|  |
| --- |
| NCBI |
| NetStorage Server |
| Overview and the Protocol Specification |
|  |
| **Sergey Satskiy** |
| **4/29/2013** |

|  |
| --- |
| Document version: 1.53 |

Changes history

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | What changes |
| 1.0 | Apr 29, 2013 | Sergey Satskiy | Initial revision |
| 1.1 | May 1, 2013 | Sergey Satskiy | Added a section provided by Dmitry Kazimirov |
| 1.2 | May 2, 2013 | Sergey Satskiy | Changes after review made by Denis Vakatov |
| 1.3 | May 3, 2012 | Sergey Satskiy | BYE request behavior updated |
| 1.4 | May 6, 2013 | Sergey Satskiy | Protocol structure changes;  BYE modifications; GETCLIENTSINFO described |
| 1.5 | May 7, 2013 | Sergey Satskiy | SessionID, ClientIP added; nearly empty sections for GETATTR and SETATTR are added |
| 1.6 | May 8, 2013 | Sergey Satskiy | Corrections for Error (moved to Errors) and Warning (moved to Warnings) fields. |
| 1.7 | May 15, 2013 | Sergey Satskiy, Dmitry Kazimirov | CREATE description added |
| 1.8 | May 16, 2013 | Dmitry Kazimirov | READ message description added |
| 1.9 | May 21, 2013 | Sergey Satskiy,  Dmitry Kazimirov | More commands, the other structuring |
| 1.10 | May 22, 2013 | Dmitry Kazimirov | GETOBJECTINFO added |
| 1.11 | Jan 23, 2014 | Sergey Satskiy | HEALTH, RECONFIGURE, ACKALERT commands; extended configuration parameters |
| 1.12 | Jan 27, 2014 | Sergey Satskiy | Splitting WRITE into CREATE and WRITE; introducing object accounting flag. |
| 1.13 | Jan 27, 2014 | Dmitry Kazimirov | Replaced the flag “Accountable” with “NoMetaData”; updated the WRITE and CREATE commands. |
| 1.14 | Feb 7, 2014 | Dmitry Kazimirov | Renamed everything that was NetFile to NetStorageObject. |
| 1.15 | Feb 12, 2014 | Dmitry Kazimirov | Renamed “object ID” to “object locator”; got rid of the Group and Owner fields. |
| 1.16 | Feb 14, 2014 | Dmitry Kazimirov | Added METADATA field description. |
| 1.17 | Feb 18, 2014 | Sergey Satskiy | Added DELATTR command |
| 1.18 | Feb 19, 2014 | Dmitry Kazimirov | Removed the METADATA field; added the metadata\_conf .ini file section. |
| 1.19 | Feb 28, 2014 | Sergey Satskiy | Adding GETMETADATACONF command |
| 1.20 | April 18, 2014 | Sergey Satskiy | Addig ‘User’ field to the ACKALERT input and HEALTH output |
| 1.21 | April 23, 2014 | Sergey Satskiy | Introducing ‘ncbi\_phid’ optional field to the incoming messages. |
| 1.22 | November 4, 2014 | Sergey Satskiy | The ‘Message’ field for alerts is added for NetStorage 1.0.2 |
| 1.23 | November 6, 2014 | Sergey Satskiy | Adding HELLO command Metadata field description for NetStorage 1.1.0 |
| 1.24 | November 19, 2014 | Sergey Satskiy | Adding SETTTL for NetStorage 1.1.0 |
| 1.25 | December 2, 2014 | Sergey Satskiy | Extending GETOBJECTINFO description. |
| 1.26 | December 9, 2014 | Sergey Satskiy | SETTTL replaced with SETEXPTIME; changes in the [metadata\_info] configuration section.  Extending GETMETADATAINFO output format with service expiration attributes. |
| 1.27 | December 17, 2014 | Sergey Satskiy | Adding GETATTRLIST for NetStorage 1.2.0 |
| 1.28 | December 29, 2014 | Sergey Satskiy | Adding GETCLIENTOBJECTS for NetStorage 1.2.0, adding MetadataOption field to the GETCLIENTSINFO output for NetStorage 1.2.0 |
| 1.29 | December 30, 2014 | Sergey Satskiy | Adding “DBClients” field to the GETCLIENTSINFO for NetStorage 1.2.0 |
| 1.30 | April 29, 2015 | Sergey Satskiy | Adding server/log\_timing for NST 2.0.1 |
| 1.31 | May 22, 2015 | Sergey Satskiy | Making the ‘ProtocolVersion’ field in the HELLO message optional and switching its type from integer to a string (like ‘1.0.0’). |
| 1.32 | June 15, 2015 | Sergey Satskiy | Adding server/log\_timing\_nst\_api and server/log\_timing\_nst\_api for NST 2.1.0 |
| 1.33 | August 25, 2015 | Sergey Satskiy | Rewriting the overview chapters to reflect the recent changes to the original ideas |
| 1.34 | December 11, 2015 | Sergey Satskiy | Adding the LOCKFTPATH message for NST 2.2.0 |
| 1.35 | December 24, 2015 | Sergey Satskiy | Extending the ISSUE element for NST 2.2.0 |
| 1.36 | January 6, 2016 | Sergey Satskiy | Extending the CONFIGURATION reply with new items. Updating the configuration parameters description in the .ini file. |
| 1.37 | January 12, 2016 | Sergey Satskiy | Extending the DELETE reply for NST 2.2.0 |
| 1.38 | January 13, 2016 | Sergey Satskiy | Adjustments to LOCKFTPATH message for NST 2.2.0 |
| 1.39 | January 20, 2016 | Sergey Satskiy | Introducing [server]/data\_path configuration parameter for NST 2.2.0 |
| 1.40 | February 10, 2016 | Sergey Satskiy | Changes in GETSIZE, EXISTS,  Messages for NST 2.2.0 |
| 1.41 | February 17, 2016 | Sergey Satskiy | Adding ‘prolong\_on\_relocate’ for NST 2.2.0 |
| 1.42 | February 32, 2016 | Sergey Satskiy | Adding DB record creation controlling optional flags to various messages for NST 2.2.0 |
| 1.43 | March 1, 2016 | Rafael Sadyrov | Introducing ‘ncbi\_context’ optional field to the incoming messages for NST 2.2.0 |
| 1.44 | March 18, 2016 | Sergey Satskiy | Dropped [server]/log\_timing setting for NST 2.2.0 |
| 1.45 | April 19, 2016 | Sergey Satskiy | Changed output format for GetClientsInfo for NST 2.2.0  Optional ‘ClientNamespace’ in HELLO and GETCLIENTOBJECTS for NST 2.2.0 |
| 1.46 | June 1, 2016 | Sergey Satskiy | Adding diagrams to highlight the AllowBackendFallback flag in the READ, DELETE and EXISTS messages. |
| 1.47 | June 6, 2016 | Sergey Satskiy | Updated diagram for the DELETE message |
| 1.48 | June 10, 2016 | Sergey Satskiy | Remove the AllowBackendFallback flag from the EXISTS message. |
| 1.49 | June 16, 2016 | Sergey Satskiy | New supported format of the prolong values |
| 1.50 | June 20, 2016 | Sergey Satskiy | Changes in the prolong\_on\_XXX parameters description; Updated SETEXPTIME message description. |
| 1.51 | July 6, 2016 | Sergey Satskiy | RELOCATE progress info messages for planned NST-2.4.0 |
| 1.52 | July 8, 2016 | Sergey Satskiy | Adding the NeedProgressReport flag for the RELOCATE message |
| 1.53 | July 27, 2016 | Sergey Satskiy | Adding GETUSERSINFO and GETUSEROBJECTS messages for NST 2.4.0 |

Table of Contents

[NetStorage Server (NST) 7](#_Toc457393985)

[Requirements 7](#_Toc457393986)

[Overview 8](#_Toc457393987)

[Basic Scenario: Creating an Object 8](#_Toc457393988)

[Metadata Database Access 9](#_Toc457393989)

[Communication Protocol 10](#_Toc457393990)

[Conventions Used in This Protocol Description 11](#_Toc457393991)

[Protocol Definition 13](#_Toc457393992)

[Common Fields of Client Requests 13](#_Toc457393993)

[SN – Serial Number 13](#_Toc457393994)

[CLIENT\_IDENTIFICATION – Indirect Client Tracking 13](#_Toc457393995)

[STD\_REQUEST – Common Client Request Fields 13](#_Toc457393996)

[STORAGE\_FLAGS – Requirements for Storage Back-ends 13](#_Toc457393997)

[ICACHE – NetCache Settings 14](#_Toc457393998)

[USER\_KEY – User-Defined Key 14](#_Toc457393999)

[OBJECT\_LOC – Unique NetStorage object locator 14](#_Toc457394000)

[OBJECT\_IDENTIFICATION –object locator or User Key 14](#_Toc457394001)

[Common Fields of Server Replies 15](#_Toc457394002)

[STATUS – Operation Result Status 15](#_Toc457394003)

[RE – Incoming Message Reference 15](#_Toc457394004)

[ISSUE – Warning or Error Report 15](#_Toc457394005)

[WARNINGS – Non-Fatal Conditions Occurred During Request Processing 15](#_Toc457394006)

[ERRORS – Conditions That Prevented Request from Being Processed 15](#_Toc457394007)

[STD\_REPLY – Common Server Reply Fields 15](#_Toc457394008)

[Client Requests and Server Responses 17](#_Toc457394009)

[INFO 17](#_Toc457394010)

[CONFIGURATION 17](#_Toc457394011)

[SHUTDOWN 18](#_Toc457394012)

[HELLO 18](#_Toc457394013)

[BYE 19](#_Toc457394014)

[GETCLIENTSINFO 20](#_Toc457394015)

[GETUSERSINFO 21](#_Toc457394016)

[GETMETADATAINFO 22](#_Toc457394017)

[GETOBJECTINFO 22](#_Toc457394018)

[GETATTRLIST 23](#_Toc457394019)

[GETCLIENTOBJECTS 24](#_Toc457394020)

[GETUSEROBJECTS 24](#_Toc457394021)

[GETATTR 25](#_Toc457394022)

[SETATTR 25](#_Toc457394023)

[DELATTR 26](#_Toc457394024)

[HEALTH 26](#_Toc457394025)

[ACKALERT 27](#_Toc457394026)

[SETEXPTIME 27](#_Toc457394027)

[RECONFIGURE 28](#_Toc457394028)

[CREATE 30](#_Toc457394029)

[WRITE 31](#_Toc457394030)

[READ 32](#_Toc457394031)

[DELETE 33](#_Toc457394032)

[RELOCATE 35](#_Toc457394033)

[EXISTS 37](#_Toc457394034)

[GETSIZE 37](#_Toc457394035)

[LOCKFTPATH 37](#_Toc457394036)

[Files Architecture 38](#_Toc457394037)

[Monitoring and Maintenance 38](#_Toc457394038)

[Commands 38](#_Toc457394039)

[AppLog 39](#_Toc457394040)

[grid\_cli Utility 39](#_Toc457394041)

[Python Module 40](#_Toc457394042)

[Command Line Arguments 40](#_Toc457394043)

[Configuration Parameters 40](#_Toc457394044)

[[server] section 40](#_Toc457394045)

[[log] section 41](#_Toc457394046)

[[metadata\_conf] section 41](#_Toc457394047)

[[database] section 42](#_Toc457394048)

# NetStorage Server (NST)

This document provides an overview of the NetStorage server functionality and requirements to the various aspects of the server lifecycle. Basically the server is a middle man which redirects storage requests between various storage service providers including FileTrack, NetCache and Amazon S3.

# Requirements

Below is a list of major requirements to the NetStorage server.

* The server must operate as a Linux operating system daemon.
* The server must read all the settings from a configuration file. Some of the parameters should be reconfigurable without restarting the server.
* The server must provide connectivity to FileTrack.
* The server must provide connectivity to NetCache.
* The server must provide connectivity to Amazon S3 (eventually).
* The server must serve many clients simultaneously.
* The server must receive clients’ information before performing any object operations (create/read/write).
* The logging facilities must be provided via AppLog.
* The server must register clients’ activity basing on the client identifiers.
* The server must provide an interface for monitoring.
* The clients must be able to provide additional information about themselves and this information must be stored in memory only. There is no need to restore this information after the server restart.
* The clients may provide some authorization specific information which could be used by the NST in the future releases.
* The protocol to communicate to the clients is UTTP.
* The server must support two options when it needs to select a backend storage:
  + The server makes the decision basing on the required storage properties
  + The client explicitly instructs the server where the data should be stored
* The server must support alerts
* The server must support user supplied attributes and store them in the external MS SQL database.
* From the MS SQL database prospective the server must support two modes of operating: with and without interactions with the MS SQL server. The mode is determined when the HELLO message is received from a client.

# Overview

NetStorage is a middle man server between clients and various storage service providers.

The diagram below shows the main actors and entities involved into a typical NetStorage application.



The NetStorage server is running on a Linux host and the only information it holds is transient information of the clients’ activity. This information is used to monitor the whole system.

The clients establish TCP/IP connections with the NetStorage server via an API, and they send both control information and data over the established connection.

## Basic Scenario: Creating an Object

Let’s consider a very basic scenario of creating a new object.



The first thing the NetStorage server expects from a client is its identification information. This information is used to keep track of the client activity. The information is provided in the HELLO message. Generally speaking a client may send as many HELLO messages as needed within one connection session. There is no need to send HELLO before each data exchange operation if nothing needs to be changed in the HELLO parameters. A general rule is that the latest HELLO parameters will be used for all the consequent operations.

Then the client instructs the server that an object needs to be created via the CREATE message. There are two options here. A client may provide its own object identification or it can ask the server to generate an object locator. Later on the generated locator or the client provided object identification can be used to refer to the object when certain operations are requested. The client can also specify a certain storage to be used or let the server to decide basing on the provided storage property preferences or defaults.

The last major step is sending data of the new object. The data are transferred to the selected storage.

In addition to the creation of objects the server supports reading, writing and relocating operations.

## Metadata Database Access

As it was mentioned above the server can work in two modes: with or without the MS SQL database assistance. The decision whether the server uses it is made when a HELLO message is received. The HELLO message has an optional parameter which can explicitly instruct the server how to work with the metadata database. If this parameter is not supplied then another optional parameter is analyzed: the service name the client uses. The server configuration file holds a list of services for which the metadata database should be used. If the client provided service matches of them then the metadata database will be used. Please note that MS SQL administrators reboot the MS servers relatively often so you might expect NetStorage to refuse requests because the corresponding MS SQL database is not available. Certainly the usage of the metadata database incurs some performance penalties as well.

The metadata database is used to store various objects properties and some client information. It includes object attributes, TTL and automatic prolongation of it on read and on write, last object access time, object expiration, arbitrary user provided object attributes, object read/write counters etc. All these data are available for monitoring via the GRID dashboard. Some metadata, notable a TTL, can be used to make a decision on what is allowed to do with an object. It is also possible to get a list of object which were created by a certain client.

# Communication Protocol

NetStorage uses a bunch of protocols. In order to communicate to certain storages it uses their specific protocols. These protocols are out of consideration in this document.

The clients are connecting to the NetStorage server using TCP/IP. On top of TCP/IP, the JSON-over-UTTP protocol is used. The basic protocol exchange unit is a pair <Request message> - <Response message> i.e. regardless of a command there is always a response. A request message could be initiated by a client or by a server. Both request and response messages have structure similar to that of JSON objects (see <http://www.json.org>), although the representation of these objects is not textual but rather binary and is transferred a sequence of UTTP chunks, numbers, and control symbols.

Although JSON-over-UTTP can transmit all JSON data types (arrays, objects, and scalars) as separate messages, the NetStorage protocol requires that each incoming and outgoing message is a JSON object (that is, a set of key-value pairs).

Here’s an example of such a message:

{  
 "SN": 12,  
 "Type": "BYE"  
}

All input message must contain at least two fields: message type (the ‘Type’ field), and a serial number (the ‘SN’ field).

All server responses must contain at least three fields: message type (the ‘Type’ field), the serial number of the originating request (the ‘RE’ field), and the ‘Status’ field, which must contain either OK or ERROR, depending on whether the requested operation has completed successfully or has failed.

Here’s a possible server reply to the command shown above:

{  
 "RE": 12,  
 "Type": "REPLY",  
 "Status": "OK"  
}

The subsections below describe the supported requests and responses. Although these structures do not describe the actual binary representation of the data transferred over the TCP connection, they do however translate into the corresponding calls of the JSON-over-UTTP API.

## Conventions Used in This Protocol Description

Note that although this convention resembles a Backus-Naur Form specification, the grammar of this protocol is not, in fact, context-free.

Scalar types are referred to as follows: <string>, <int>, <double>, and <bool>, where <int> is a 64-bit integer and <double> is a double-precision floating-point number. Additional non-normative restrictions may be defined after a scalar type name using the following syntax:

<type: free-text description of the restrictions>

For example:

<int: must be a power of 2>

Constants are defined according to the JSON specification, including the following keywords: true, false, and null. String constants are surrounded by double quotation marks.

Curly brackets are used whenever a JSON object is expected, and square brackets are used for JSON arrays. Semicolons separate keys from their values in JSON objects.

The following syntax for nonterminal symbols is used to declare repetitive fragments (the ellipsis denotes any valid JSON fragment):

<STRUCTURE> ::= ...

Example:

<CLIENT\_IDENTIFICATION> ::=  
 "SessionID": <string>,  
 "ClientIP": <string: IP address>

These nonterminal symbols are referred in other parts of the message definition, for example:

<REQUEST> ::=  
{  
 "SN": <int>,  
 <CLIENT\_IDENTIFICATION>  
}

Parentheses are used to group sequences of tokens. The vertical bar indicates a choice and takes higher precedence than commas and semicolons within the scope of paired brackets or parentheses. Example:

<REQUEST> ::=  
{  
 ("ObjectLoc": <OBJECT\_LOC> | <USERKEY>)  
}

An optional token or group of tokens is represented by a pair of question marks on both sides:

<REQUEST> ::=  
{  
 ?<CLIENT\_IDENTIFICATION>?,  
 ?<WARNINGS>?,  
}

To indicate that a token or a group of tokens can repeat, it is followed by an ellipsis:

<WARNINGS> ::= <ISSUE>, ...

## Protocol Definition

### Common Fields of Client Requests

#### SN – Serial Number

This field must be included in every client request.

<SN> ::= "SN": <int: positive>

#### CLIENT\_IDENTIFICATION – Indirect Client Tracking

This fragment defines two request fields that provide information about the remote client on behalf of which the request is performed.

<CLIENT\_IDENTIFICATION> ::=  
 "SessionID": <string>,  
 "ClientIP": <string: IP address>,  
 ?("ncbi\_phid": <string>),?

?("ncbi\_context": <string>)?

#### STD\_REQUEST – Common Client Request Fields

These fields appear in most client requests.

<STD\_REQUEST> ::=  
 <SN>,  
 ?<CLIENT\_IDENTIFICATION>?

#### STORAGE\_FLAGS – Requirements for Storage Back-ends

A combination of these optional flags defines which storage back-end will be chosen by the server for blob creation or relocation:

* The “Fast” flag suggests using a fast storage (i.e. NetCache).
* “Persistent” (which takes precedence over “Fast”) suggests using a long-term storage (i.e. FileTrack).
* “Movable” can only be used during blob creation and provides for blob relocation between storage types at a later time.
* “Cacheable” enables blob caching using NetCache.
* “NoMetaData” disables using the metadata database for operations on the new object. By default, the database is used if the service name specified in the HELLO command is present in the [metadata\_conf] configuration section.

The default value of all flags is false, that is, none of the flags is set by default.

<STORAGE\_FLAGS> ::= "StorageFlags":  
{  
 ?("Fast": <boolean>),?  
 ?("Persistent": <boolean>),?  
 ?("Movable": <boolean>),?  
 ?("Cacheable": <boolean>),?  
 ?("NoMetaData": <boolean>)?  
}

#### ICACHE – NetCache Settings

This set of parameters is used for storing blobs in NetCache as well as for caching.

<ICACHE> ::= "ICache":  
{  
 "ServiceName": <string: LBSM service name>,  
 "CacheName": <string: NetCache database name>  
}

#### USER\_KEY – User-Defined Key

This type of keys allows users to use their own namespace and key names to address blobs.

<USER\_KEY> ::= "UserKey":  
{  
 "UniqueID": <string>,  
 "AppDomain": <string>  
}

#### OBJECT\_LOC – Unique NetStorage object locator

Base64url-encoded string returned by the object creation requests.

<OBJECT\_LOC> ::=  
 "ObjectLoc": <string: base64url-encoded object locator>

#### OBJECT\_IDENTIFICATION –object locator or User Key

This field is required in all commands that access an existing object. The field contains of either the object locator generated by the NetStorage server or a user-supplied key with additional location information.

<OBJECT\_IDENTIFICATION> ::=  
 (<OBJECT\_LOC> | (<USER\_KEY>,  
 ?<STORAGE\_FLAGS>?,  
 ?<ICACHE>?))

### Common Fields of Server Replies

#### STATUS – Operation Result Status

A required part of server replies that denotes whether the operation succeeded or not.

<STATUS> ::= "Status": ("OK" | "ERROR")

#### RE – Incoming Message Reference

This field of the server reply must contain the serial number of the originating client request.

<RE> ::= "RE": <int: taken from the SN field of the incoming message>

#### ISSUE – Warning or Error Report

The ISSUE structure is used to describe errors and warnings.

<ISSUE> ::=  
{  
 "Code": <int>,  
 "Message": <string>,  
 "Scope": <string>,  
 "SubCode": <int>  
}

The ‘Scope’ field value is one of the following:

* An exception class name for all descendants of the CException class
* ‘std::exception’ for the standard C++ exceptions and all the deriving classes
* ‘unknown\_exception’ for the cases when an exception type is not recognized
* ‘IMessage’ for the cases when an issue is formed basing on an IMessage instance
* ‘logic’ for the cases when NetStorage decides to create an issue without involving the C++ exceptions mechanism.

#### WARNINGS – Non-Fatal Conditions Occurred During Request Processing

<WARNINGS> ::= "Warnings": [<ISSUE>, ...]

#### ERRORS – Conditions That Prevented Request from Being Processed

<ERRORS> ::= "Errors": [<ISSUE>, ...]

#### STD\_REPLY – Common Server Reply Fields

These fields appear in most server replies.

<STD\_REPLY> ::=  
 "Type": "REPLY",  
 <STATUS>,  
 <RE>,  
 ?<WARNINGS>?,  
 ?<ERRORS>?,

### Client Requests and Server Responses

#### INFO

Inquire about the current server version, binary location, and build date.

The request can appear at any moment.

**Input message:**

<INFO\_REQUEST> ::=  
{  
 "Type": "INFO",  
 <STD\_REQUEST>  
}

**Output message:**

<INFO\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 "ServerVersion": <string>,  
 "ProtocolVersion": <string>,  
 "PID": <int: positive>,  
 "BuildDate": <string: date representation with second precision>,  
 "StartDate": <string: date representation with second precision>,  
 "ServerSession": <string>,  
 "ServerBinaryPath": <string: absolute pathname>,  
 "ServerCommandLine": <string>  
}

#### CONFIGURATION

Retrieve the actual configuration of the server. The entire content of the configuration file is returned in the “Configuration” field of the server reply.

The request can appear at any moment.

**Input message:**

<CONFIGURATION\_REQUEST> ::=  
{  
 "Type": "CONFIGURATION",  
 <STD\_REQUEST>  
}

**Output message:**

<CONFIGURATION\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 "Configuration": <string>,  
 "ConfigurationFilePath": <string: absolute pathname>,  
 "BackendConfiguration": <dictionary (v.2.2.0 and up)>,  
 "DBExecuteSPTimeout": <float (v.2.2.0 and up)>  
}

#### SHUTDOWN

This command shuts the server down. It requires that the client has issued the HELLO command earlier. The client name must match one of the configured administrator names. There are two types of shutdown requests: soft and hard. Soft mode allows the requests that are currently being executed by the server to complete naturally while rejecting new connections. With hard mode, the server terminates immediately and abruptly before completing the currently running requests.

The request can appear at any moment.

**Input message:**

<SHUTDOWN\_REQUEST> ::=  
{  
 "Type": "SHUTDOWN",  
 <STD\_REQUEST>,  
 "Mode": ("soft" | "hard")  
}

**Output message:**

<SHUTDOWN\_RESPONSE> ::= {<STD\_REPLY>}

Please note that in “hard” shutdown mode, the reply will not be sent.

#### HELLO

The request sets the current client identification for the connection. If another HELLO request was received earlier, then it is overwritten by the latest one. Some other request may have a prerequisite of the HELLO request issued for the connection.

The request can appear at any moment.

The "Service" field is required for most operations with object data (e.g. CREATE, WRITE, GETOBJECTINFO, etc.).

**Input message:**

<HELLO\_REQUEST> ::=  
{  
 "Type": "HELLO",  
 <STD\_REQUEST>,  
 "Client": <string>,  
 "ClientNamespace": [string: default is an empty string, client name namespace; since NST 2.2.0 and up],  
 "Application": <string: pathname>,  
 ?("Service": <string: NetStorage LBSM service name>)?,  
 ?("Metadata": <string: see description below>)?,  
 ?("ProtocolVersion": <string: three "." separated digits, e.g. "1.0.0" >)?,  
 ?("Ticket": <string>)?  
}

The optional field “Ticket” can be used to authorize the client to perform certain operations.

The optional field “Metadata” is introduced in NetStorage 1.1.0. It supports the values described in the table below:

|  |  |
| --- | --- |
| Metadata field value | Description |
| Required | If server-side configuration doesn't list the provided service, then the command fails (whether it's a HELLO or a per-object command in which the service contained in the object's key is different from the "HELLO-session" one) |
| Disabled | CREATE creates blobs without metadata (incl. in the object locator); no modification of the database whatsoever; commands like GET/SETATTR will fail |
| Monitoring | no data- nor attribute-changing commands (like CREATE, WRITE, SETATTR) are allowed; also, data access won't change server-side state (such as last-access time) |
| Not provided | each command checks if the effective service is listed in the server-side configuration, and behaves accordingly |

Note: If service is specified in the key, then it'll become the effective service (overriding the one specified in the HELLO).

Note: If the ‘ProtocolVersion’ field is not provided then the server will implicitly consider it as "1.0.0".

**Output message:**

<HELLO\_RESPONSE> ::= {<STD\_REPLY>}

#### BYE

The server does not close the connection. It is supposed that the client will close the connection upon receiving the server response. If any requests are received after the BYE request, the server replies with an error code and closes the connection.

The request can appear at any moment.

**Input message:**

<BYE\_REQUEST> ::=  
{  
 "Type": "BYE",  
 <STD\_REQUEST>  
}

**Output message:**

<BYE\_RESPONSE> ::= {<STD\_REPLY>}

#### GETCLIENTSINFO

The request can appear at any moment.

**Input message:**

<CLIENT\_INFO\_REQUEST> ::=  
{  
 "Type": "GETCLIENTSINFO",  
 <STD\_REQUEST>  
}

**Output message:**

<CLIENT\_INFO\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 "Clients":  
 [  
 ?{  
 "Name": <string>,  
 "Namespace": <string, provided starting from NST 2.2.0>  
 "Application": <string: pathname>,  
 "TicketProvided": <boolean>,  
 "Type": <string>,  
 "PeerAddress": <string: host name or IP address>,  
 "RegistrationTime": <string>,  
 "LastAccess": <string>,  
 "BytesWritten": <int: unsigned>,  
 "BytesRead": <int: unsigned>,  
 "BytesRelocated": <int: unsigned>,  
 "ObjectsWritten": <int: unsigned>,  
 "ObjectsRead": <int: unsigned>,  
 "ObjectsRelocated": <int: unsigned>,  
 "SocketErrors": <int: unsigned>,  
 "MetadataOption": <string>  
 }?,  
 ...  
 ],  
 "DBClients": <string> || [ <string>?, ... ]  
}

For each client in the retuned structure, “Name” and “Application” contain the same values that the client specified earlier in the HELLO command.

The “DbClients” field may be a plain string or a list of strings or a list of dictionaries as described below

|  |  |
| --- | --- |
| “DbClients” field | Description |
| String | It may happened in two cases:   * the meta info database access was not granted due to HELLO metadata option or due to the HELLO service is not configured * there were errors while retrieving data from the database. In this case a corresponding warning will be attached to the reply message |
| List of strings | The strings in the list are the names of the DB registered clients  **NOTE**: up to NST 2.2.0 |
| List of dictionaries | Each dictionary has two items:  ClientNamespace: <string>  ClientName: <string>  **NOTE**: up to NST 2.3.0 |

#### GETUSERSINFO

The request can appear at any moment.

Note: The message is introduced in NST 2.4.0

**Input message:**

<USER\_INFO\_REQUEST> ::=  
{  
 "Type": "GETUSERSINFO",  
 <STD\_REQUEST>  
}

**Output message:**

<USER\_INFO\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 "Users":  
 [  
 ?{  
 "Name": <string>,  
 "Namespace": <string>  
 }?,  
 ...  
 ]  
}

The user name and user namespace are coming from the request context received with the object modifying messages.

#### GETMETADATAINFO

The request can appear at any moment.

**Input message:**

<CLIENT\_INFO\_REQUEST> ::=  
{  
 "Type": "GETMETADATAINFO",  
 <STD\_REQUEST>  
}

**Output message:**

<CLIENT\_INFO\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 "Services":  
 [ { "Name": <string>,  
 "TTL": <string>,  
 "ProlongOnRead": <string>  
 "ProlongOnWrite": <string> } ?, ...? ]  
}

#### GETOBJECTINFO

Retrieve detailed information about an object from its current storage back-end.

**Input message:**

<GETOBJECTINFO\_REQUEST> ::=  
{  
 "Type": "GETOBJECTINFO",  
 <STD\_REQUEST>,  
 <OBJECT\_IDENTIFICATION>  
}

**Output message:**

The fields “Location” and “ObjectLocInfo” and “CreationTime” and “ExpirationTime” are always present. The rest of the fields (“Size”, “StorageSpecificInfo”) are present only if “Location” is not equal to “NotFound”.

<GETOBJECTINFO\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 "Location": <string: "NotFound" | "NetCache" | "FileTrack">,  
 "ObjectLocInfo": {...},  
 "CreationTime": <string>,  
 "ExpirationTime": <string>,  
 ?("Size": <int: unsigned>,  
 "StorageSpecificInfo": {...})?  
}

“CreationTime” and “ExpirationTime” fields may have one of the fixed values or an actual timestamp as described below.

|  |  |
| --- | --- |
| Value | Description |
| Timestamp | This is not a fixed value. It is the actual timestamp in a format provided by CTime::AsString(). |
| NotSet | Fixed value.  It means that the object record has been found in the meta info database however the corresponding cell has the NULL value. |
| NoMetadataFound | Fixed value.  It means that the object record has not been found in the meta info database. |
| NoMetadataAccess | Fixed value.  It means that the meta info database was restricted by some reasons. |
| MetadataAccessWarning | Fixed value.  It means that there was a problem while getting access to the meta info database. In this case the response also has a warning attached which has more specific information. |

A response with an error is provided if the object is expired.

#### GETATTRLIST

Return a list of attribute names for the given object. It requires that the client has issued the HELLO command earlier and the client name must not be empty.

**Input message:**

<GETATTRLIST\_REQUEST> ::=  
{  
 "Type": "GETATTRLIST",  
 <STD\_REQUEST>,  
 <OBJECT\_IDENTIFICATION>,  
}

**Output message:**

<GETATTRLIST\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 { "AttributeNames": [ <string>?, ... ] }  
}

#### GETCLIENTOBJECTS

Return a list of client object locators as well as the total number of client objects. It requires that the client has issued the HELLO command earlier and the client name must not be empty.

**Input message:**

<GETCLIENTOBJECTS\_REQUEST> ::=  
{  
 "Type": "GETCLIENTOBJECTS",  
 "ClientName": <string>,  
 ["ClientNamespace": <string: default is an empty string, client name namespace; since NST 2.2.0 and up>,]  
 ["Limit": <integer greater than zero>,]  
 <STD\_REQUEST>,  
 <OBJECT\_IDENTIFICATION>,  
}

**Output message:**

<GETCLIENTOBJECTS\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 { "ObjectLocators": [ <string>?, ... ],  
 "TotalClientObjects": <integer> }  
}

#### GETUSEROBJECTS

Return a list of user object locators as well as the total number of the user objects. It requires that the client has issued the HELLO command earlier and the user name and the user namespace must not be empty.

**Input message:**

<GETUSEROBJECTS\_REQUEST> ::=  
{  
 "Type": "GETUSEROBJECTS",  
 "UserName": <string>,  
 "UserNamespace": <string >,  
 ["Limit": <integer greater than zero>,]  
 <STD\_REQUEST>,  
 <OBJECT\_IDENTIFICATION>,  
}

**Output message:**

<GETUSEROBJECTS\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 { "ObjectLocators": [ <string>?, ... ],  
 "TotalUserObjects": <integer> }  
}

#### GETATTR

Return the current value of the specified object attribute. It requires that the client has issued the HELLO command earlier and the client name must not be empty.

**Input message:**

<GETATTR\_REQUEST> ::=  
{  
 "Type": "GETATTR",  
 <STD\_REQUEST>,  
 <OBJECT\_IDENTIFICATION>,  
 "AttrName": <string identifier: Base64url alphabet>  
}

**Output message:**

<GETATTR\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 "AttrValue": <string>  
}

#### SETATTR

Set a new value for the specified object attribute. It requires that the client has issued the HELLO command earlier and the client name must not be empty.

**Input message:**

<SETATTR\_REQUEST> ::=  
{  
 "Type": "SETATTR",  
 <STD\_REQUEST>,  
 <OBJECT\_IDENTIFICATION>,  
 "CreateIfNotFound": [Boolean, default: True,  
 since NST 2.2.0],  
 "AttrName": <string identifier: Base64url alphabet>,  
 "AttrValue": <string>  
}

**Output message:**

<SETATTR\_RESPONSE> ::= {<STD\_REPLY>}

#### DELATTR

Deletes the specified object attribute. It requires that the client has issued the HELLO command earlier and the client name must not be empty.

**Input message:**

<DELATTR\_REQUEST> ::=  
{  
 "Type": "DELATTR",  
 <STD\_REQUEST>,  
 <OBJECT\_IDENTIFICATION>,  
 "AttrName": <string identifier: Base64url alphabet>  
}

**Output message:**

<DELATTR\_RESPONSE> ::= {<STD\_REPLY>}

#### HEALTH

Provides the server wide health information. It requires that the client has issued the HELLO command earlier. The client name must match one of the configured administrator names.

**Input message:**

<HEALTH\_REQUEST> ::=  
{  
 "Type": "HEALTH",  
 <STD\_REQUEST>  
}

**Output message:**

<HEALTH\_RESPONSE> ::=   
{  
 <STD\_REPLY>,  
 "Alerts":  
 [  
 ?{  
 "Name": <string>,  
 "Acknowledged": <bool>,  
 "Count": <int: unsigned>,  
 "LastOccured": <string: datetime>,  
 "LastAcknowledged": <string: datetime>,  
 "User": <string>  
 "Message": <string>  
 }?,  
 ...  
 ]  
}

The count shows the number of times the alert occurred.

Note: the ‘Message’ field is added for NetStorage 1.0.2.

#### ACKALERT

Acknowledges an alert. It requires that the client has issued the HELLO command earlier. The client name must match one of the configured administrator names.

**Input message:**

<ACKALERT\_REQUEST> ::=  
{  
 "Type": "ACKALERT",  
 "Name": <string>,  
 "User": <string>,  
 <STD\_REQUEST>  
}

**Output message:**

<ACKALERT\_RESPONSE> ::=   
{  
 <STD\_REPLY>  
}

#### SETEXPTIME

Sets an object time to live. It requires that the client has issued the HELLO command earlier.

Note: the command is introduced for NST 1.1.0.

Note: NST 2.3.0 also uses the provided value (if not ‘infinity’) as an object individual TTL to be stored in the database. The object specific TTL may be used to calculate the prolongation time (if configured as <multiplier ttl>, see prolong\_on\_XXX configuration parameters description).

**Input message:**

<SETTL\_REQUEST> ::=  
{  
 "Type": "SETEXPTIME",  
 <STD\_REQUEST>,  
 "CreateIfNotFound": [Boolean, default: True,  
 since NST 2.2.0],  
 <OBJECT\_IDENTIFICATION>,  
 "TTL": <string>  
}

**Output message:**

<SETTTL\_RESPONSE> ::=   
{  
 <STD\_REPLY>  
}

The TTL parameter is a time to live for the object counted from the moment when the server handles the command. The accepted time span format is “dTh:m:s”. A special string value (case insensitive) “infinity” is also supported. If “infinity” is provided then the object will never expire.

#### RECONFIGURE

Reconfigures the server without restarting. It requires that the client has issued the HELLO command earlier. The client name must match one of the configured administrator names.

Note: the ‘Database’ section values could not be reconfigured. The only way to change the database access parameters is view restarting the server.

**Input message:**

<RECONFIGURE\_REQUEST> ::=  
{  
 "Type": "RECONFIGURE",  
 <STD\_REQUEST>  
}

**Output message:**

<RECONFIGURE\_RESPONSE> ::=   
{  
 <STD\_REPLY>  
 "What": <free-form JSON object or array>  
}

#### CREATE

The CREATE commands creates an object in a storage back-end and uploads data to the created object. The storage back-end is chosen based on the combination of flags defined by the “StorageFlags” field.

The command requires that the client has issued the HELLO command earlier and the client name must not be empty.

A communication between the client and the server during CREATE is slightly different from most other requests. After the request message, the client sends a sequence of UTTP chunks containing the data to be written. The server sends two confirmation messages: one after the request message is received, and another one after the data has been written:

Client Server  
------ ------  
  
<CREATE\_REQUEST> ->  
 <- <CREATE\_RESPONSE>  
  
UTTP chunks ->  
UTTP control symbol '\n' ->  
  
 <- <CREATE\_CONFIRMATION>

Note that the server accepts data chunks only if the “Status” field in CREATE\_RESPONSE is “OK”; the client must read CREATE\_RESPONSE completely prior to sending any data to the server.

**Input message:**

<CREATE\_REQUEST> ::=  
{  
 "Type": "CREATE",  
 <STD\_REQUEST>,  
 ?<STORAGE\_FLAGS>?,  
 ?<ICACHE>?  
}

**Output message:**

<CREATE\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 <OBJECT\_LOC>  
}

**Confirmation message:**

<CREATE\_CONFIRMATION> ::= {<STD\_REPLY>}

#### WRITE

The WRITE command uploads data to the specified object.

It requires that the client has issued the HELLO command earlier and the client name must not be empty.

Communication between the client and the server during WRITE is slightly different from most other requests. After the request message, the client sends a sequence of UTTP chunks containing the data to be written. The server sends two confirmation messages: one after the request message is received, and another one after the data has been written:

Client Server  
------ ------  
  
<WRITE\_REQUEST> ->  
 <- <WRITE\_RESPONSE>  
  
UTTP chunks ->  
UTTP control symbol '\n' ->  
  
 <- <WRITE\_CONFIRMATION>

Note that the server accepts data chunks only if the “Status” field in WRITE\_RESPONSE is “OK”; the client must read WRITE\_RESPONSE completely prior to sending any data to the server.

**Input message:**

<WRITE\_REQUEST> ::=  
{  
 "Type": "WRITE",  
 <STD\_REQUEST>,  
 "CreateIfNotFound": [Boolean, default: True,  
 since NST 2.2.0],  
 <OBJECT\_IDENTIFICATION>  
}

**Output message:**

<WRITE\_RESPONSE> ::= {<STD\_REPLY>}

**Confirmation message:**

<WRITE\_CONFIRMATION> ::= {<STD\_REPLY>}

#### READ

Retrieve object data by its ID.

It requires that the client has issued the HELLO command earlier and the client name must not be empty.

In response to the input READ message, the server sends two messages with object data in between:

Client Server  
------ ------  
  
<READ\_REQUEST> ->  
  
 <- <READ\_RESPONSE>  
 <- UTTP chunks  
 <- UTTP control symbol '\n'  
 <- <READ\_CONFIRMATION>

If the response to the READ message was OK, then the server sends a sequence of UTTP packets with object content. In case of an error during transmission, the server will terminate the sequence of UTTP packets by sending a zero-length UTTP packet followed by an error message. In case if the entire object content has been transferred successfully, the server will send a confirmation message.

**Input message:**

<READ\_REQUEST> ::=  
{  
 "Type": "READ",  
 <STD\_REQUEST>,  
 "AllowBackendFallback": [Boolean, default: True,  
 since NST 2.2.0],  
 <OBJECT\_IDENTIFICATION>  
}

**Output message:**

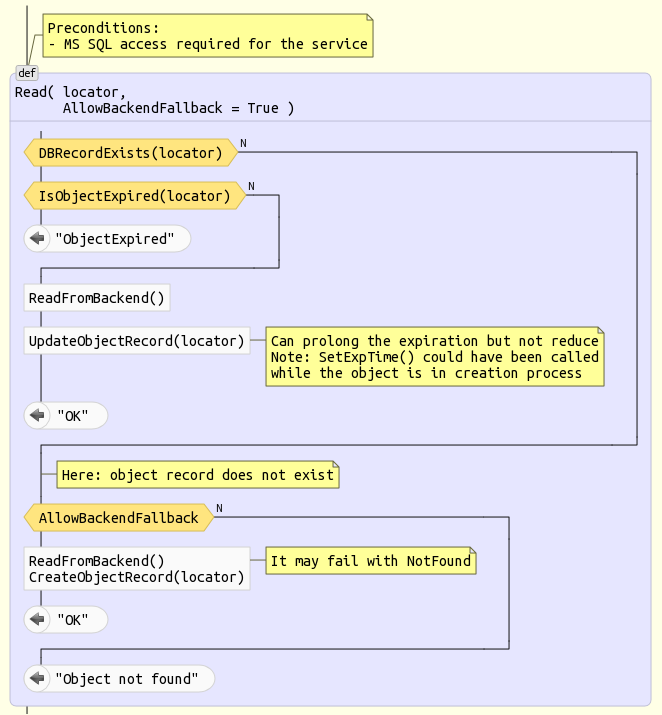
<READ\_RESPONSE> ::= {<STD\_REPLY>}

**Confirmation message:**

<READ\_CONFIRMATION> ::= {<STD\_REPLY>}

##### Implementation Highlights

The diagram below highlights the message implementation details including the AllowBackendFallback flag role.



#### DELETE

Delete an object from the underlying storage back-end.

It requires that the client has issued the HELLO command earlier and the client name must not be empty.

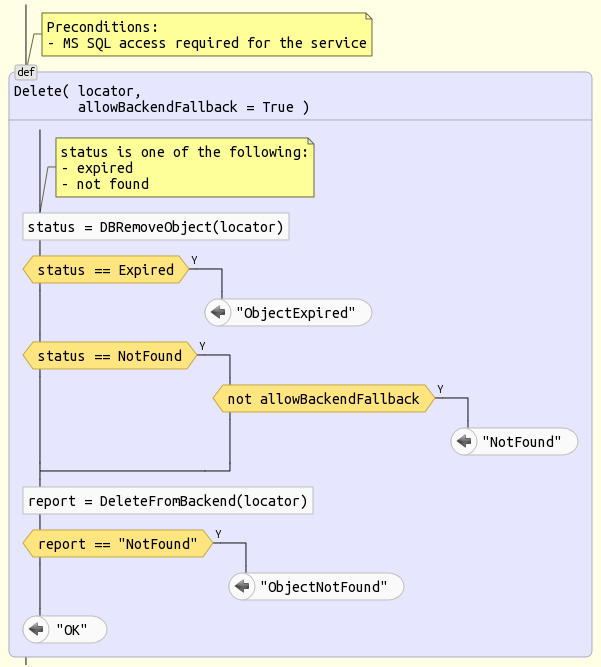
<DELETE\_REQUEST> ::=  
{  
 "Type": "DELETE",  
 <STD\_REQUEST>,  
 "AllowBackendFallback": [Boolean, default: True,  
 since NST 2.2.0],  
 <OBJECT\_IDENTIFICATION>  
}

**Output message:**

<DELETE\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 "NotFound": <Boolean: true => not found in the backend storage,  
 false => was found and deleted;  
 Introduced in NST 2.2.0>  
}

##### Implementation Highlights

The diagram below highlights the message implementation details including the AllowBackendFallback flag role.



#### RELOCATE

Move the specified object from one storage back-end to another.

It requires that the client has issued the HELLO command earlier and the client name must not be empty.

<RELOCATE\_REQUEST> ::=  
{  
 "Type": "RELOCATE",  
 <STD\_REQUEST>,  
 <OBJECT\_IDENTIFICATION>,  
 "CreateIfNotFound": [Boolean, default: True,  
 since NST 2.2.0],  
 "NeedProgressReport": [Boolean, default: False,  
 since NST 2.4.0 (planned)]  
 "NewLocation":  
 {  
 <STORAGE\_FLAGS>,  
 ?<ICACHE>?  
 }  
}

**Output message:**

<RELOCATE\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 <OBJECT\_LOC>  
}

The returned object locator reflects the new object location and can be used instead of the original object locator for faster object access.

**Progress messages:**

**Note**: starting from NST-2.4.0 (planned)

Sometimes the process of relocating objects may take a long time so the client may close a connection on a timeout while the operation is still in progress. Another consideration is that the client may want to know the progress of the relocation.

So strating from NST 2.4.0 (planned) the server will regularly send back a progress report messages as follows (if the incoming message had the “NeedProgressReport” flag set to True):

<PROGRESS\_REPORT> ::=  
{  
 <STD\_REPLY>,  
 "ProgressInfo":  
 {  
 "BytesRelocated": [Integer],  
 "Message": [String],  
 ?<Other fields>?  
 }  
}

#### EXISTS

Tests if the specified object exists in the underlying storage.

The request can appear at any moment.

<EXISTS\_REQUEST> ::=  
{  
 "Type": "EXISTS",  
 <STD\_REQUEST>,  
 <OBJECT\_IDENTIFICATION>  
}

**Output message:**

<EXISTS\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 "Exists": <boolean>  
}

#### GETSIZE

Returns object size from the underlying storage.

The request can appear at any moment.

<GETSIZE\_REQUEST> ::=  
{  
 "Type": "GETSIZE",  
 "ConsultBackendIfNoDBRecord": [Boolean, default: True,  
 since NST 2.2.0]  
 <STD\_REQUEST>,  
 <OBJECT\_IDENTIFICATION>  
}

**Output message:**

<GETSIZE\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 "Size": <int: unsigned>  
}

#### LOCKFTPATH

Locks or unlocks file track objects.

The request can appear at any moment. Implemented in NST 2.2.0 and up.

<LOCKFTPATH\_REQUEST> ::=  
{  
 "Type": "LOCKFTPATH",  
 <STD\_REQUEST>,  
 <OBJECT\_IDENTIFICATION>  
}

**Output message:**

<LOCKFTPATH\_RESPONSE> ::=  
{  
 <STD\_REPLY>,  
 "Path": <string>  
}

# Files Architecture

The diagram below shows the files used by NetStorage server.



NetStorage reads its configuration file (usually named netstoraged.ini) and configures the actual data storages correspondingly.

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

# Monitoring and Maintenance

NetStorage 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.

## Commands

The table below describes the commands which are usually associated with monitoring and maintenance.

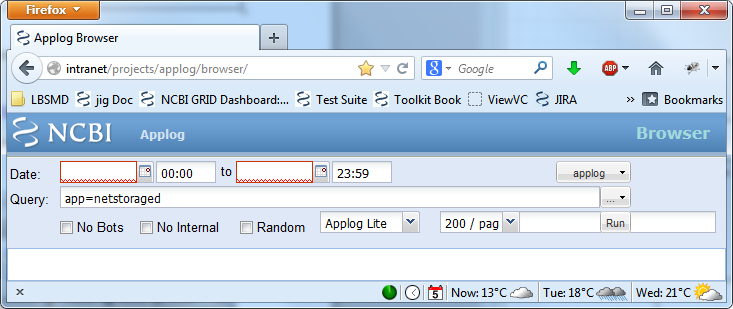
|  |  |
| --- | --- |
| Command | Description |
| CONFIGURATION | Provides the content of the current configuration file. |
| INFO | Provides the server version, the protocol version, the server PID and the build and start dates. |
| HEALTH | Provides the information about general server health, notably includes the alerts information. |
| ACKALERT | Acknowledges an alert |
| RECONFIGURE | Reconfigures the server |

## AppLog

The NetStorage 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=netstoraged in it. It is also recommended to have the “No Bots” and “No Internal” check boxes unchecked.



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. NetStorage does not use status codes in the range 300 – 399.

## grid\_cli Utility

The grid\_cli command line utility is planned to be extended with commands specific to NetStorage. These commands however have not been implemented yet at the time of writing. To see the grid\_cli utility commands type:

grid\_cli --help

## Python Module

A python module which currently serves connectivity to NetSchedule and partially to NetCache could potentially be extended with connectivity to NetStorage. There no plans however when and whether such support will be introduced.

# Command Line Arguments

The table below describes the server command line arguments.

|  |  |
| --- | --- |
| Argument | Description |
| -help | Prints help message and exits. |
| -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. |
| -pidfile | The file where the server saves its PID. |

# Configuration Parameters

NetStorage reads the configuration from a file. The default name of the server is netstoraged so (if the –conffile command line argument is not provided) the default configuration file name will be netstoraged.ini.

The configuration file uses the NCBI 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 |
| max\_connections | Maximum number of simultaneously opened connections.  Default: 500 |
| max\_threads | Maximum number of threads for processing client requests.  Default: 50 |
| init\_threads | Initial number of threads for processing client requests.  Default: 10 |
| port | TCP/IP port on which the server expects incoming connections.  The recommended port range is 9800-9809.  Default: None. This is a mandatory field and the server will not start if the port is not provided. |
| 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 | The flag to switch on/off logging the server activity.  Default: true |
| log\_timing | The flag to switch on/off logging timing information for the commands. Switching to true will have effect only if log == true.  Default: false  Introduced in NST 2.0.1  Dropped in NST 2.2.0 |
| admin\_client\_name | A list of client names which can execute commands requiring administrative privileges, e.g. SHUTDOWN. The separators for the client names are: semicolon, comma or space characters.  Default: empty list which means that nobody will be able to execute administrative commands. |
| log\_timing\_nst\_api | Switching on/off collecting timing for the NetStorage API operations.  Default: false  Switching to true will have effect only if log\_timing == true and log == true  Introduced in NST 2.1.0 |
| log\_timing\_client\_socket | Switching on/off collecting timing for the client socket operations.  Default: false  Switching to true will have effect only if log\_timing == true and log == true  Introduced in NST 2.1.0 |
| data\_path | Path to a directory where private data are stored (e.g. start-after-crash signaling file). If the directory does not exist then the server will create it.  Default: ./data.<port>  Introduced in NST 2.2.0 |

## [log] section

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

## [metadata\_conf] section

The section specifies the services for which metadata information should be supported and some service objects expiration default parameters.

Note: when the service name specified by the client in the HELLO command is found among the services configured in this section then the metadata information facilities might be provided for the objects used in this HELLO session.

|  |  |
| --- | --- |
| Value | Description |
| services | String, default is empty string.  List of services separated by one or more of: " \t\r\n\v\f,"  Obsolete in NST 2.2.0 and above. The NST 2.2.0 walks over all the sections which names start with "service\_" and forms a list of services if those sections have the "metadata" parameter set to true. |
| Ttl | String, default INFINITY  The object TTL upon creation. Supported format: $dd $hh $mm $ss. A special value (case insensitive) “infinity” is supported. If “infinity” is provided then the object will never expired. |
| prolong\_on\_read | String, default 0s  Defines how the object time to live should be prolong upon reading.  See the supported formats in the table below. |
| prolong\_on\_write | String, default 0s  Defines how the object time to live should be prolong upon writing.  See the supported formats in the table below. |
| prolong\_on\_relocate | String, default 0s  Defines how the object time to live should be prolong upon relocating.  See the supported formats in the table below.  Note: introduced in NST v.2.2.0 |

|  |  |
| --- | --- |
| Prolong values format | Description |
| Timespan | $dd $hh $mm $ss  Timespan is specified as separate days, hours, minutes and seconds with appropriate suffixes, e.g.:  20d 14h 30m |
| Multiplied TTL | <double value> ttl  Timespan is specified as the TTL value multiplied to the provided double value, e.g.:  0.5 ttl  If the effective TTL value is 10 days then the calculated value is 5 days. The effective TTL is calculated as follows: if a specific TTL is set for an object (via SETEXPTIME message) then this value is used. Otherwise the service TTL is used.  Note: introduced in NST v.2.3.0 |

The “ttl”, “prolong\_on\_read”, “prolong\_on\_write” and “prolong\_on\_relocate” (only for NST v.2.20 and up) values define default values for all the configured services. Each service in turn may have its own section e.g. [service\_foo] for the “foo” service and this section may overwrite the corresponding values if necessary.

## [database] section

The parameters from this section may not be reconfigured at the run time via RECONFIGURE command.

|  |  |
| --- | --- |
| Value | Description |
| service | Database service name or  <database server>:<database server port> |
| user\_name | User name |
| password | User password |
| database | Database name |
| execute\_sp\_timeout | Timeout for each individual stored procedure to be executed.  Default: 20.0 (float, in seconds)  Introduced in NST 2.2.0 |

NB:

The FileTrack configuration is hardcoded and can be moved to the configuration file eventually.

The NetCache configuration section is standard for the already developed client API. It is read automatically by the client code and thus is not described in this document.