc451183527)

[Appendix A. Response Depending on Security Token 0](#_Toc451183528)

# NetSchedule Server

This document provides an overview of the NetSchedule server version 4.10.0 and up. The older versions may not support some of the mentioned features.

# Overview

NetSchedule server is a distributed job execution dispatcher.

The diagram below shows the main actors and entities involved in a typical NetSchedule application.



The NetSchedule server is running on a Linux host and holds queues of jobs. The queues are identified by their names and there can be any number of queues configured on a single instance of the NetSchedule server.

Submitters can submit jobs to NetSchedule and check their state periodically or receive notifications about changes of the job states. When a job is submitted its input has to be supplied and optionally a job affinity (see the [Affinity](#Affinity) section below for a detailed discussion) can be provided. NetSchedule does not have any restrictions on what data can be supplied as a job input (except for the size) and it is the submitters’ and worker nodes’ responsibility to interpret the input in the appropriate way. There can be as many submitters as required.

Worker nodes are those who do the calculations required by a certain job and provide the results of the calculations back. Worker nodes request NetSchedule server if a job is available in a queue and if so grab it for execution. When the calculation is finished the job return code and the output are returned to the server and another job can be requested. There can be as many worker nodes as required.

A certain job may have a huge output which does not fit what NetSchedule can accept. Worker nodes in this case can store the output outside of NetSchedule e.g. in NetCache and provide a reference to the external storage in the job output instead of the real output. Readers of such jobs may request from NetSchedule a job for reading and then notify NetSchedule when reading from an external storage is finished. As with submitters and worker nodes, any number of readers can exist on a network.

The last actor is an administrator. This role involves operations related to the whole server like collecting statistics, checking and possibly changing the server configuration, monitoring jobs etc. There can be as many administrators as necessary.

One thing that should be mentioned here is that there is no limitation on the roles played by a single executable module. For example an application can be a submitter and a reader together with issuing some administrative commands from time to time. The role separation is done for structuring purposes and for the possibility to introduce different permissions for clients depending on their roles.

# Queues

All the jobs are stored in queues inside NetSchedule. So one of the first things a client must do is to identify which queue it is going to work with. The queues are identified by their names and the names must be unique within one NetSchedule instance. Usually a separate queue is created for jobs of a certain type and this helps to interpret the job’s input and output in the appropriate way for the submitters, worker nodes and readers, based on the queue name.

NetSchedule supports two types of queues: static and dynamic.

Static queues come from a configuration file and they cannot be deleted using NetSchedule commands without reconfiguring. If static queues are added or removed from the configuration file and the RECO command is issued, then the added queues are created in NetSchedule’s data structures and the removed queues are marked for deletion. Queues marked for deletion will be deleted after all their jobs have been removed. Static queues survive server restarts.

Dynamic queues can be created (see the QCRE command) on a running server for a temporary usage and then deleted (see the QDEL command). Dynamic queues do not survive server restarts.

# Communication Protocol

NetSchedule communicates with all types of client using a TCP/IP connection. The protocol is based on human readable strings exchange so even a simple telnet application could be used to communicate to NetSchedule.

A C/C++ API is available as well and Python support is on its way. These APIs are recommended while a direct connection to NetSchedule is targeted to administrators and very experienced users.

See also the NetSchedule Commands Reference (<https://sp.ncbi.nlm.nih.gov/IEB/ISS/Shared%20Documents/Forms/AllItems.aspx>).

# Files Architecture

The diagram below shows the files used by NetSchedule server.



NetSchedule reads its configuration file (usually named netscheduled.ini) and creates the queues described in it. The jobs in the queues are also backed up in a database so if there is a server restart then the saved jobs are restored from the database.

NetSchedule logs every single command (as well as some other internal events) in a log file which is then available for analysis using the AppLog application.

# Basic Scenario

Basically what NetSchedule does is it keeps track of each submitted job. From the NetSchedule point of view a job simply changes its state. The diagram below shows a very basic straight forward state diagram for a submitted job.



The job life cycle starts when a submitter submits a new job to NetSchedule (see the SUBMIT command). In response to the submit NetSchedule creates a new job, generates a unique string identifier for it and moves the job to the Pending state. Then a worker node comes to NetSchedule and requests a job for execution (see GET2 command). In response to the request NetSchedule picks a job and moves it to the Running state. When the worker node finishes all the required calculations it notifies NetSchedule that the job is done and provides the job output as well as its return code (see the PUT2 command). In response to this NetSchedule moves the job to the Done state. Then a reader comes and asks for a job for reading (see the READ command). In response to this NetSchedule picks a job and moves it to the Reading state. When reading is finished the reader informs NetSchedule (see the CFRM command). In response NetSchedule moves the job to the Confirmed state.

After a certain configurable time the job will be deleted from all the NetSchedule data structures including the backup database.

# Complete Job State Diagram

The scenario above does not consider some real life situations like timeouts and failures. The actual NetSchedule job state diagram is given below. The commands received from clients are given in capital letters while internal events and explanations are in small letters.



A single job can be submitted (see the SUBMIT command) or a batch of jobs can be submitted (see the BSUB command). Regardless of how a job is submitted NetSchedule moves it to the Pending state. The Pending state means that a job is available for execution and can be given to a worker node for execution.

In a typical scenario a worker node requests a job using the GET2 command or exchanges a finished job for another available job using the JXCG2 command. The result of this operation is that the job is moved from the Pending state to the Running state.

While executing a job, a worker node may decide that it is not in a mood to complete this certain job, e.g. because a database resource is not available at the moment. To return the job back to NetSchedule the worker node can use the RETURN2 command and the result of this operation is that the job gets back to the Pending state. The job run counter is not increased for the RETURN2 command.

There are four indications of a failed job from the NetSchedule point of view:

* A worker node explicitly reports that the job execution failed using the FPUT2 command.
* A worker node does not report anything about the job within a timeout.
* A worker node decided to shut itself down for some reason and reports this using the CLRN command. That means that all the data associated with the worker node should be cleaned, including the jobs which were executed by this worker node.
* A worker node connects to NetSchedule using the same client identifier that was used when the job was given to the worker node, but using a different session identifier. This most probably means that the worker node has been restarted and thus all the data associated with the worker node should be cleaned, including the jobs which were executed by this worker node.

If NetSchedule has detected a failed job then the job run counter is checked. If it exceeds the configured value (per queue, configuration file) then the job is moved to the Failed state. If not then the job is moved to the Pending state.

When a worker node successfully finishes a job execution it submits the results to NetSchedule (see the PUT2 command) or exchanges the job for another available one (see the JXCG2 command). In response to these commands NetSchedule moves the job from the Running state to the Done state.

It may also happen that a job was given to a worker node and the worker node did not report the job completion within a timeout. If the job has not exceeded the run tries counter then it will be moved to the Pending state. If the worker node subsequently reports the job is done, then NetSchedule will accept the job execution results (see the PUT2 and JXCG2 commands) and will move the job to the Done state, even though the job was in the Pending state.

When a job is in the Done or Failed or Canceled state a reader may request a job for reading (see the READ command). If an available job is found then it is moved to the Reading state. If a canceled job was already given for reading before then it will not be provided to the reader the second time.

While reading a job a reader might decide that it’s better if another reader reads this job. In such a case the reader can return the job back to NetSchedule using the RDRB command. The result of this operation is that the job is moved back to the Done state and the jobs reader count is not changed.

There are four indications that reading has failed from the NetSchedule point of view:

* A reader explicitly reports that the job reading failed using the FRED command.
* A reader does not report anything about the job within a timeout.
* A reader decides to shut itself down by for some reason and reports this using the CLRN command. That means that all the data associated with the reader should be cleaned, including the jobs that were read by this reader.
* A reader connects to NetSchedule using the same client identifier as was used when a job was given to the reader, but using a different session identifier. This most probably means that the reader has been restarted and thus all the data associated with the reader should be cleaned, including the jobs which were read by this reader.

If job reading failed for any of these reasons then the job read counter is checked. If it exceeds the configured value (per queue, configuration file) then the job is moved to the ReadFailed state. If not the job is moved to the state preceding the Reading state.

When a reader successfully finishes reading the job it informs NetSchedule using the CFRM command. In response to this NetSchedule moves the job to the Confirmed state.

It may also happen that a job was given to a reader and the reader not did not report the reading completion within a timeout. If the job has not exceeded the read tries counter then it will be moved to the Done state. If the reader subsequently reports the job reading is done, then NetSchedule will accept the job reading (see the CFRM command) and will move the job to the Confirmed state, even though the job was in the Done state.

Jobs are moved from any state to the Canceled state when the CANCEL command is received, and they remain there until deleted. After a certain configurable time, jobs in the Canceled state will be deleted from all NetSchedule data structures including the backup database.

Jobs may remain in the Pending, Done, Failed, ReadFailed, or Confirmed states indefinitely. They will be restored to their state after a server restart.

If NetSchedule detects that an invalid transition is requested it will report an error or a warning depending on a situation.

# Affinities

When a job is submitted to NetSchedule it can be attributed with an affinity. A job affinity is a string identifier of an arbitrary length and the allowed symbols are [a-z][A-Z][0-9] and underscore, e.g. the identifier MyAffinity\_001 is a valid affinity identifier.

A job may have zero or one affinity. Many jobs may have the same affinity.

A worker node in turn can inform NetSchedule about its preferred affinities using the CHAFF command. The CHAFF command supports two lists of affinity identifiers – a list of affinities to be added and a list of affinities to be removed from the preferred affinities list. It is a worker node responsibility to keep NetSchedule informed about the preferred affinities in a timely and correct manner. NetSchedule automatically cleans the preferred affinities list for a worker node in the following cases:

* A worker node explicitly reports its restart using the CLRN command.
* A worker node connects to NetSchedule using the same client identifier that was used earlier, but uses a different session identifier. This most probably means that the worker node has been restarted and thus all the data associated with the worker node should be cleaned, including the list of preferred affinities.
* A worker node does not expose any activities within a configured timeout (default: 40 sec). This most probably means that the worker node died or that there is a significant network connectivity error. So, to avoid a situation that a job with certain affinity is not given to another worker node just because a dead worker node has this affinity in its preferred affinities list, the server resets the preferred affinities when inactivity is detected. Note 1: the running and reading jobs are not reset. Note 2: this case is introduced in NetSchedule 4.11.0.

Later on, a worker node can be specific about a job it wants to get for execution (see the GET2 and JXCG2 commands). The worker node can specify the following parameters in its job request:

* Explicit list of affinities. If there is a job in a Pending state and that job was submitted with an affinity which matches one of the provided, then the job will be given for execution.  
  The explicit list of affinities is an optional parameter and it has the first priority in the picking job procedure.  
  If there are many affinities in the explicit list of affinities they are treated equally by default. However a flag could be provided to treat the explicit list of affinities as an ordered one in accordance with the affinity priorities: the most prioritized affinity comes first.
* A flag to consider the worker node preferred affinities or not. If the flag is set to true then NetSchedule checks if there are any jobs in the Pending state which were submitted with an affinity which matches one of those in the preferred list. If such a job is found then it will be given for execution.  
  The flag is a mandatory parameter and it has the second priority in the picking job procedure.
* A flag which tells if any affinity suits the worker node. If the flag is set and the first and second priority criteria did not match any jobs, then any job in the Pending state will be given for execution.  
  The flag is a mandatory parameter and it has the third priority in the picking job procedure.
* A flag which tells that a job without any affinity or with an affinity which is not in preferred lists of any known worker nodes suits the worker node. If a job with an affinity is picked when this flag is set then this affinity is added to the preferred list automatically.  
  The flag is mutually exclusive with the any affinity flag. The flag is optional, has the fourth priority and was introduced in NetSchedule 4.11.0.

There is also a queue settings parameter called max\_pending\_wait\_timeout which may alter the algorithm of picking a job for a worker node. If the parameter is set to a positive value then it affects the cases when a worker node asks for a job with considering preferred affinities and is willing to accept exclusive new affinities. In such cases a first candidate job is picked as described above. Then a second candidate is searched among those jobs which are in a pending state longer than the configured timeout. The first candidate wins if the job exceeds the pending timeout or if there is no second candidate. This feature targets the cases when a worker node died but its preferred affinities are still registered for it which prevents other worker nodes to pick vacant jobs.

The last thing that must be mentioned is that NetSchedule has some limitations on how many affinities can coexist. There is a configuration parameter which specifies the max number of affinities per queue. A command may lead to exceeding this limit, e.g. submitting a new job with an affinity not used before may lead to overflowing the affinity registry. Such commands will fail with a corresponding error messages. NetSchedule also supports garbage collecting for unused affinities.

# Notifications

NetSchedule supports two kinds of notifications:

* Job state change notifications
* Job availability notifications

## Job State Changes

When a submitter submits a job (see the SUBMIT command) it can provide two optional parameters: a timeout within which the submitter wants to receive notification about the submitted job state changes and a UDP port on which those notifications are expected.

If those parameters are supplied then NetSchedule sends a single UDP packet, i.e. not guaranteed to be delivered, when a job reaches the Done, Failed or Canceled states.

## Job Availability

When a worker node requests a job for execution (see the GET2, JXCG2 commands) it can provide two additional parameters: a timeout and a port number. These parameters are used only if NetSchedule does not have at the moment an available job which matches the request criteria. In this case NetSchedule memorizes the request parameters and when a job becomes available for execution the criteria are checked again. Should the job match, NetSchedule starts sending notifications to the given UDP port.

A similar functionality is implemented for the READ command. The command also supports a timeout and a port number which are handled similar how it is done for the GET2 command.

Bearing in mind that the UDP delivery is not guaranteed NetSchedule sends many UDP packets. The diagram below explains how it is done.



The zero time moment is when a job was found for the worker node. Starting from this point Netschedule sends UDP packets within the high frequency period (configurable) with the high frequency interval (configurable). After the high frequency interval NetSchedule sends packets with the low frequency interval but each time two of the same packets are sent. The slowdown rate is also configurable.

Similar policy of sending notification packets is implemented for the READ notifications. The same configuration parameters as for GET2 notifications are used to tune the packets timeouts.

The GET2 notifications are stopped when one of two things happens:

* The worker node requests a job.
* The timeout provided in the initial GET2 or JXCG2 is over.

The READ notifications are similarly stopped when one of two things happens:

* The reader requests a job.
* The timeout provided in the initial READ command is over.

NetSchedule can provide a list of notifications – both currently active and those which will be send when a condition is met (see the STAT NOTIFICATIONS command).

# Job Groups

Sometimes it is convenient to refer to a group of jobs as to a single entity. NetSchedule support job grouping via a user supplied group identifier. A group identifier is a string identifier of an arbitrary length and the allowed symbols are [a-z][A-Z][0-9] and underscore, e.g. the identifier MyGroup\_001 is a valid group identifier.

When a job is submitted as a single one (see the SUBMIT command) or many jobs are submitted as a batch (see the BSUB command) the user can provide an optional parameter, group=<GroupID>. The job(s) will be included into the given group. A group is created implicitly if it did not exist before. NetSchedule destroys a group when no more jobs reference it (see the Garbage Collection section for details).

Having groups of jobs the following functionality is supported:

* NetSchedule can provide a list of job groups it is aware of at the moment (see the STAT GROUPS command). Optionally, individual job keys within each group can also be provided.
* NetSchedule can provide the number of jobs per status within a group (see the STAT JOBS group=<GroupID> command).
* NetSchedule can dump jobs within a group (see the DUMP group=<GroupID> command).
* NetSchedule can cancel all the jobs within a group (see the CANCEL group=<GroupID> command).
* NetSchedule can provide a job for reading restricting candidates by the given group (see the READ group=<GroupID> command).

# Blacklists

It is possible that a job could not be completed by a worker node regardless of how many times it tries. An example of such a scenario could be a situation when a job requires a lot of memory to be completed while a worker node does not have enough. To avoid rescheduling a job for the same worker node in case of problems NetSchedule keeps track of blacklisted jobs for each worker node.

A job will be put into a worker node blacklist in the following cases:

* Worker node reports a job as failed.
* Worker node returns a job.
* NetSchedule detects a timeout of a job executing.

Currently the blacklisted jobs stay there forever for a certain worker node. Later versions of NetSchedule may introduce timeouts for keeping jobs in blacklists (implemented in NS 4.17.0).

NetSchedule 4.17.0 introduces a new parameter in the RETURN2 command. This parameter tells NetSchedule if a job should be added into the worker node blacklist. By default the job will be added to the worker node blacklist.

# Clients

NetSchedule distinguishes two types of clients: anonymous and identified clients. This classification applies regardless of what commands a client issues.

When a new connection to NetSchedule is opened the first string NetSchedule expects is the client description. If the client provides two optional parameters client\_node and client\_session, then the client is treated as identified. If those parameters are not provided then the client is treated as anonymous. Anonymous clients are deprecated but still supported.

NetSchedule keeps track of its clients using the clients’ registry. The clients’ registry stores some client attributes like last activity time, roles the client played, number of submitted and executed jobs, currently executing jobs etc. The records in the clients’ registry however are created only for the identified clients.

Some important functionality like

* blacklisted jobs support
* automatic rescheduling of running jobs when a worker node restarted or signaled restart

relies on information stored in the clients registry, so it will be supported only for the identified clients.

Starting from version 4.17.0 NetSchedule supports transient client data for the identified clients. Using the SETCLIENTDATA command any data can be stored for a client. The data are transient i.e. they will not survive the server restart.

# Garbage Collection

Once jobs are submitted to NetSchedule they do not stay there forever. NetSchedule implements a garbage collector to clean the data structure of those jobs which become out of interest.

The garbage collector thread becomes active regularly (configurable parameter) and scans a configurable number of jobs. If a configurable timeout since the last activity with a job is exceeded then the job is marked for deletion. This means that there is still a record about the job in the database however the clients will get no information about the job. E.g. there will be no output in the DUMP command for the jobs marked for deletion. Later on the jobs marked for deletion are deleted from the database, but no more than a configurable number are deleted at a time. These limits are introduced to avoid blocking the database for too long which can cause delays in serving the major requests like submitting jobs and providing them for executing.

This approach may lead to a situation when the speed of marking jobs for deletion is higher than the speed of actual deletions from the database. To avoid constantly increasing the amount of garbage, the configured speed of deletions should match or exceed the speed of submitting jobs.

The number of jobs that are marked for deletion but not deleted yet is displayed in response to the STAT command – see the garbage\_jobs value.

The garbage collection thread is also responsible for cleaning the affinity registry and the job groups registry.

# Job Security Token

In order to prevent errors related to improper job handling, e.g. a worker node reports that a job has failed while this job has never been given to that worker node, NetSchedule introduces job security (authorization) tokens. A security token is a string identifier which consists of two parts:

* job passport (fixed at the time the job is created)
* piece which is generated each time the job is given for execution or reading

A security token is provided to the user when a job is given for executing or for reading (see the GET2 and READ commands). Later on when a worker node or a reader reports that an operation is completed (see the PUT2, FPUT2, RETURN2, CFRM, RDRB, FRED commands) it has to provide the security token it received. NetSchedule will accept the operation results with no warnings if the security token matches. If only the job passport matched then the results will be accepted and a warning is generated. If the job passport did not match then the results will be rejected.

Appendix A describes in detail what NetSchedule will do depending on a job state and an incoming command.

# Queue Pausing and Resuming

NetSchedule 4.17.0 introduces a feature of pausing a queue. If a queue is paused (see the QPAUSE command) then when worker nodes request a new job for execution they will not get a job. Later on a queue could be instructed to resume providing the jobs as usual (see the QRESUME command).

There are two modes of pausing a queue. They are with or without pullback. The mode (if a queue is paused) is provided to the worker node when it requests a job (see GET2, STATUS2, SST2 and WST2 commands).

NetSchedule server does not make any decisions which depend on a pause mode. The mode is solely intended for a worker node. When a worker node checks a job status (WST2) it can analyze the pause node and decide what to do with the currently running jobs. Worker node can stop executing a job and return it to the server (pullback mode) or continue with the current jobs (no pullback mode).

# Service to Queue

NetSchedule server supports translation of a service name to a queue name (starting from 4.17.0). It might come handy in cases when it is better not to have a queue name configured on the client side but to have only the service name configured.

If a service name translation is configured on the server (see [service\_to\_queue] configuration file section) then the client may come to the server and issue the QINF2 command providing a service name. If translation is configured then the server will respond with the corresponding queue name and its parameters. Having the queue name at hand the client can set it as the current and continue working as if a queue name was configured on the client.

This feature is solely purposed to simplify the client configuring. Instead of two configuration items – service name and NetSchedule queue name – the clients will have only one: the service name.

# Arbitrary Queue Properties

Starting from version 4.17.0 NetSchedule supports arbitrary queue properties in a configuration file via linked sections. A queue can specify any number of linked\_section\_yyy parameters. The value of the parameter is another section name which must appear in the configuration file. All the linked section values will be provided in the QINF2 output with the ‘yyy.’ prefix.

This feature is to support client configuration. NetSchedule does not make any decisions basing on the linked sections values.

# Job Scopes

Starting from version 4.25.0 NetSchedule supports job scopes. Scopes allow to split all the jobs into non-intersecting groups.

Once submitted a job belongs to a certain scope or does not belong to any scope (by default). A scope name is provided by a connection context. A scope name may appear in a connection context using one of the two ways:

* at the handshake stage a scope name could be provided
* SETSCOPE command sets the scope name for all the consequent commands

Most of the commands respect the current scope. For example when a job is submitted it picks the scope from the connection context. When a job is requested by a worker node the current scope is respected too – the candidate jobs will be restricted by the current scope jobs only in addition to the standard job picking procedure.

# Monitoring and Maintenance

NetSchedule monitoring and maintenance can be done using a direct TCP/IP connection to the server and / or by using some other applications and utilities. This section briefly describes all these tools.

## Alerts

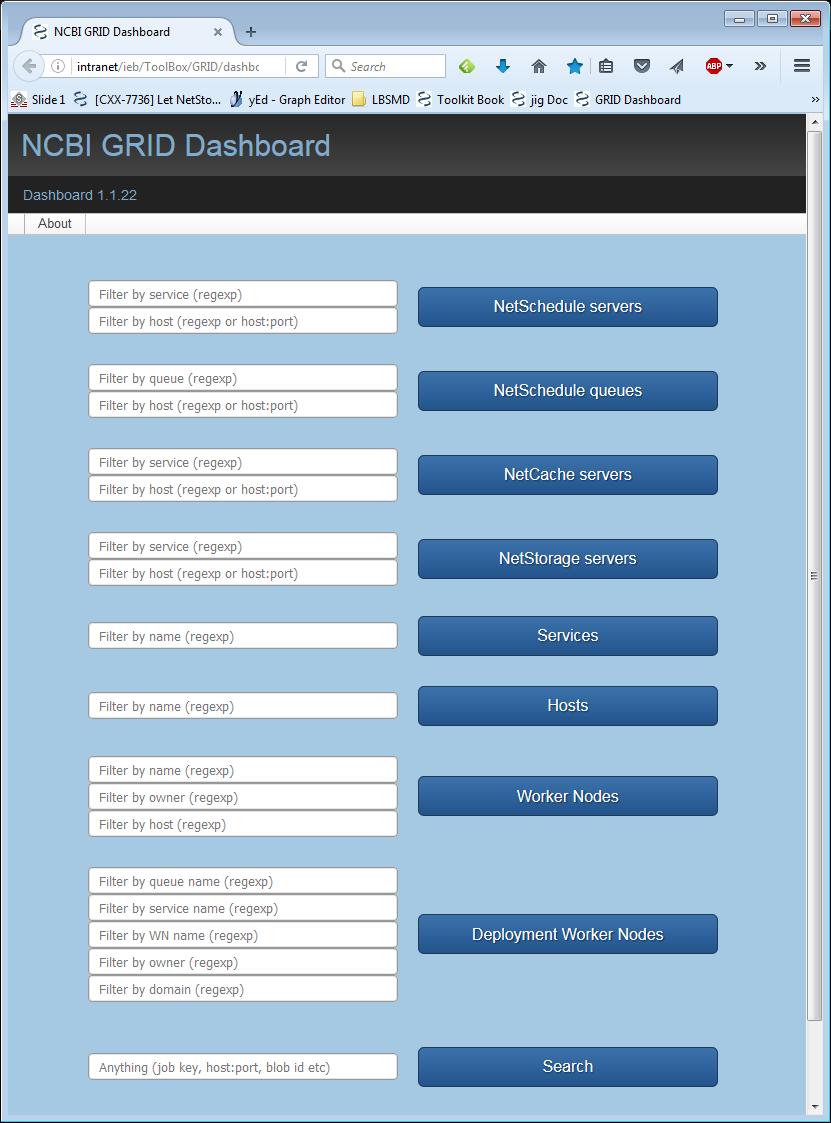
In case of certain error indicating events the server (starting from 4.17.0) can raise alerts. For example alerts will be raised if:

* a problem in a configuration file is detected at the startup time
* pid file could not be created at the startup
* the server started after crash
* etc

The alerts are not sent anywhere however they could be retrieved via the STAT ALERTS command. A brief alert information is also provided in response to the VERSION command.

## GRID Dashboard

GRID dashboard is a web application which supports NetSchedule servers in particular. The user is able to see all the current server information and in some cases even perform administrative actions. For example, a queue could be paused, an alert could be acknowledged, jobs in a queue could be cancelled. The list of supported actions is going to grow in the future. GRID dashboard is available at <http://intranet/ieb/ToolBox/GRID/dashboard/dashboard.cgi>.



## Commands

The table below describes the commands which are usually associated with monitoring and maintenance. There is no limitation on who can use these commands. Only the most important commands are described here. See the complete commands reference for all the NetSchedule commands (<https://sp.ncbi.nlm.nih.gov/IEB/ISS/Shared%20Documents/Forms/AllItems.aspx>).

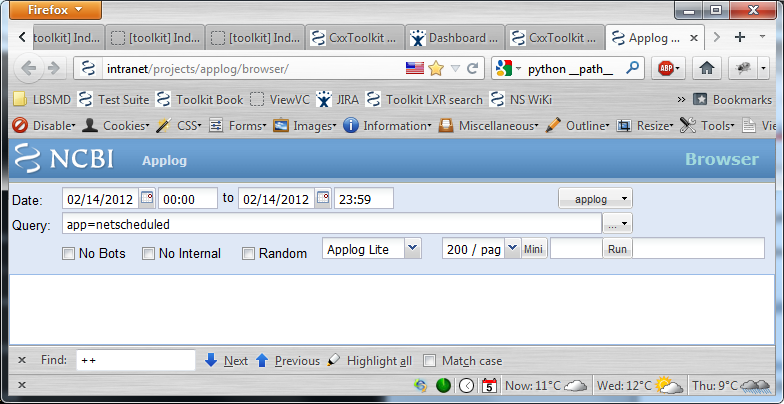
|  |  |
| --- | --- |
| Command | Description |
| GETCONF | Provides the content of the current configuration file.  Please note that the displayed values may not be the currently effective ones. This may happen if a configuration file was altered after the server started and then RECO command was given. In this case the configuration file is loaded but not all the altered values had been accepted.  The command requires administrative privileges. |
| VERSION | Provides the server version, the protocol version and the data storage version. |
| RECO | NetSchedule rereads its configuration files and changes its settings. Note that not all the settings can be altered without restarting the server. The detailed description of what parameters can be altered without restarting the server is in the Configuration Parameters section.  The command requires administrative privileges. |
| QLST | Prints the list of the queues the server has at the moment. |
| STAT | Prints the server status information including the job transition counters. Works per queue and per server. |
| STAT QUEUES | Prints all the server queues. |
| STAT QCLASSES | Prints all the queue classes. |
| STAT CLIENTS | Prints the identified clients registry. |
| STAT NOTIFICATIONS | Prints the notifications registry. |
| STAT AFFINITIES | Prints the affinities registry. |
| STAT GROUPS | Prints the job groups registry. |
| STAT JOBS | Prints the number of jobs per status. |
| STAT ALERTS | Prints the server alerts. |
| DUMP | Prints detailed information including the job events history for a single job or for many jobs. |

## AppLog

The NetSchedule logs are collected by AppLog so they could be analyzed whether from a command line or via a web interface.

The web interface can be accessed here: <http://intranet/projects/applog/browser>.

The query string should have app=netscheduled in it. It is also recommended not to have the “No Bots” and “No Internal” check boxes ticked.



The rest of the parameters and query conditions could be set as required.

The request stop status codes respect the HTTP approach, i.e. the code 200 means that everything is fine. The status codes in the range 400 – 499 means a client error. The status codes in the range 500 and up means that a server side error appeared. NetSchedule does not use status codes in the range 300 – 399.

## grid\_cli Utility

While the NetSchedule monitoring and maintenance commands could be executed using a direct telnet connection, the recommended way is to use a command line utility designed to simplify communications with both NetSchedule and NetCache servers. To see the grid\_cli utility commands type:

grid\_cli --help

## Python Module

If some monitoring and maintenance operations need to be automated it is recommended to use a python module. The module provides a native python wrapper around communications with NetSchedule via the grid\_cli utility. Please address your questions to Dmitry Kazimirov should you be interested in the python module.

# Command Line Arguments

The table below describes the server command line arguments.

|  |  |
| --- | --- |
| Argument | Description |
| -help | Prints help message and exits. |
| -reinit | Recreates the database regardless of whether it existed or not. If this argument is not given and a database exists from a previous server run then the existing database will be used. |
| -nodaemon | If given then the server does not daemonize. |
| -version | Prints the server version and exits. |
| -version-full | Prints the server version, the storage version and the protocol version and then exits. |
| -logfile | The file to which the server log should be redirected. |
| -conffile | The file from which the server should read the configuration. |

# Configuration Parameters

NetSchedule reads the configuration from a file. The default name of the server is netscheduled so (if the –conffile command line argument is not provided) the default configuration file name will be netscheduled.ini (the suffix .ini replaces the application name suffix if so).

The configuration file uses the industry standard ini file format with sections and values within sections. The sections below describe each section of the configuration file separately.

## [server] section

|  |  |
| --- | --- |
| Value | Description |
| no\_default\_queues | If not set, every queue section will create a queue along with queue class.  If set, queue section is a full equivalent of qclass  Default: false (for compatibility with older set-ups where there were only queue\_\* sections)  Dropped starting from NetSchedule 4.14.0 |
| reinit | If set to true then the database will be recreated even if it existed after a previous server run.  Default: false |
| max\_connections | Maximum number of simultaneously opened connections.  Default: 100 |
| max\_threads | Maximum number of threads for processing client requests.  Default: 25 |
| init\_threads | Initial number of threads for processing client requests.  Default: 10 |
| port | TCP/IP port on which the server expects incoming connections.  Default: 9100 |
| use\_hostname | If set to true then the job keys will have a host name instead of an IP address of the server.  It is safer to set this value to false.  Default: false |
| network\_timeout | If there is no client activity within this period of time the server will close the connection.  Default: 10 (integer, in seconds) |
| log | Top level logging flag. If set to false then the server will produce no logs at all. If set to true then the server will produce some basic logging plus more specific logging flags will be taken into account (see below).  The setting is taken into account by the RECO command.  Default: true |
| log\_batch\_each\_job | If set to true then each job in a batch submit will be logged as if it was submitted individually. If set to false then a batch submit will be logged as a single record in the log.  The setting is taken into account by the RECO command.  Default: true |
| log\_notification\_thread | If set to true then the notifications thread will produce logging.  The setting is taken into account by the RECO command.  Default: false |
| log\_cleaning\_thread | If set to true then the garbage collecting thread will produce logging.  The setting is taken into account by the RECO command.  Default: true |
| log\_execution\_watcher\_thread | If set to true then the thread which watches jobs execution and reading timeouts will produce logging.  The setting is taken into account by the RECO command.  Default: true |
| log\_statistics\_thread | If set to true then the thread which prints transition statistics periodically will produce logging.  The setting is taken into account by the RECO command.  Default: true |
| del\_batch\_size | The maximum number of jobs the garbage collector will delete from the database at once.  Default: 100 |
| markdel\_batch\_size | The maximum number of jobs the garbage collector marks for later deletion.  Default: 200 |
| scan\_batch\_size | The maximum number of jobs the garbage collector scans till the del\_batch\_size candidates for deletion are identified.  Default: 10000 |
| purge\_timeout | Timeout between two consecutive runs of the garbage collector.  Default: 0.1 (float, in seconds. Must be divisible of 0.1) |
| max\_affinities | Maximum number of entries (per queue) the server can have in the affinity registry.  Default: 10000 |
| admin\_host | A list of hosts from which administrators can connect to the server. The separators for the host names are: ‘;’, ‘,’, space, ‘\n’, ‘\r’.  The setting is taken into account by the RECO command.  Default: empty list which means any host is allowed. |
| admin\_client\_name | A list of client names which can execute commands requiring administrative privileges. The separators for the client names are: ‘;’, ‘,’, space, ‘\n’, ‘\r’.  The setting is taken into account by the RECO command.  Default: empty list which means that nobody will be able to execute administrative commands. |
| affinity\_high\_mark\_percentage | If the affinity registry has more records than specified by this parameter then aggressive cleaning of the registry is switched on.  Default: 90 (%, integer) |
| affinity\_low\_mark\_percentage | If the affinity registry has less records than specified by this parameter then no cleaning registry will be performed.  If the number of records is between affinity\_low\_mark\_percentage and affinity\_high\_mark\_percentage then a normal cleaning of the registry is switched on respecting the affinity\_dirt\_percentage value (see below).  Default: 50 (%, integer) |
| affinity\_high\_removal | Maximum number of records to be removed at one time by the garbage collector when aggressive cleaning is switched on.  Only those records which have no jobs associated with them are deleted.  Default: 1000 |
| affinity\_low\_removal | Maximum number of records to be removed at one time by the garbage collector when aggressive cleaning is switched off.  Only those records which have no jobs associated with them are deleted.  Default: 100 |
| affinity\_dirt\_percentage | If the number of delete candidate records in the registry is less than this value and the number of records the registry has is between affinity\_low\_mark\_percentage and affinity\_high\_mark\_percentage then there will be no cleaning.  Default: 20 (%, integer) |
| stat\_interval | Statistics thread logging interval (if allowed by settings above)  The value must be >= 1  Default: 10 (seconds, integer)  Introduced in NS 4.16.11 |
| max\_client\_data | Integer. Max size for the client transient data.  The value must be >= 1  Default: 2048 (bytes)  Introduced in NS 4.17.0 |
| reserve\_dump\_space | The size of the empty file which will be created in data/dump directory to reserve space for the queues flat files dump  Default: 1GB  Introduced in NS 4.23.0 |
| max\_groups | Maximum number of entries (per queue) the server can have in the group registry.  Default: 10000  **Note**: introduced in NS 4.25.0 |
| group\_high\_mark\_percentage | If the group registry has more records than specified by this parameter then aggressive cleaning of the registry is switched on.  Default: 90 (%, integer)  **Note**: introduced in NS 4.25.0 |
| group\_low\_mark\_percentage | If the group registry has less records than specified by this parameter then no cleaning registry will be performed.  If the number of records is between group\_low\_mark\_percentage and group\_high\_mark\_percentage then a normal cleaning of the registry is switched on respecting the group\_dirt\_percentage value (see below).  Default: 50 (%, integer)  **Note**: introduced in NS 4.25.0 |
| group\_high\_removal | Maximum number of records to be removed at one time by the garbage collector when aggressive cleaning is switched on.  Only those records which have no jobs associated with them are deleted.  Default: 1000  **Note**: introduced in NS 4.25.0 |
| group\_low\_removal | Maximum number of records to be removed at one time by the garbage collector when aggressive cleaning is switched off.  Only those records which have no jobs associated with them are deleted.  Default: 100  **Note**: introduced in NS 4.25.0 |
| group\_dirt\_percentage | If the number of delete candidate records in the registry is less than this value and the number of records the registry has is between group\_low\_mark\_percentage and group\_high\_mark\_percentage then there will be no cleaning.  Default: 20 (%, integer)  **Note**: introduced in NS 4.25.0 |
| max\_scopes | Maximum number of entries (per queue) the server can have in the scope registry.  Default: 10000  **Note**: introduced in NS 4.25.0 |
| scope\_high\_mark\_percentage | If the scope registry has more records than specified by this parameter then aggressive cleaning of the registry is switched on.  Default: 90 (%, integer)  **Note**: introduced in NS 4.25.0 |
| scope\_low\_mark\_percentage | If the scope registry has less records than specified by this parameter then no cleaning registry will be performed.  If the number of records is between scope\_low\_mark\_percentage and scope\_high\_mark\_percentage then a normal cleaning of the registry is switched on respecting the scope\_dirt\_percentage value (see below).  Default: 50 (%, integer)  **Note**: introduced in NS 4.25.0 |
| scope\_high\_removal | Maximum number of records to be removed at one time by the garbage collector when aggressive cleaning is switched on.  Only those records which have no jobs associated with them are deleted.  Default: 1000  **Note**: introduced in NS 4.25.0 |
| scope\_low\_removal | Maximum number of records to be removed at one time by the garbage collector when aggressive cleaning is switched off.  Only those records which have no jobs associated with them are deleted.  Default: 100  **Note**: introduced in NS 4.25.0 |
| scope\_dirt\_percentage | If the number of delete candidate records in the registry is less than this value and the number of records the registry has is between scope\_low\_mark\_percentage and scope\_high\_mark\_percentage then there will be no cleaning.  Default: 20 (%, integer)  **Note**: introduced in NS 4.25.0 |

## [log] section

|  |  |
| --- | --- |
| Value | Description |
| file | File name where the server stores the log messages. |

## [bdb] section

|  |  |
| --- | --- |
| Value | Description |
| force\_storage\_version | Version of the storage data model to be forced at the start time.  If a database exists and is not recreated at the startup, the server reads the data model version and then checks against the force\_storage\_version value. If the value does not match then the server does not start.  Default: the current version of the storage data model.  Obsolete in NS 4.23.0 |
| path | Directory where the database files are stored.  It is recommended to have this directory on the fastest available filesystem.  No default, the parameter is mandatory. |
| max\_queues | Maximum number of queues served by the server.  Default: 50 |
| mem\_size | Default: 0 |
| mutex\_max | Default: 0 |
| max\_locks | Default: 0 |
| max\_lockers | Default: 0 |
| max\_lockobjects | Default: 0 |
| log\_mem\_size | Default: 0 |
| checkpoint\_kb | Default: 5000 |
| checkpoint\_min | Default: 5 |
| sync\_transactions | Default: false |
| direct\_db | Default: false |
| direct\_log | Default: false |
| private\_env | Default: false |

## [service\_to\_queue] section

|  |  |
| --- | --- |
| Value | Description |
| <serviceID> | Provides the queue name serviceID corresponds to.  There could be arbitrary number of values. The service ID is not case sensitive.  The values are used to resolve the service to the queue name when  QINF2 service=…  command is received.  Example of the section:  [service\_to\_queue]  NS\_gMap\_DEV=gMap  NS\_VirusVariation=virus\_variation |

## [qclass\_YYY] section

The section introduces a new queue class YYY. The class holds all the queue parameters (see the [queue\_ZZZ] section description below) except the ‘class’ parameter. Later, the introduced classes could be used to create static queues (via the config file) and/or dynamic queues (via QCRE command). Classes do not introduce queues by themselves.

There could be as many ‘qclass\_’ prefixed sections as necessary. When a dynamic queue refers to a queue class name in the QCRE command or when a static queue refers to a class via its ‘class’ parameter, the ‘queue\_’ prefix should not be provided.

## [queue\_ZZZ] section

Each static queue must have a separate section which describes the queue settings. The queue name follows the ‘queue\_’ prefix in the section name, e.g. the section in the title describes the queue called ZZZ.

Starting from NetSchedule 4.14.0 a queue does not introduce a queue class name in no circumstances.

The table below describes settings which affect a specific queue only.

|  |  |
| --- | --- |
| Value | Description |
| class | Queue class to use for creating this queue. The queue will derive all the parameters from the class and those which are explicitly specified in the section will overwrite the class parameters.  The class is an optional parameter.  Default: empty string, i.e. no class will be used to derive  Introduced in NetSchedule 4.14.0 |
| timeout | Inactivity timeout for non-running and non-reading jobs which triggers the job to be marked for deletion.  Default: 3600 (float, seconds) |
| notif\_hifreq\_interval | Interval for available job notifications when they are sent with high frequency.  Default: 0.1 (float, seconds) |
| notif\_hifreq\_period | Period of time within which available job notifications are sent with high frequency if there were no requests from a worker node which requested a job earlier.  Default: 5 (float, seconds) |
| notif\_lofreq\_mult | Multiplier for the notif\_hifreq\_interval to calculate the interval between notifications when they are sent with low frequency.  Default: 50 (integer) |
| notif\_handicap | Delay for sending UDP notifications that there is a vacant job for all worker nodes except one. If configured (i.e. != 0) and there are more than one candidates for notifications then the one to send to will be picked randomly.  Default: 0.0 (float, seconds)  Introduced in NetSchedule 4.16.3 |
| dump\_buffer\_size | The size of a buffer for reading jobs from a database before dumping them.  Default: 100 (jobs) |
| dump\_client\_buffer\_size | Number of clients printed in a single batch in the STAT CLIENTS command. Allowed range is 100-10000.  Default: 100 (integer, clients) |
| dump\_aff\_buffer\_size | Number of affinities printed in a single batch in the STAT AFFINITIES command. Allowed range is 100-10000.  Default: 100 (integer, affinities) |
| dump\_group\_buffer\_size | Number of groups printed in a single batch in the STAT GROUPS command. Allowed range is 100-10000.  Default: 100 (integer, groups) |
| run\_timeout | If there is no information about a job in the Running state within this timeout then the server considers this try as failed and moves the job to the appropriate state.  The timeout is used only if there were no individual running timeout provided by the user.  Default: 3600 (float, seconds) |
| run\_timeout\_precision | The time interval which is used to check job expiration.  Default: 3600 (integer, seconds) for NS 4.16.7 and below  Default: 3 (float, seconds) for NS 4.16.8 and up  **Note**: obsolete for NS 4.20.0. It is calculated at the startup time. |
| read\_timeout | If there is no information about a job in the Reading state within this timeout then the server considers this try as failed and moves the job to the appropriate state.  The timeout is used only if there were no individual reading timeout provided by the user.  Default: 10 (float, seconds)  Introduced in NetSchedule 4.17.2 |
| program | List of client names and their versions which are allowed for the queue. When a client connects it is checked against this list and if the name is not in the list or the version is below allowed then the client will be rejected.  The separators for the programs are: ‘;’, ‘,’.  Default: empty string which means there are no restrictions. |
| failed\_retries | Number of retries to execute a job.  Default: 0 |
| read\_failed\_retries | Number of retries to read a job  Default: the value accepted for failed\_retries  Introduced in NetSchedule 4.20.0 |
| blacklist\_time | The maximum time a job will be kept in a blacklist till it can be given for execution to the same worker node after it failed the job.  Not supported in NS from 4.10.0 to 4.16.1 inclusive.  0 means that a job will not be in the blacklist at all.  Default: 2147483647 (float, seconds) |
| read\_blacklist\_time | The maximum time a job will be kept in a blacklist till it can be given for reading to the same reader after it failed reading the job.  0 means that a job will not be in the blacklist at all.  Default: the value accepted for blacklist\_time (float, seconds)  Introduced in NetSchedule 4.20.0 |
| max\_input\_size | Maximum size of a job input.  Default: 2048 (bytes) |
| max\_output\_size | Maximum size of a job output.  Default: 2048 (bytes) |
| subm\_host | A list of hosts which are allowed to submit jobs.  The separators for the host names are: ‘;’, ‘,’, space, ‘\n’, ‘\r’.  Default: empty string which means that any host can submit jobs. |
| wnode\_host | A list of hosts which are allowed to request jobs for execution.  The separators for the host names are: ‘;’, ‘,’, space, ‘\n’, ‘\r’.  Default: empty string which means that any host can request jobs for execution. |
| reader\_host | A list of hosts which are allowed to request jobs for reading.  The separators for the host names are: ';', ',', space, '\n', '\r'.  Default: empty string which means that any host can request jobs for reading. |
| wnode\_timeout | Worker node inactivity timeout.  If a registered worker node has no activity within the given timeout then it is marked as inactive and its affinities are cleared.  Default: 40 (float, seconds)  Introduced in NetSchedule 4.11.0 |
| reader\_timeout | Reader inactivity timeout in seconds.  If a reader has no activity within the given timeout then it is marked as inactive and its read preferred affinities are cleared  Default: 40 (float, seconds)  Introduced in NetSchedule 4.20.0 |
| pending\_timeout | Pending jobs timeout.  The timeout is measured starting from the submit time. If the job is still in the pending state (regardless of the pending to running to pending loops) when the timeout is detected, then the job will be deleted.  Default: 60\*60\*24\*7=604800 sec., i.e. 1 week (float)  Introduced in NetSchedule 4.11.0 |
| max\_pending\_wait\_timeout | Max time a pending job is not given to a worker node due to its affinity is not exclusively new.  The timeout is measured starting from the submit time.  The value 0.0 means that this feature is switched off.  Default: 0.0 (float, seconds)  Introduced in NetSchedule 4.13.0 |
| max\_pending\_read\_wait\_timeout | Max time a done, failed or canceled job is not given to a reader due to its affinity is not exclusively new.  The timeout is measured starting from the moment when the job first became available for reading.  The value 0.0 means the feature is switched off.  Default: 0.0 (float, seconds)  Introduced in NetSchedule 4.20.0 |
| netcache\_api | Reference to another section which specifies the NetCache API parameters. If a non-empty value is given then a section with this name must exist. If the section is not found the configuration file is considered invalid and will be rejected.  If the section is found then its content will be provided in the GETP2 command output.  Default: empty string  Introduced in NetSchedule 4.16.9  Obsolete and removed in 4.17.0 |
| scramble\_job\_keys | Controls how job keys are generated.  Regardless of the parameter value, NetSchedule is able to handle both scrambled and non-scrambled job keys. The parameter affects only how the job keys are printed (logged or sent to the clients via sockets).  Default: false, the job keys are not scrambled  Introduced in NetSchedule 4.16.10 |
| linked\_section\_PPP | References values from another section of the same configuration file.  There could be many parameters like this. PPP is an arbitrary prefix for the referenced section values output in the QINF2 command. For example if there is the following parameter:  linked\_section\_nc = other\_section  and there is a section like:  [other\_section]  name1 = value1  name2 = value2  Then the QINF2 for the queue will include the output below:  …&nc.name1=value1&nc.name2=value2 |
| client\_registry\_timeout\_worker\_node | This is a client registry garbage collector parameter.  A timeout of inactivity after which a worker node becomes a candidate for deletion.  The value must be greater than wnode\_timeout. If the provided value does not meet the criteria then (at the startup only) it will be calculated as max(2\*wnode\_timeout, 2\*run\_timeout, 3600)  Default: 3600 (float, seconds)  Introduced in NetSchedule 4.20.0 |
| client\_registry\_min\_worker\_nodes | This is a client registry garbage collector parameter.  The minimum number of worker nodes to be kept in the registry.  Default: 20 (integer)  Introduced in NetSchedule 4.20.0 |
| client\_registry\_timeout\_admin | This is a client registry garbage collector parameter.  A timeout of inactivity after which an admin becomes a candidate for deletion.  Default: 20 (float, seconds)  Introduced in NetSchedule 4.20.0 |
| client\_registry\_min\_admins | This is a client registry garbage collector parameter.  The minimum number of admins to be kept in the registry.  Default: 10 (integer)  Introduced in NetSchedule 4.20.0 |
| client\_registry\_timeout\_submitter | This is a client registry garbage collector parameter.  A timeout of inactivity after which a submitter becomes a candidate for deletion.  Default: 20 (float, seconds)  Introduced in NetSchedule 4.20.0 |
| client\_registry\_min\_submitters | This is a client registry garbage collector parameter.  The minimum number of submitters to be kept in the registry.  Default: 10 (integer)  Introduced in NetSchedule 4.20.0 |
| client\_registry\_timeout\_reader | This is a client registry garbage collector parameter.  A timeout of inactivity after which a reader becomes a candidate for deletion. If not provided then calculated as max(2\*reader\_timeout, 2\*read\_timeout, 600)  Default: 600 (float, seconds)  Introduced in NetSchedule 4.20.0 |
| client\_registry\_min\_readers | This is a client registry garbage collector parameter.  The minimum number of readers to be kept in the registry.  Default: 10 (integer)  Introduced in NetSchedule 4.20.0 |
| client\_registry\_timeout\_unknown | This is a client registry garbage collector parameter.  A timeout of inactivity after which an unknown type client becomes a candidate for deletion.  Default: 20 (float, seconds)  Introduced in NetSchedule 4.20.0 |
| client\_registry\_min\_unknowns | This is a client registry garbage collector parameter.  The minimum number of unknown type clients to be kept in the registry.  Default: 10 (integer)  Introduced in NetSchedule 4.20.0 |

## [error\_emulator] section

**Note 1**: this section is analyzed only if the code is compiled in debug mode.

**Note 2**: the effective NetSchedule values from this section cannot be changed using the RECONFIGURE command.

The section is introduced in NS 4.21.0

|  |  |
| --- | --- |
| Value | Description |
| fd\_report | It is a string value of the following format: F:Ff Fb-Fe  where  F integer, number of used FD in the HEALTH report instead the real usage. If -1 (default) then there will be no substitution.  Ff integer, frequency with which the corresponding event is emulated; zero means never; 1 means every time; 2 means every other time; etc. It's optional, default is: 1 (every time)  Fb-Fe integers, the range of requests's serial numbers for which the particular type of error emulation is turned on. It's optional, default is: from zero to MAX\_UINT |
| mem\_report | It is a string value of the following format: M:Fm Mb-Me  where  M integer, number of used memory bytes in the HEALTH report instead of the real usage. If -1 (default) then there will be no substitution.  Fm integer, frequency with which the corresponding event is emulated; zero means never; 1 means every time; 2 means every other time; etc. It's optional, default is: 1 (every time)  Mb-Me integers, the range of requests's serial numbers for which the particular type of error emulation is turned on. It's optional, default is: from zero to MAX\_UINT |
| delay | It is a string value of the following format: D:Fd Db-De  where  D double, delay in seconds before writing into the client socket  Fd integer, frequency with which the corresponding event is emulated; zero means never; 1 means every time; 2 means every other time; etc. It's optional, default is: 1 (every time)  Db-De integers, the range of requests's serial numbers for which the particular type of error emulation is turned on. It's optional, default is: from zero to MAX\_UINT |
| drop\_before\_reply | It is a string value of the following format: B:Fb Bb-Be  where  B boolean, if TRUE then the connection should be dropped straight before a response is written to the client  Fb integer, frequency with which the corresponding event is emulated; zero means never; 1 means every time; 2 means every other time; etc. It's optional, default is: 1 (every time)  Bb-Be integers, the range of requests's serial numbers for which the particular type of error emulation is turned on. It's optional, default is: from zero to MAX\_UINT |
| drop\_after\_reply | It is a string value of the following format: A:Fa Ab-Ae  where  A boolean, if TRUE then the connection should be dropped straight after a response is written to the client  Fa integer, frequency with which the corresponding event is emulated; zero means never; 1 means every time; 2 means every other time; etc. It's optional, default is: 1 (every time)  Ab-Ae integers, the range of requests's serial numbers for which the particular type of error emulation is turned on. It's optional, default is: from zero to MAX\_UINT |
| reply\_with\_garbage | It is a string value of the following format: G:Fg Gb-Ge  where  G boolean, if TRUE then the data specified below in the garbage\_data parameter will be sent instead of the real response  Fg integer, frequency with which the corresponding event is emulated; zero means never; 1 means every time; 2 means every other time; etc. It's optional, default is: 1 (every time)  Gb-Ge integers, the range of requests's serial numbers for which the particular type of error emulation is turned on. It's optional, default is: from zero to MAX\_UINT |
| garbage\_data | String  If reply\_with\_garbage is set to true then this will be sent to the client instead of the real response  Default: please define [error\_emulator]/garbage\_data parameter value |

# Appendix A. Response Depending on Security Token

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Job state | Pending | Running | Done | Reading |  |
| incoming command |  |
| GET/WGET (JXCG) | OK | N/A | N/A | N/A | Complete match |
| OK | N/A | N/A | N/A | Passport match |
| OK | N/A | N/A | N/A | No match |
| RETURN | ERR:eInvalidJobStatus | OK | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | Complete match |
| OK:WARNING | OK:WARNING | OK:WARNING | OK:WARNING | Passport match |
| ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | No match |
| PUT (JXCG) | OK | OK | OK:WARNING | ERR:eInvalidJobStatus | Complete match |
| OK | OK | OK:WARNING | ERR:eInvalidJobStatus | Passport match |
| ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | No match |
| FPUT | ERR:eInvalidJobStatus | OK | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | Complete match |
| OK:WARNING | OK:WARNING | OK:WARNING | OK:WARNING | Passport match |
| ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | No match |
| READ | N/A | N/A | OK | N/A | Complete match |
| N/A | N/A | OK | N/A | Passport match |
| N/A | N/A | OK | N/A | No match |
| RDRB | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | OK | Complete match |
| ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | OK:WARNING | OK:WARNING | Passport match |
| ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | No match |
| CFRM | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | OK | Complete match |
| ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | OK | OK | Passport match |
| ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | No match |
| FRED | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | OK | Complete match |
| ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | OK:WARNING | OK:WARNING | Passport match |
| ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | No match |
| CANCEL | OK | OK | OK | OK | Complete match |
| OK | OK | OK | OK | Passport match |
| OK | OK | OK | OK | No match |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Job state | Failed | ReadFailed | Confirmed | Canceled |  |
| incoming command |  |
| GET/WGET (JXCG) | N/A | N/A | N/A | N/A | Complete match |
| N/A | N/A | N/A | N/A | Passport match |
| N/A | N/A | N/A | N/A | No match |
| RETURN | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | Complete match |
| OK:WARNING | OK:WARNING | OK:WARNING | ERR:eInvalidJobStatus | Passport match |
| ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR: eInvalidAuthToken | No match |
| PUT (JXCG) | OK | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | Complete match |
| OK | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | Passport match |
| ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR: eInvalidAuthToken | No match |
| FPUT | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | Complete match |
| OK:WARNING | OK:WARNING | OK:WARNING | ERR:eInvalidJobStatus | Passport match |
| ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR: eInvalidAuthToken | No match |
| READ | OK | N/A | N/A | N/A | Complete match |
| OK | N/A | N/A | N/A | Passport match |
| OK | N/A | N/A | N/A | No match |
| RDRB | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | Complete match |
| OK:WARNING | OK:WARNING | OK:WARNING | ERR:eInvalidJobStatus | Passport match |
| ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR: eInvalidAuthToken | No match |
| CFRM | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | Complete match |
| ERR:eInvalidJobStatus | OK:WARNING | OK:WARNING | ERR:eInvalidJobStatus | Passport match |
| ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR: eInvalidAuthToken | No match |
| FRED | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | ERR:eInvalidJobStatus | Complete match |
| OK:WARNING | OK:WARNING | OK:WARNING | ERR:eInvalidJobStatus | Passport match |
| ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR:eInvalidAuthToken | ERR: eInvalidAuthToken | No match |
| CANCEL | OK | OK | OK | OK:WARNING | Complete match |
| OK | OK | OK | OK:WARNING | Passport match |
| OK | OK | OK | OK:WARNING | No match |

Notes:

* Anonymous clients do not provide security tokens, so they are treated as though they had a matching security token.
* The actual job state change happens only when a cell is marked as OK. If it is OK:WARNING then no job state change occurs.