Status Server Detailed Design

Tom Vermeulen

28 May 2002

This document is available on the Web at: http://software.cfht.hawaii.edu/sserver/detail_design/

Contents

List of Figures

1 Introduction

1.1 Purpose

The purpose of this document is to define the detailed design specification for the Status Server. The detailed design specification documented here is based on a set of previously defined requirements and functional specification. The implementation of the Status Server will be based on the design outlined in this document. While the Client API is mentioned in some detail, this document does not adequately cover the detailed design of the Client API. Either this document will be amended to fully cover the Client API, or the detailed design of the Client API will be covered in a separate document.

The first draft of this document must be reviewed by the members of the software group and will be amended following review. The implementation stage will not start until this review has been completed and the document updated.

1.2 Scope

Unless otherwise noted, the detailed design specifications identified in this document are intended to be implemented in the first release of the Status Server. However, release requirements may dictate the priority and staging of functionality.

1.3 References

The design of the Status Server is based on a previously established set of requirements and functional specification. In addition, the Status Server will utilize a previously written CFHT Socket Library, which is often referred to as “sockio” in this document. More details regarding the requirements, functional specification, and sockio library can be found at the following locations on the CFHT intranet.

- Status Server Requirements Document - http://software.cfht.hawaii.edu/sserver/requirements/
- Status Server Functional Specification - http://software.cfht.hawaii.edu/sserver/func_spec/
- CFHT Socket I/O Library - http://software.cfht.hawaii.edu/sockio/

2 Development and Execution Environment

Both the Status Server and Client API library will be developed using the C language and conform to the established CFHT coding standard. The software will be compiled for the three major UNIX architectures in use at CFHT; HP-UX, Sun Solaris, and Linux. The Client API library will be available on the three previously mentioned architectures.

2.1 Hardware

The amount of processing power required to run the Status Server is largely dependent on the load placed upon it by its client connections. Based on the previously defined requirement that clients not update data at frequencies greater than 1 Hz, a moderately configured system should be more than sufficient to run the Status Server. Benchmarking must be performed following implementation to determine what the typical memory and CPU utilization for the Status Server will be.

The machine running the Status Server must have a 100Mbps Ethernet connection and be accessible by other machines on the CFHT network.

2.2 Software

Clients wishing to use the Client C-API must be running either HP-UX, Sun Solaris, VxWorks, or Linux. The C-API library will be compiled and linked as a static library target.

3 Status Server System Overview

A visual representation of the high-level components involved in the Status Server System can be found in figure 2. This document focuses on the Status Server and Client C API components and the message protocol used between client and server. It is possible for users to connect to the Status Server via a telnet session, using the same message protocol used by the client C API.

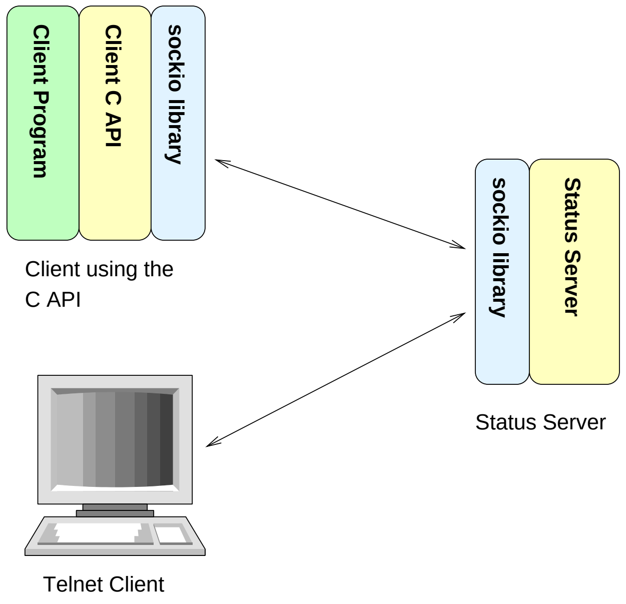

Figure 1: Status Server High-Level System Diagram

3.1 Client ⇔ Server Communication

3.1.1 Overview

The Status Server will listen over a socket interface to client requests. The server will service each request and send back an associated response. With the exception of a disconnect request, each client request will receive a response from the Status Server. In most cases, the client will receive a single line response to a request. The exception to the single response model is the case where a client has requested monitoring updates or the client has requested the contents of a directory. Multiple line response messages will always be terminated with an end-of-transaction (EOT) return message. The client must not send any new commands until it has fully processed the current command. If, for some reason, the server receives a new command request from a client before it has sent the client the last response, it may inform the client that a protocol error has occurred. At this point, the Status Server will expect the client to close the connection. If, however, the client sends another command, the Status Server will close the client connection.

In the case of monitored objects, it is possible for a client to receive an unsolicited message across the interface. This message is triggered the first time a client-monitored object is updated beyond the “deadband” restriction and the client has not already been informed of a monitor update. Once a client is informed that it has monitored information to retrieve, it must initiate a “poll” request to retrieve the information. This is an event-driven model which triggers the client to always initiate a retrieval of monitor information. This document does not address the handshaking implementation concerning how a client using the Client API will be notified that monitors are available and the triggering mechanism to retrieve them. This will be defined in the detailed design of the Client API.

The Status Server will utilize the sockio library to handle the low-level socket details. The sockio library uses a single-threaded non-blocking approach to handling client connections. The interaction between the sockio library functions and the Status Server will be discussed in more detail in the software design section of this document. In addition, you can review the the CFHT Socket I/O Library document for more information regarding the design of this library.

Both the Status Server and sockio library are designed in such a way that any data sent across the socket can be gracefully handled. This includes receiving binary data or unusually long messages which may or may not be properly terminated with a newline character. If a client attempts to connect from outside the CFHT network, or a client violates the established message protocol, whenever possible its connection will be terminated.

Each request received by the Status Server will be checked to make sure it is both a valid command and does not contain any invalid characters. The Status Server will only process requests which contain URL encoded 7 bit ASCII printable characters terminated with a newline (CR/LF or LF). If a non-conforming request is received, it will be rejected with a “syntax error” response. In the Status Server encoding scheme, only printable characters with the exception of some special characters can be sent unencoded. Figure 2 shows the characters which must be explicitly encoded prior to being received by the Status Server.

Character	**ASCII Value (Hex)**	**Reason for Encoding**
Percent Sign (’%’)	25	Used to URL encode/escape other characters, so it should itself also be encoded.
Single Quote (”’)	27	Used as a wrapper around distinct fields of data. The parser will treat data within a single quotes as one field.
Double Quote (’”’)	22	Used as a wrapper around distinct fields of data. The parser will treat data within double quotes as one field.
Control Characters	< 20	Must be encoded to prevent unpredictable behavior.
Extended Characters	$> 7E$	Must be encoded to prevent display issues via an interactive telnet session.

Figure 2: Characters Which must be URL Encoded

URL encoding of a character consists of a “%” symbol, followed by the two-digit hexadecimal representation (case-insensitive) of the character value. For example, a tab character would be encoded as “%09”.

Since the Status Server does not perform any encoding or decoding functionality, functions will be available in the Client API to perform encoding and decoding of Strings from an 8 bit character format to URL encoded format. Clients which decide to access the Status Server via a telnet session or custom socket implementation, must be aware of the URL encoding requirements of the Status Server and perform the necessary encoding.

It is important to note that any encoding schemes used to encapsulate data are completely hidden from clients using the Client API. A client using the Client API does not need to call any encoding or decoding functions.

All string data stored and manipulated within the Status Server is 7 bit only.

3.1.2 Directory Structure

Objects within the Status Server are grouped together in a tree-like fashion patterned after the UNIX file system. As a result, it will be possible for a client to traverse and manipulate objects within the Status Server much like traversing a directory tree and manipulating files in a file system. Objects within the Status Server can be referenced either via a fully qualified directory path/object name combination, or a relative path-name combination. In order to manage relative path references, a current path will be maintained for each client connection. Rules to determine whether a path-name combination is expressed as an absolute path or relative path will be applied the same way they are in a UNIX file system. A visual example of the type of structure used to hold Status Server information is shown in figure 3.

```
# Status Server database
# ---------------------
#
# Format is /path/and/varname =  # Description in comment#
# Formatting of <sample value> indicates "native" type for this field.
# This is handled internally by the routine that serializes the database.
#
# TRUE/FALSE     - Booleans will have *un-quoted* TRUE or FALSE as the value.
# "string"       - Strings always saved with " as the first char in value field.
# 10.            - Float values will always show a decimal point (even if .0)
# 15             - Numeric values without a decimal indicate integers.
#
# Top-level "directories":
#
# /i/ with a subdirectory for each instrument (often same names as handlers)
# /t/ with subdirectories for each telescope subsystem
# /p/ for plant environment, weather, data-logger variables
# /f/ has subdirectories for each exposure where FITS headers are accumulated
#
/i/                                        # Instruments
/i/megacam/                                # Megacam agent stuff
/i/megacam/etime       = 10.               # Current exposure time
/i/megacam/etype       = "BIAS"            # Current exposure type
/i/megacam/filter      = 0                 # Current filter position
#
# /i/cfh12k/ are all generated by 12kcom(detcom) and used to be in .,12kcom.par
#
/i/cfh12k/status       = "Idling"          # Camera status for GUI
/i/cfh12k/raster       = "FULL"            # Current raster setting
/i/cfh12k/etime        = 10.               # Current exposure time
/i/cfh12k/etype        = "FLAT"            # Current exposure type
/i/cfh12k/filter       = 0                 # Current filter position
/i/cfh12k/filter[0]    = "R"               # Desc. of filter in slot 0
/i/cfh12k/filter[1]    = "V"               # Desc. of filter in slot 1
/i/cfh12k/filter[2]    = "B"               # Desc. of filter in slot 2
/i/cfh12k/filter[3]    = "I"               # Desc. of filter in slot 3
/i/cfh12k/observer     = "Galileo"         # Current OBSERVER header
/i/cfh12k/object       = "TF dawn"         # Current OBJECT header
/i/cfh12k/comment      = "Twilight flats"  # Current CMMTOBS header
/i/cfh12k/piname       = "Mellier"         # Current PINAME header
/i/cfh12k/runid        = "99IIF142"        # Current RUNID header
```

Figure 3: Status Server Hierarchical Name-Value Pair Representation

3.1.3 Directory and Data Object Information

Each directory and data object in the Status Server consists of a series of attributes. These attributes include:

1. **Name** - Within the Status Server the object name, in combination with its associated directory path location, must be unique. The name and directory path must consist of a string of 7 bit ASCII printable characters. In addition, the following series of characters will not be permitted in a directory path or object name (“, ’, =, space).

2. **Value** - The value stored with the object. The value of an object stored in the Status Server will always be enclosed within double quotes if it is valid. If the value of the object is not valid, it will not be enclosed within double quotes. For example, the following values would be considered valid: “data”, “0.0”, or “sample data”. If a value is not enclosed within double quotes, it must always be one of the following values.

   (a) **NONEXISTENT** - If a request is received by the Status Server to initiate a monitor on a data object which does exist, a data object will be created, and its value will be set to NONEXISTENT. For a client perspective, any request made to update or retrieve the value of this object will fail with an indication that the object does not exist, until the object is created with a “touch” command request (see section 4.1.3).
   (b) **UNDEFINED** - If a client has created an object, but has not assigned a value. This would be the initial value of an object following a “touch” command request (see section 4.1.3).
   (c) **EXPIRED** - If the object has not been updated within a “lifetime” length of time since its last update.

3. **Comment** - An entry describing what the object is.
4. **Lifetime** - Indicates the maximum amount of time this object can be considered valid. As an example, the current seeing may only be defined to be valid for an hour.

If a data object has a value of NONEXISTENT, it will be completely removed and deallocated whenever its use counts are zero. This means that a data object can not be completely removed if a client has performed a touch, monitor, or directory listing request on the object. This is a requirement to enforce pointer integrity within the Status Server.

More details regarding the attributes associated with a directory or data object can be found in the software design section (see 6.1.2).

3.1.4 Message Flows between Client and Server

In order to understand the communication between client and server, a set of message flows is included to illustrate normal operational flows and error conditions. In each flow, solid lines indicate Client API or Status Server initiated messages while dotted lines indicate low-level socket messages.

- **Client Connection** - Figure 4 illustrates the normal sequence of messages used when a client connects to the server. In the case of a Client API connection, the Process ID (PID) and Program Name will be sent to the Status Server in a message following the initial connection. The Client API should always receive a positive response to this message. If not, this will be considered an internal error and the Client API will log debug information and exit.

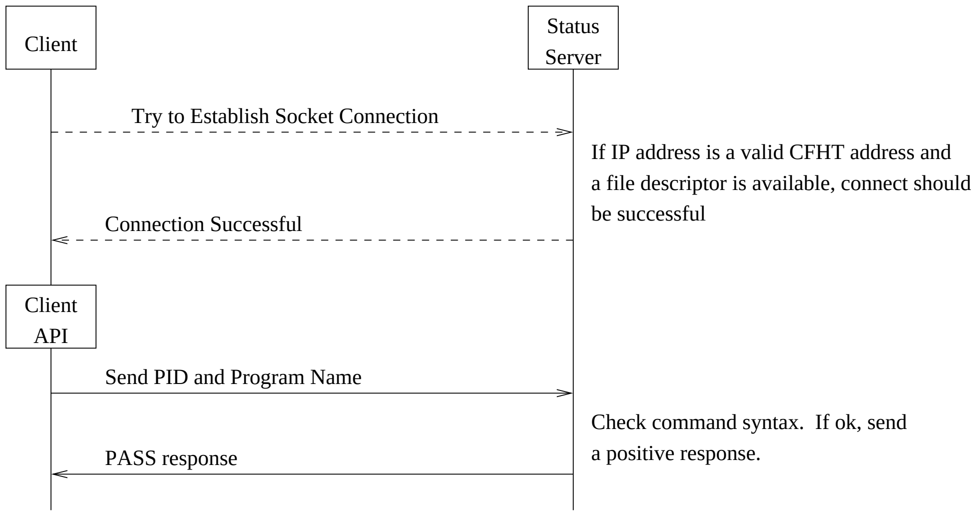
Client
Status Server
Try to Establish Socket Connection
If IP address is a valid CFHT address and a file descriptor is available, connect should be successful
Connection Successful
Client API
Send PID and Program Name
Check command syntax. If ok, send a positive response.
PASS response

Figure 4: Client Connection Message Flow

- **Client Disconnection** - Figure 5 illustrates the normal sequence of messages when a telnet client disconnects from the server. It assumes that the client sent a message to disconnect from the server. It is also possible for the client to close its end of the socket connection in order to trigger a disconnect. The Client API will close the socket connection instead of sending a message to the Status Server. It will always be safe to allow a client program using the Client API to exit without explicitly triggering an ssLogoff() function call. Once the program exits, the socket should automatically be closed.

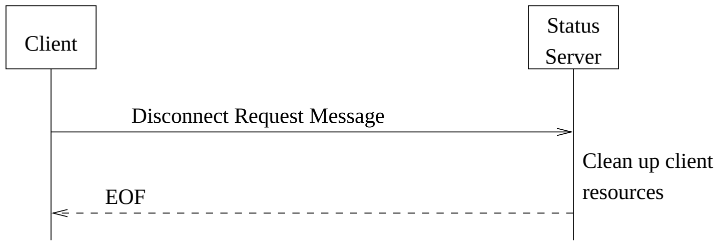

Figure 5: Client Disconnect Message Flow

- **Command Request** - Figure 6 illustrates the normal message sequence when a client sends a request. This will be the standard request-response flow for almost all the commands with the exception of the “getdir”, “poll”, “logoff”, and “shutdown” commands. The response from the Status Server will indicate whether the command was processed successfully and may contain data if the response is valid. If the command failed, the response will indicate the reason for the failure. It is possible for the client to receive a mailbox message indicating that objects monitored by the client have changed prior to receiving the command response. The Client API must handle this situation properly.

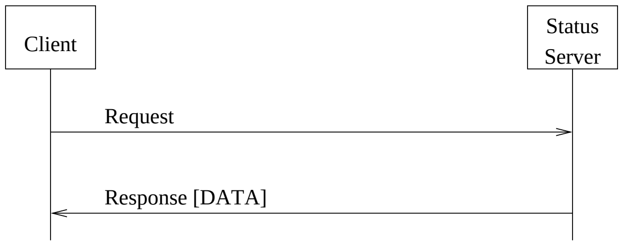

Figure 6: Client Command Request Message Flow

- **Server Shutdown Request** - Figure 7 illustrates the message sequence when a client requests a shutdown of the Status Server. Once the Status Server has saved the directory structure to disk, it will clean up its internal resources and exit. This process will cause each client connected to the Status Server to receive an indication that its socket connection to the Status Server has been closed by the server.

- **Client Out-of-Sequence Request** - Figure 8 illustrates a sample message sequence if a client initiates an out-of-sequence message. The client must always wait for the response to a previous message before sending a new message. For clients using the Client API, the API will take care of waiting before a new message is sent. If a client violates this protocol, the server will send a message to the client indicating that a protocol violation occurred. The client should then close the socket connection. If the client does not close the socket connection, the Status Server will close the client connection upon receiving another request.

- **Server Out-of-Sequence Response** - Figure 9 illustrates a sample message sequence if a client receives a response from the Status Server which it believes is out-of-sequence. On the client side, this would be considered a protocol violation by the Status Server. If the Client API encounters this situation, it will send a “protocol error” message to the Status Server. The Status Server will respond by logging error information to the CFHT log and closing the client connection. The Client API must be able to handle a “mailbox” message at almost any

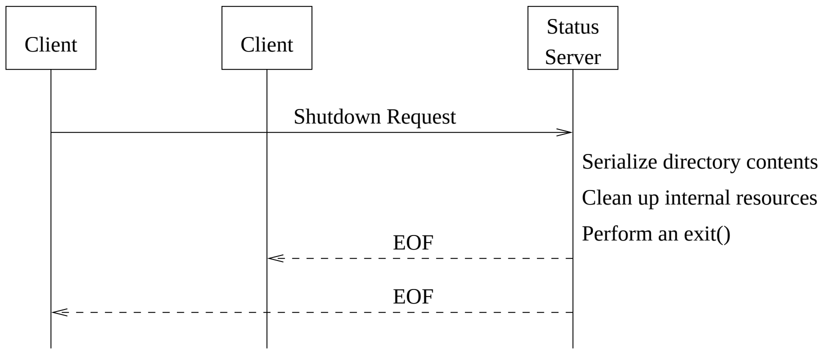

Figure 7: Client Shutdown Request Message Flow

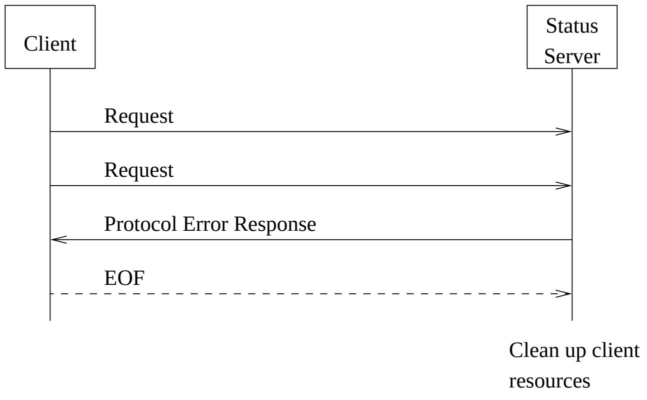

Figure 8: Out-of-Sequence Client Request Message Flow

time, so this will not be considered a protocol violation. The exception is if the Server sends multiple “mailbox” messages before the client sends a “poll” request.

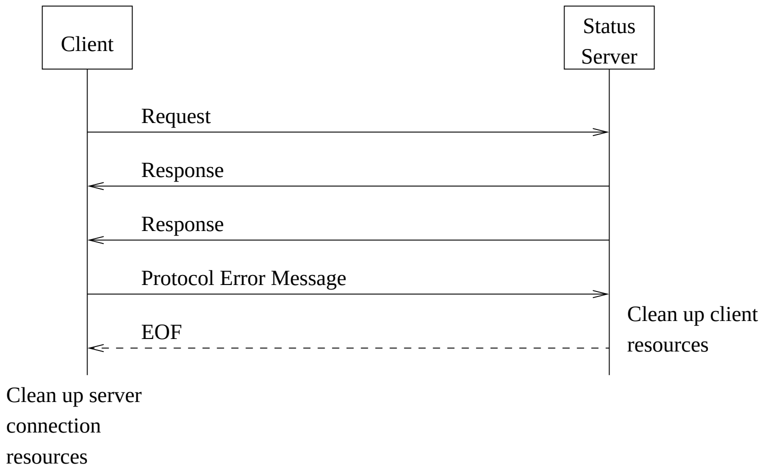

Figure 9: Out-of-Sequence Server Response Message Flow

- **Directory Listing Request** - Figure 10 illustrates the message sequence when a client sends a valid directory listing request. The server will send the directory header followed by the set of directories and objects. Once all the directory contents have been sent, the Status Server will send an end-of-transaction (EOT) message.

If the directory is empty, the directory header will be sent followed by an immediate EOT message. If the client requests a directory which does not exist, or if the command syntax is invalid, the Status Server will send a descriptive error message instead of sending the directory header. The Status Server will not accept any additional requests from the client until after the EOT message has been sent. If this situation occurs, and is detected by the Status Server, it will send a protocol error message to the client.

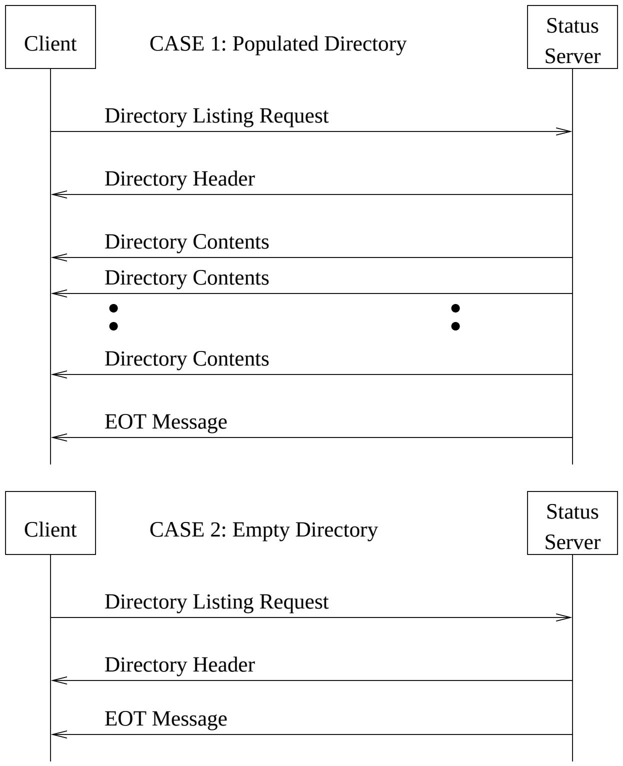

Figure 10: Directory Listing Request Message Flow

- **Complete Monitoring Flow** - Figure 11 illustrates the complete set of messages exchanged in a typical monitoring flow. Once the Status Server receives the monitor retrieval request, it will reset its flag indicating that a mailbox has previously been sent to the client. As a result, subsequent changes to client monitored objects may trigger a new mailbox message to be sent to the client. An example of this situation is illustrated in figure 12. It is possible that the latest monitor update will be sent to the client in the current monitor retrieval already in progress. If so, when the client initiates another monitor retrieval request the Status Server may immediately return an EOT message. The Status Server will not accept any additional requests from the client until after the end-of-transaction message has been sent. If this situation occurs, and is detected by the Status Server, it will send a protocol error message to the client.

4 Command Protocol

The previous section illustrated the high-level message passing protocol between client and server. This section addresses the detailed syntax and format of the individual messages. As previously mentioned, each and every client request, with the exception of the “logoff” and “shutdown” request, will receive a response. The server will only handle single line URL encoded messages terminated with a newline (“\r\n” or “\n”). If a request is received, which is not properly URL encoded, it will be rejected.

The first character of a Server Response indicates the type of response sent by Status Server. There are five different responses which the Status Server can send to the Client. Each of these responses is summarized in the following list:

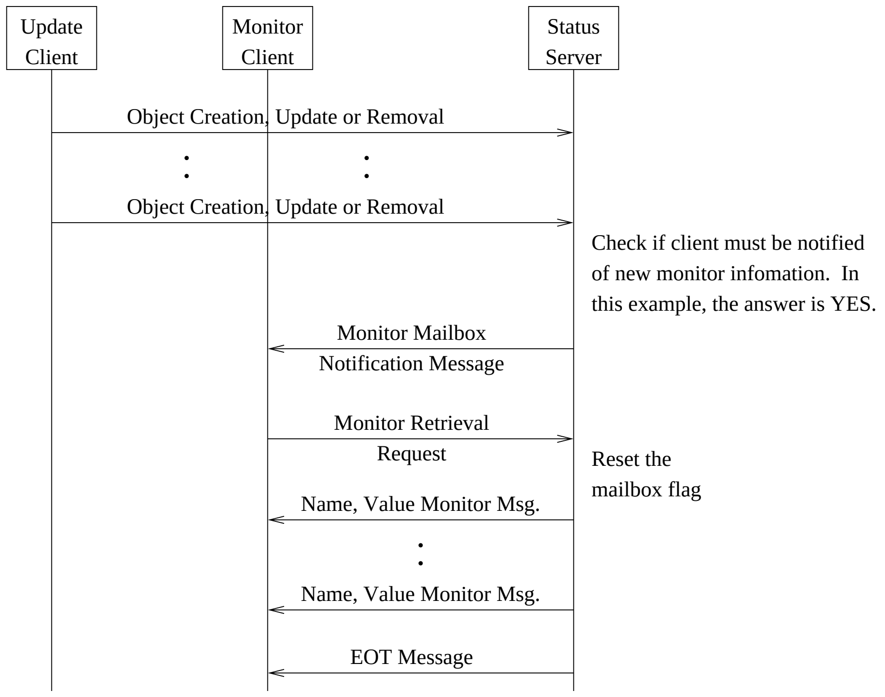
Update Client
Monitor Client
Status Server
Object Creation, Update or Removal
Object Creation, Update or Removal
Check if client must be notified of new monitor infomation. In this example, the answer is YES.
Monitor Mailbox Notification Message
Monitor Retrieval Request
Reset the mailbox flag
Name, Value Monitor Msg.
Name, Value Monitor Msg.
EOT Message

Figure 11: Typical Monitoring Message Flow

- **Command Passed ('.')** - This response indicates that the request initiated by the Client was successfully processed. The End of Transmission (EOT) message will always be preceded by a '.'.
- **Multiple Line Response ('+')** - This response indicates that it is part of a multiple line response message from the Status Server and at least one more line will be sent by the Status Server. The final message for a multi-line response, such as a directory listing or retrieval of monitoring information, will always be ". EOT".
- **Command Failed ('!')** - This response indicates that the Status Server was not able to process the client request. Details regarding the error will follow the explanation point. A sample error response is "! syntax error".
- **Protocol Error ('?')** - This response indicates that the Status Server was not able to process the client request because a protocol error occurred. The client will receive the following message "? protocol error".
- **Mailbox ('*')** - This response indicates that monitored information is available for retrieval. As mentioned earlier, this response may be received at any time by the client. The full mailbox message will be sent as "* MAIL".

The first character encoding scheme is defined in such a way that it is not specific to any particular command request sent across the interface from the client.

4.1 Client Command Syntax

This section contains the full set of commands, the required syntax of each command, and the response a client should expect to see. Optional arguments are included in square brackets "[]". The command syntax is based on a "mixed positional/keyword" system. In the case of mandatory arguments, they can be provided either as space delimited fields or as a fully qualified argument. As an example, the "REGISTER" command can either be specified as REGISTER 124 "foo" or as REGISTER PID=124 NAME="foo". Optional arguments must always be specified in a fully qualified format. Command and argument specifiers are case insensitive. For example, the "REGISTER" command could be

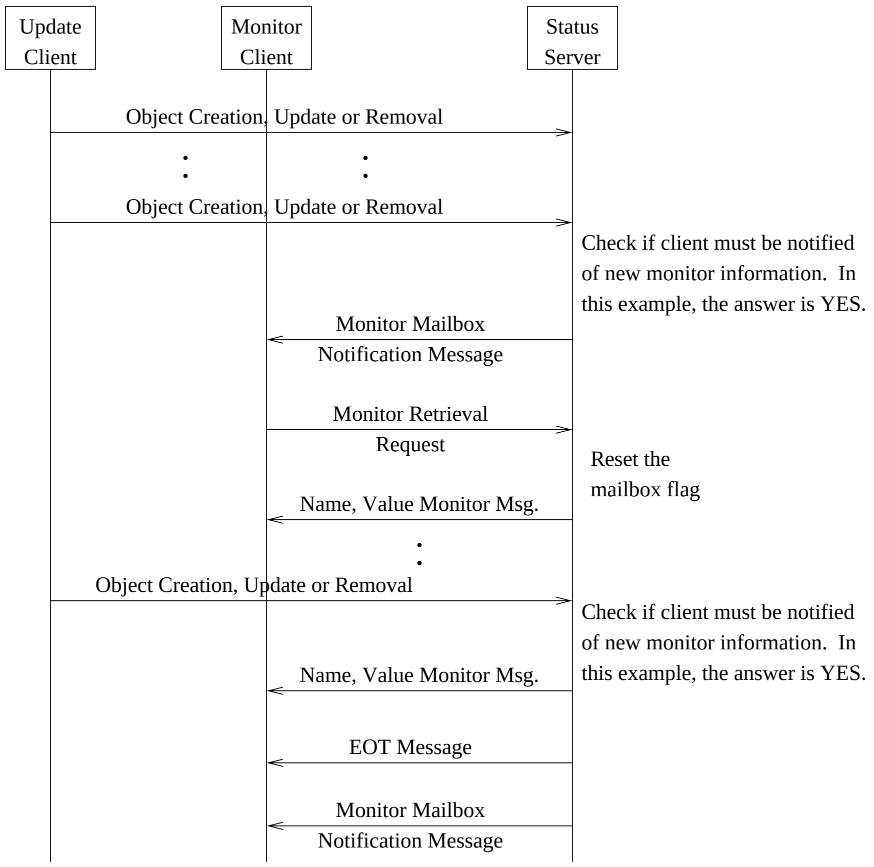
Update Client
Monitor Client
Status Server
Object Creation, Update or Removal
Object Creation, Update or Removal
Check if client must be notified of new monitor information. In this example, the answer is YES.
Monitor Mailbox Notification Message
Monitor Retrieval Request
Reset the mailbox flag
Name, Value Monitor Msg.
Object Creation, Update or Removal
Check if client must be notified of new monitor information. In this example, the answer is YES.
Name, Value Monitor Msg.
EOT Message
Monitor Mailbox Notification Message

Figure 12: Monitoring Message Flow with Midstream Update

sent in either upper or lower case. For clarity purposes, command and argument specifiers are shown in upper case with the arguments in lower case within this document.

If a string argument contains embedded spaces, it must be enclosed within either single or double quotes indicating that the spaces are part of the argument string. Without the enclosing quotes, the parser will interpret the string as separate arguments. Numeric parameters should not be enclosed within quotes. If the Status Server receives a numeric parameter enclosed within quotes, it will strip off the quotes before converting the string to a number.

The Client API will always send commands with fully qualified arguments and place quotes around each string parameter sent to the Status Server.

It is important to note here that the value of an object stored in the Status Server will always be enclosed within double quotes if it is valid. If the value of the object is not valid, it will not be enclosed within double quotes. For example, the following values would be considered valid: “data”, “0.0”, or “sample data”. If a value is not enclosed within double quotes, it must always be one of the following values: NONEXISTENT, UNDEFINED, or EXPIRED. As a result of this encoding scheme, it is possible for the Status Server to send back a response to the client which seems to indicate that the command passed, but the Status Server value is actually invalid. It will be the responsibility of the Client API to handle these as conditions and supply the appropriate error information available to the client.

All responses that return a data object name or directory name will always return this value in its fully-qualified absolute path format. The exception is directory listings where the absolute path of the directory will be included as part of the header line and each data object name within the directory will be expressed in relative path format.

In some cases, the response from the Status Server contains the command and in other cases it does not. Since a client can only send one command request at a time, it is not necessary to encode the command request within the response. Instead, the command responses have been set up in such a way that it could be ossible to set up a cacheing proxy server to relay Status Server information. In order to enable this, wherever possible a command/value response is returned with the object name and object value. A proxy server could store a local copy of the values sent to and from the Status Server in order to reduce the load on the Status Server. There are no plans currently in place to deploy such a server, but the message protocol has been implemented in such a way that this could be a possibility in the future.

It is possible for the Status Server to supply additional information following some of the documented command responses. While the additional information may be ignored by the Client API, it could be useful for interactive use. For example, the Status Server may include the number of items returned for a directory listing request following the end-of-transaction (EOT) message.

4.1.1 Register Client with the Status Server (OPTIONAL)

Once a socket connection is established between the Client and Status Server, additional details describing the client can be sent to the Status Server. This registration message is optional, but will always be sent as part of the Client API initialization. Typically, a user interacting with the Status Server via a telnet session will not use this command. The message flow for this command can be found in figure 4.

```
REGISTER [PID=]pid [NAME=]client_name
```

Input parameters for the registration command are as follows:

- **pid** - Process ID (PID) of the client process interacting with the Status Server. This will be the PID of the process invoking the Client C-API.
- **client_name** - Program name. This is the same as argv[0].

Response the client would expect to see:

- . welcome client_name
- ! syntax error

In the case of the Client C-API, receiving a syntax error should never occur, and would be considered an internal error.

4.1.2 Disconnect from the Status Server

The connection between client and Status Server will remain persistent until the client chooses to disconnect, or the network connection between client and server is broken. When using a telnet connection, it is sometimes easier to type a short command than the CTRL-']' quit sequence. The Status Server will have a command available to enable a client to terminate the connection. This command will cause the client to receive an EOF across the interface indicating that the Status Server has closed the connection. The message flow for this command can be found in figure 5.

```
QUIT
```

This command does not have any input parameters and will not generate a message response.

4.1.3 Create an object or register the intent to modify an object or directory

Prior to either updating or removing an object within the Status Server, the client must perform a "touch" on the object. The touch enables a client to specify the intent to update or modify a value in the Status Server. The message flow for this command can be found in figure 6.

```
TOUCH [NAME=]obj_name [COMMENT=obj_comment] [LIFETIME=obj_lifetime]
```

Input parameters for the registration command are as follows:

- **obj_name** - Name of the object. The object name can be specified either as a name containing the fully qualified absolute directory path or as a name containing a relative directory path. Absolute references will be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". The object name may not have a trailing '/' character, since this is used to indicate a directory.
- **obj_comment** (OPTIONAL) - A description of the object.
- **obj_lifetime** (OPTIONAL) - An integer number expressed in seconds, which identifies the amount of time the object will be considered valid following a modification.

Response the client would expect to see:

- . obj_name TOUCHED (If the object did not previously exist, the value will be "UNDEFINED")
- ! syntax error

In the case of the Client C-API, receiving a syntax error should never occur and would be considered an internal error.

4.1.4 Update an object

Once a successful touch has been performed on an object, it is possible for the client to initiate an update request. The update request causes the value associated with the object to be modified. The message flow for this command can be found in figure 6.

```
PUT [NAME=]obj_name [VALUE=]obj_value
```

Input parameters for the update command are as follows:

- **obj_name** - Name of the object. The object name can be specified either as a name containing the fully qualified absolute directory path or as a name containing a relative directory path.
- **obj_value** - New value for the object.

The client will receive one of the following responses:

- . obj_name obj_value (The returned obj_value will always be enclosed within double quotes, since it should be valid in this case).
- ! syntax error
- ! object does not exist
- ! permission denied (Would occur if the client did not previously perform a touch for the object)

4.1.5 Retrieve an object or retrieve the status of an object

Since the Status Server is an open repository without permissions, any client has the ability to retrieve an object from the Status Server. This command will return the value of an object in the Status Server if it is valid. If not, the reason it is not valid will be returned. Valid values will always be encapsulated within double quotes. The message flow for this command can be found in figure 6.

```
GET [NAME=]obj_name
```

Input parameters for the retrieval command are as follows:

- **obj_name** - Name of the object. The object name can be specified either as a name containing the fully qualified absolute directory path or as a name containing a relative directory path.

The client will receive one of the following responses:

- . obj_name obj_value (Always enclosed within double quotes)
- . obj_name EXPIRED (If the object is expired)
- . obj_name UNDEFINED (If the object has not been initialized)
- ! object does not exist
- ! syntax error

4.1.6 Initiate a monitor on a directory or data object

Clients can initiate monitors on Status Server directory or data objects in order to be informed whenever the value of an object changes. To reduce the load on both the Status Server and client, a client also has the opportunity to specify a “deadband” range for both floating point and integer objects. The message flow for this command can be found in figure 6.

It is not necessary for the directory or data object to exist in the Status Server in order to place a monitor on it. In this case, the monitor will be applied by the Status Server once the object is created. The client does not need to perform a touch on the object before placing a monitor on the object. The ability to place a monitor on an object, which does not yet exist, is available to prevent potential race conditions during client start up.

A monitor placed on a directory object will cause the client to be notified when the contents of a directory change. If the Status Server contained a directory with individual data objects for each filter currently installed in the CFH12K instrument, it could be possible for a client to place a monitor on this directory. As a result, a dynamic drop-down list could be generated with the current filter options. It is important to note that when a client is notified that a directory object has changed, the client must retrieve the contents of the directory via an “ls” command.

```
MONITOR [NAME=]obj_name [DB=deadband_val]
```

Input parameters for the registration command are as follows:

- **obj_name** - Name of the directory or data object. The object name can be specified either as a name containing the fully qualified absolute directory path or as a name containing a relative directory path.

- **deadband_val** (OPTIONAL) - This defines the offset which a data object value must exceed before the Status Server will inform the client of a change in value. The server will suppress updates which are within "deadband" units of the value last sent to the client. For example, if the client specifies a deadband of 2.5 and the value of an object changes from 1 to 3, the client would not be informed of a change, since the value changed by 2 units and the deadband is 2.5 units. This parameter must be specified as a positive number. If a deadband parameter of 0 is specified, this is the same as not providing the deadband argument. In this case, a deadband limit is not used. If the deadband parameter is not numeric, or is a negative number, this will be considered a syntax error.

The client will receive one of the following responses:

- . obj_name MONITORED
- ! syntax error

In the case of the Client C-API, receiving a syntax error should never occur and would be considered an internal error.

4.1.7 Remove a monitor from a directory or data object

Any time a monitor is added to an object in the Status Server, it can be removed by initiating a removal request. Monitors are not automatically removed when an object is removed from the Status Server. The message flow for this command can be found in figure 6.

```
UNMONITOR [NAME=]obj_name
```

Input parameters for the retrieval command are as follows:

- **obj_name** - Name of the object. The object name can be specified either as a name containing the fully qualified absolute directory path or as a name containing a relative directory path.

The client will receive one of the following responses:

- . obj_name UNMONITORED
- ! syntax error
- ! monitor does not exist

4.1.8 Retrieve monitor updates

The Status Server will send an "out-of-band" notification any time it has monitored information to send to the client. As mentioned earlier, this notification will be indicated by an asterisk ('*') in the first column. The full message will be "* MAIL". At this point, the client will most likely respond with a request to retrieve monitor information. The Status Server will then send all monitored information to the Client followed by an end-of-transmission (EOT) message indicating that all monitored information has been sent. The full EOT message will be ". EOT". It is possible for the Status Server to place additional information following the mailbox and EOT messages perhaps for a comment. While this will be unused initially by the client API, it may be useful to have. For example, when a directory listing is performed interactively it may be useful to have a comment indicating the number of objects returned. The message flow for this command can be found in figure 11.

```
POLL
```

This command does not have any input parameters.

The POLL command will generate individual responses for each monitored value which has been modified beyond the deadband threshold. Once all the individual monitor updates have been sent by the Status Server, the client will receive an end-of-transmission (EOT) message. It is also possible for the client to send a POLL command and not receive any response outside of the EOT. This could occur if a monitored value changed enough to warrant a monitor notification (mailbox message), but when the Status Server received a POLL from the client, the value was within the deadband threshold for the last value sent to the client. The possible responses the client should expect to see are as follows:

- + obj_name obj_value (New value of a valid object)
- + obj_name NONEXISTENT (If the object has been removed)
- + obj_name EXPIRED (If the object has expired)
- + obj_name UNDEFINED (If the object has not been initialized)
- . EOT (following the last updated value)
- ! nothing monitored by client (if the client requests a POLL, but the client does not have any monitors defined on objects).

The first character of the response indicates what the response means. If the first character is a '+', this indicates that the value of a monitored object has changed. If the obj_value is enclosed within double quotes, it is a valid value. If not, the state of the Status Server data object is now invalid. The ''+' character is also used to indicate that at least one more response will be sent by the Status Server for the POLL request. Once the client receives the ". EOT" message, it will know that the Status Server is done sending monitor updates. In addition, it is possible for the client to receive a single-line error response indicating that the client requested a POLL, but the Status Server does not have any record of monitored objects for the client.

If the Status Server should detect that the client sent a POLL command without having received a mailbox message, this will be considered a protocol error and the client will receive a protocol error response. At this point, any subsequent message received from the client will result in the Status Server terminating the client connection. The Client API will make sure that a POLL command is never sent to the Status Server without itself having received a mailbox message, so this should not be an issue with the Client API. This is added as a safeguard to prevent clients who are directly using the socket protocol from incorrectly implementing the protocol for monitor retrieval and possibly generating an excessive number of POLL requests.

4.1.9 Remove an object

Once a successful touch has been performed on an object, it is possible for the client to initiate a removal request of the object within the Status Server. The message flow for this command can be found in figure 6.

```
RM [NAME=]obj_name
```

Input parameters for the object removal command are as follows:

- **obj_name** - Name of the object. The object name can be specified either as a name containing the fully qualified absolute directory path or as a name containing a relative directory path.

The client will receive one of the following responses:

- . obj_name NONEXISTENT
- ! syntax error
- ! object does not exist
- ! permission denied (Would occur if the client did not previously perform a touch on the object)

4.1.10 Get the current directory path

Objects within the Status Server can be referenced whether via a fully qualified directory path/object name combination, or a relative path/name combination. In order to manage relative path references, the current path will be maintained by the Status Server for each client connection. This request enables the client to retrieve its current path. The message flow for this command can be found in figure 6.

```
PWD
```

This command does not have any input parameters.

There is really only one response which the client should expect to see as a result of this command.

- . PWD current_path (This will be represented as an absolute directory path)

The initial path for each client is '/'. This path will remain the default path until it is changed by the client via the CD command.

4.1.11 Change the current directory

This request will cause the Status Server to modify what it uses as the current directory for relative path references made by a client. The client can specify the new current directory with either a relative path or absolute path. The message flow for this command can be found in figure 6.

```
CD [PATH=]dir_path
```

Input parameters for the change directory path command are as follows:

- **dir_path** - New directory path to be used by the Status Server for relative path references made by this client. The directory path can be specified either as an absolute path or a relative path offset from the currently defined relative path.

The client will receive one of the following responses:

- . PWD dir_path (This will be represented as an absolute directory path)
- ! syntax error
- ! directory does not exist

4.1.12 Create directory or register intent to remove a directory

While required directories are automatically created as part of the touch command when objects are created, it is also possible to explicitly create a directory. This option will create a directory if it doesn't already exist.

This command must be used prior to removing a directory and all of its contents. When a touch is performed on a directory, it is possible to remove the directory and all the objects within it without performing an explicit touch on each object. The message flow for this command can be found in figure 6.

```
TOUCHDIR [DIR=]dir_path [COMMENT=dir_comment]
```

Input parameters for the directory touch command are as follows:

- **dir_path** - The directory path can be specified either as an absolute path or a relative path offset from the current directory path.
- **dir_comment** (OPTIONAL) - A description of the directory

The client will receive one of the following responses:

- . dir_path TOUCHED (This will be represented as an absolute directory path)
- ! syntax error

4.1.13 Remove a directory

It is possible to remove a directory and all its objects. In order to help prevent an inadvertent removal of a directory, a touchdir must be performed on the directory before it can be removed. In addition, the directory must not contain any subdirectories.

```
RM -R [NAME=]dir_path
```

Input parameters for the directory removal command are as follows:

- **dir_path** - The directory path can be specified either as an absolute path or a relative path offset from the current directory path.

The client will receive one of the following responses:

- . dir_path REMOVED
- ! syntax error
- ! directory not found
- ! directory contains subdirectories
- ! directory contains hidden objects
- ! permission denied

4.1.14 Retrieve the contents of a directory

Much like the “ls” command on the UNIX file system, it will be possible to retrieve the contents of a directory. This command may return more than one line as a response. The client will receive a first line indicating whether the command was successful followed by a sequence of responses with the contents of the directory. The last line sent by the server will indicate the end of the transaction. Objects and directories will be returned in an ascending ASCII sort order by name. In addition, this command will allow for the same regular expression matching rules used by the UNIX “ls” command. The message flow for this command can be found in figure 10.

```
LS [DIR=]dir_path [-l]
```

Input parameters for the directory retrieval command are as follows:

- **dir_path** - Target of the directory listing. The target can be specified either as a name containing the fully qualified absolute directory path or as a name containing a relative directory path. In addition, the target can contain a regular expression to reduce the list of contents returned. An example would be “LS /fits/633333o/a*”.

- **-l** (OPTIONAL) - This parameter will indicate that the directory listing will return the update time, expiration time, and comment as well in addition to the default fields.

The LS commend will generates a sequence of replies. Upon response to the initial LS request, the Status Server will send one of the following responses:

- + directory target information (Header line echoing back the listing request by the user)
- ! syntax error
- ! directory does not exist

If the client received a positive response to the listing request, the Status Server will generate individual responses for each object or subdirectory which matches the query request. Subdirectories can be differentiated from objects by the trailing '/'. Once all the contents of the directory have been sent by the Status Server, the client will receive an end-of-transmission (EOT) message. The possible responses the client should expect following a successful LS request are:

- + {object name or directory name} {value}
- + {object name or directory name} {value} {update time} {expiration time} {comment} (This would only be returned if the user chooses the "-l" option)
- . EOT

In the case of a full listing, the Status Server will attempt to format it in a readable format, so each category is left aligned in a column. Strings representing the update time and expiration time will be expressed using the dd-mmm-yyyy h24:mm:ss format. A directory will be considered not to have a value and will be represented with the string "DIRECTORY". Data objects which have a valid value will have the returned value enclosed within double quotes. Invalid values will be expressed without quotes and can be either UNDEFINED or EXPIRED. Any data objects which were created, but have a value of NONEXISTENT will not be returned as part of the directory listing.

4.1.15 Initiate a trace

For diagnostic purposes, it may be important to have a more detailed view of what is happening within the Status Server. This may help solve an issue with the way the Status Server is working or help diagnose a misbehaving client.

Trace information will be stored as debug information within the CFHT log. As a result, it will be possible to associate activity within the Status Server with external events to help identify and narrow down problems. The message flow for this command can be found in figure 6.

```
TRACE ON
```

This command does not have any input parameters

The client should receive the following response:

- . TRACE ON

4.1.16 Stop a trace

Once a trace is initiated, a client request must be performed to stop it. The message flow for this command can be found in figure 6.

```
TRACE OFF
```

This command does not have any input parameters

The client should receive the following response:

- . TRACE OFF

4.1.17 Serialize Status Server data to a file

The Status Server will serialize a copy of itself to disk any time it receives a message from a client, or if the 10 minute interval has expired. The only information which will be saved is the data associated with the structure used to hold the Status Server objects. It is important to note that information associated with client connections will not be saved as part of the serialization process. This is because all client connections will be lost as part of a Status Server restart and restore operation. The message flow for this command can be found in figure 6.

```
AUTOSAVE
```

This command does not have any input parameters

The client should receive the following response:

- .AUTOSAVE INITIATED

The Status Server will send a response prior to the initiation of the forked process to perform the serialization. This is done in order to limit the complexity required to send a true return value as a result of the serialization operation in the forked process. If the serialization should fail, details regarding the failure should be available in the CFHT log. As a result, if a client actively chooses to initiate a serialization request, he/she should check to make sure the serialization file was successfully written.

4.1.18 Shutdown the Status Server

For maintenance reasons, it may be necessary to shutdown the Status Server. In order to preserve the current state of information within the server, a copy of the Status Server information will be serialized to disk before an exit is performed. The message flow for this command can be found in figure 7

```
SHUTDOWN
```

This command does not have any input parameters and will not generate a response outside of the EOF sent across the interface indicating that the socket connection was closed by the server.

4.1.19 Protocol Error

At some point, the client may believe that the Status Server has committed a protocol error violation. If this is detected by the Client API, it will send a “PROTOCOL ERROR” message to the Status Server. Once the Status Server receives this message, it will log error information to the CFHT log and terminate the client connection. Figure 9 contains a sample of a client detected protocol error.

```
PROTOCOL ERROR
```

This command does not have any input parameters and will not generate a response outside of the EOF sent across the interface to the client indicating that its connection was closed by the server.

5 Client C API

This section outlines the available functions within the C API for clients to interact with the Status Server. More details regarding the functionality provided by the client API is provided in the subsequent API Reference section.

5.1 API Reference

5.1.1 Access the Status Server

The client will initiate a connection request to the Status Server. For traceability purposes, the Client API will send a subsequent message with the UNIX Process ID and program name. When a client chooses to establish a connection with the Status Server, it must decide whether the Client API should automatically try to re-establish a connection to the Status Server if the connection is broken and whether a socket timeout threshold is desired.

A predefined socket timeout threshold will be defined as part of the Client API. This threshold can be modified by setting a new timeout value via the timeout parameter. If this parameter is less than or equal to 0, the system default will be used. Otherwise, the new value will be used for the socket timeout. The timeout must be specified in seconds.

It is possible for the socket connection between the Client API and Status Server to go down unexpectedly. The Client API will provide the option to retry until the connection is re-established. If the retry option is used, the client program will be blocked until the socket connection is re-established. Any time the Client API retries the connection, it will resend all “touch”, “touchdir” and “monitor” commands prior to processing the current API call. When the retry_pause parameter is set to a value greater than or equal to 0, automatic reconnection is turned on. If this value is less than 0, automatic reconnection is turned off. In addition, the retry_pause indicates the number of seconds the Client API will wait to initiate another connection to the Status Server.

The return value from the API call will indicate whether the operation was successful. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
PASSFAIL ssLogon(const char *program_name,
                 const int timeout,
                 const int retry_pause)
```

In addition, the client has the option of setting up client callback functions with the C API for the case where a timeout of the socket occurs or when the socket connection between the Client API and Status Server goes down unexpectedly. If automatic reconnection is enabled, the client callback for disconnection should never be called. Instead a function within the the Client API will be called to initiate a retry.

```
void ssRetryCallback(ss_callback_func retry_fn)

void ssDiscCallback(ss_disc_func disc_fn)
```

5.1.2 Disconnect from the Status Server

The C API will disconnect from the Status Server by closing the socket. As part of the disconnect, the Client API must clean up internal resources associated with the Status Server connection. This is also true for the case where the client detects that the client connection with the Status Server has been broken. When using the C API, a disconnect request should not fail unless a client connection isn’t available. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

If the client makes a call to ssLogoff, this will not cause the automatic retry mechanism to trigger. A connection to the Status Server must be subsequently established with another call to ssLogon.

```
PASSFAIL ssLogoff(void)
```

5.1.3 Create an object or register the intent to modify an object

The client must specify the name of the object and optionally specify a comment or lifetime. If a lifetime is not specified, the value of the object will not expire. When using the C API, there should not be any cases where a touch would fail.

```
void ssTouchObject(const char *name,
                   const char *comment,
                   const int *lifetime)
```

5.1.4 Update an object

The client has the ability to update Status Server objects of the following types:

- **String** - Strings can consist of a sequence of 8 bit ASCII characters with the exception of the NULL character. The NULL character will be used to terminate the string.
- **Boolean** - Data type consisting of two possible values; either TRUE or FALSE.
- **Floating Point** - Double precision floating point number.
- **Integer** - Signed integer number.

Clients using a telnet session will send all data as strings across the socket interface. This is the same way data is sent to the Status Server by the Client API. In order to support the ability to handle 8 bit ASCII characters within a string, the string is encoded into 7 bit ASCII printable characters by the Client API prior to being sent across the interface. In addition, boolean, floating point, and integer data is converted to a string prior to being sent to the Status Server. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
PASSFAIL ssPutInt(const char *name,
                  const int value)

PASSFAIL ssPutDouble(const char *name,
                     const double value)

PASSFAIL ssPutString(const char *name,
                     const char *value)

PASSFAIL ssPutBoolean(const char *name,
                      const BOOLEAN value)
```

5.1.5 Retrieve an object

The client will make a request to retrieve the value of an object. If the request is successful, the value of the Status Server object will be converted from it's encoded string format to the type requested via the C API call. This value will then be stored in the address specified by the user. If the request is not successful, the details regarding the error will be stored in cfht_errno.

```
PASSFAIL ssGetInt(const char *name,
                  int *value)

PASSFAIL ssGetDouble(const char *name,
                     double *value)

PASSFAIL ssGetString(const char *name,
                     char *value)

PASSFAIL ssGetBoolean(const char *name,
                      BOOLEAN *value)
```

5.1.6 Check for the existence and status of an object

For clients using the C API, the status of the object will be stored in a memory location specified by the user. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
PASSFAIL ssStat(const char *name,
                ss_stat_t *status)
```

5.1.7 Initiate a monitor on an object

The client has the option of placing a monitor on any object with optional age and deadband values. If deadband and/or deadband are not desired, it can be set to 0. This indicates the client would like to monitor each change to an object value. When monitors are applied using the C API library, an address must be supplied for both the monitored object value as well as the return code. The return code enables the server to notify a client whenever the state of an object changes. As a result, the client can be informed when an object becomes expired, NULL, or removed.

Once the client is informed that a monitor was successfully applied, the Client API will store an association between the name of the object and the value and return value addresses for the object. The Client API can then process monitor update notification messages and store the value and return codes in the proper memory locations for subsequent use by the client. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
PASSFAIL ssMonitorInt(const char *name,
                      const int deadband,
                      const int age,
                      int *value,
                      ss_ret_t *value_status)

PASSFAIL ssMonitorDouble(const char *name,
                         const double deadband,
                         const int age,
                         double *value,
                         ss_ret_t *value_status)

PASSFAIL ssMonitorString(const char *name,
                         const size_t max_length,
                         const int age,
                         char *value,
                         ss_ret_t *value_status)

PASSFAIL ssMonitorBoolean(const char *name,
                          const int age,
                          BOOLEAN *value,
                          ss_ret_t *value_status)
```

5.1.8 Remove a monitor from an object

The client will initiate the object removal request and should process the return value. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
ss_ret_t ssKillMonitor(const char *name)
```

5.1.9 Retrieve monitor updates

The client will initiate the monitor retrieval request and must be prepared to process the return information sent by the Status Server. The client will process each line of data until it receives the end-of-transaction indicator.

For clients using the C API, each line of monitored data will be converted and stored in the memory location which was previously defined during the setup of the monitor. Integer, floating point, and boolean data types will be converted from the string format received over the interface to the monitor requested data type. Any errors detected, either during the conversion process or from the data response sent by the Status Server, will be stored in the previously allocated memory location for return code information. From the C API point of view, this call should not fail unless a client connection is not available. When using the C API, there should not be any cases where a "poll" would fail.

```
void ssPoll(void)
```

5.1.10 Remove an object

The client will initiate the removal request and should check the return value to determine whether the operation was successful. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
PASSFAIL ssRemove(const char *name)
```

5.1.11 Get the current directory path

The client will initiate the current directory request and receive a pointer to a string containing the directory path. When using the C API, there should not be any cases where a request to retrieve the current directory path would fail.

```
char *ssPwd(void)
```

5.1.12 Change the current directory

The client will initiate the change current directory request and should check the return value to determine whether the operation was successful. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
PASSFAIL ssChdir(const char *path)
```

5.1.13 Create directory or register intent to remove a directory

Prior to removing a directory and its contents the client must perform a "touchdir" on the directory. The "touchdir" also enables a client to create the directory if it doesn't already exist and to alter the comment associated with a directory. When using the C API, there should not be any cases where a touchdir would fail.

```
void ssTouchDir(const char *path,
                const char *comment)
```

5.1.14 Remove a directory

The client will initiate the directory removal request and should check the return value to determine whether the operation was successful. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
PASSFAIL ssRmdir(const char *path)
```

5.1.15 Retrieve the contents of a directory

The client will initiate the directory content retrieval request and must be prepared to process the return information sent by the Status Server. The client will process each line of data until it receives the end of message indicator.

When using the C API, the client will first make a request to open the directory. The directory can be opened with a recursive option indicating that all underlying contents will be returned as part of the request. The return value from the ssOpendir call will indicate whether the directory could successfully be opened. The client must then call ssReaddir until it receives a NULL indicating that all the directory contents have been returned. Directory contents will be returned in an Object Name=Value and Dir=Name format. For example, /i/cfh12k/etype=”FLAT”. If the ssOpendir call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
PASSFAIL ssOpendir(const char *path)

char *ssReaddir(void)
```

5.1.16 Initiate a trace

The client can initiate a trace on all Status Server activity. When using the C API, there should not be any cases where a trace initiation request would fail.

```
void ssTraceOn(void)
```

5.1.17 Stop a trace

A client can stop a trace currently running in the Status Server. When using the C API, there should not be any cases where a request to stop a trace would fail. If a trace was not running and a request was made to stop a trace, the Status Server will still send back a positive response.

```
void ssTraceOff(void)
```

5.1.18 Serialize Status Server data to a file

A client can request that the Status Server serialize itself to a file. When using the C API, there should not be any cases where a serialization request would fail. It is possible that the serialization itself might fail, but the Status Server does not send a response back to the client once the Serialization is completed. Instead, it sends back a response once it receives a request to initiate serialization.

```
void ssAutosave(void)
```

5.1.19 Shutdown the Status Server

The client can request a shutdown of the Status Server. This function will return once the message is sent to the Status Server.

```
    void ssShutdown(void)
```

6 Software Design

This section provides the design details of the Status Server. At this point, the document only covers the data structures and software components which make up the Status Server. Coverage of the Client API will be added to this document soon.

6.1 Status Server Data Structures

There are several key data structures used to hold Status Server data. Figure 13 illustrates the data structures used by the Status Server. The lines within the figure illustrate how the data is structured and the relationships between data structures. Each data structure is explained in more detail in the following sections.

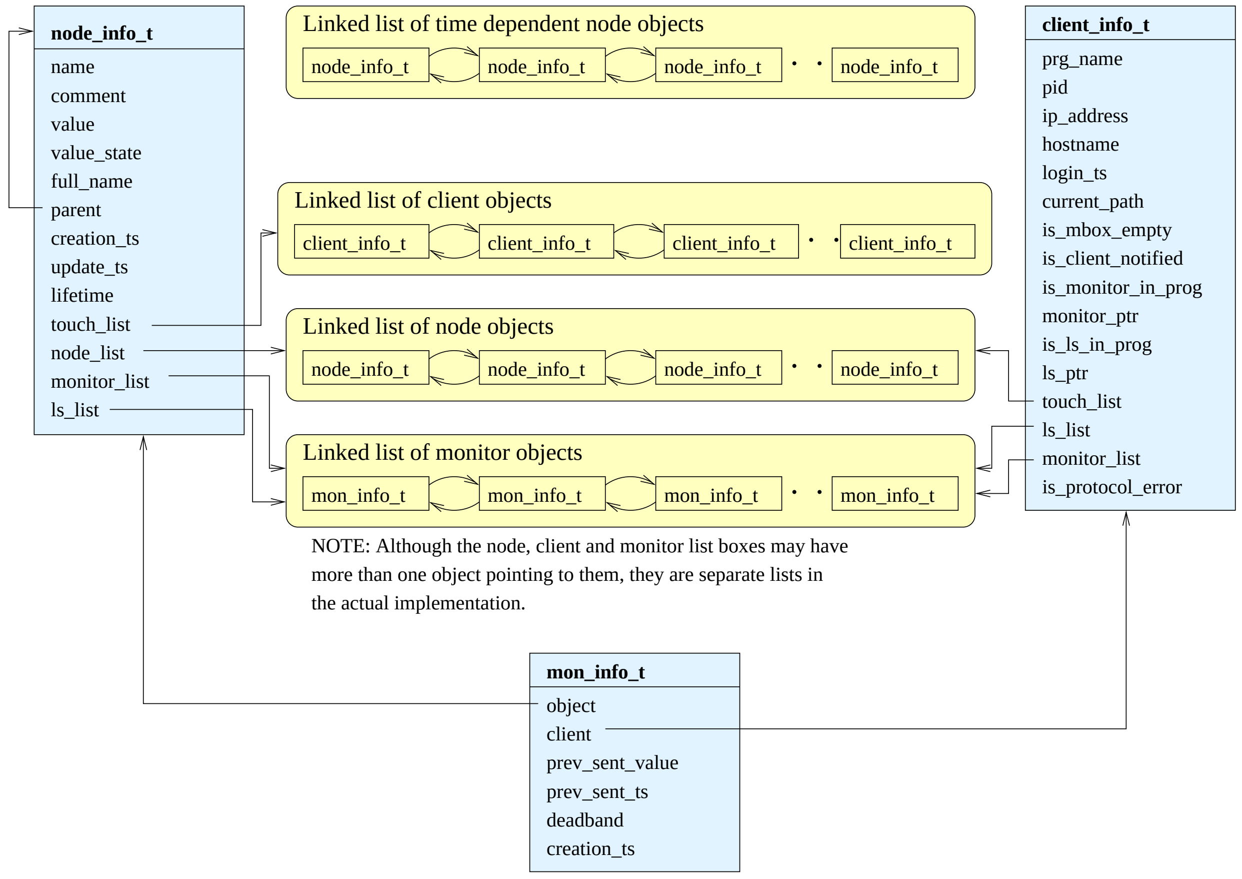

Figure 13: Status Server Data Structures

6.1.1 Client Connection Data (client_info_t)

The sockio library will manage the low-level socket details associated with the client-server socket connection. At a higher level, this data structure will be used to manage Status Server information associated with each client connection.

```
typedef struct {
   char *prg_name;
   pid_t pid;
   unsigned_char ip_address[4];
   char *hostname;
   time_t login_ts;
   char *current_path;
   BOOLEAN is_mbox_empty;
   BOOLEAN is_client_notified;
   BOOLEAN is_monitor_in_prog;
   mon_info_t *monitor_ptr;
   BOOLEAN is_ls_in_prog;
   mon_info_t *ls_ptr;
   linked_list *touch_list;
   linked_list *ls_list;
   linked_list *monitor_list;
   BOOLEAN is_protocol_error;
} client_info_t;
```

More details regarding each of the fields in the client_info_t structure can be found in figure 14.

6.1.2 Directory and Object Data (node_info_t)

This section outlines the data structure used to define the Status Server directory hierarchy and store object data. Each node within the Status Server hierarchical structure is described by a node_info_t structure. A single data structure is used to hold either object data or directory data since both node types have very similar data requirements. The data stored within the node_info_t data structure will be object data if the node_list is set to NULL. If the node_list field is not set to NULL, the data stored in the node_info_t data structure will be directory data. When the node_list is not null, it is possible for a directory to have both subdirectories and/or objects associated with the directory. The lifetime and monitor_list fields within the data structure are data object specific and will not apply to directory data. If it is subsequently determined that placing a monitor on a directory is useful, the value and monitor_lists could be used for this purpose.

The value field within the node_info_t structure will contain valid data or an indication of why the value is not valid. Valid values will always be enclosed within double quotes. If the value of an object is not valid, it will not be enclosed within double quotes. For example, the following values would be considered valid: “data”, “0.0”, or “sample data”. If a value is not enclosed within double quotes, it must always be one of the following values: NONEXISTENT, UNDEFINED, or EXPIRED in the case of a data object. Directories will always contain the string “DIRECTORY” in the value field.

It is possible for a data object to have NONEXISTENT populated in the value field. While it may seem strange to define a data object with a state indicating that it doesn’t exist, this is used to support the ability to apply a monitor on a data object which has not been created. In this case, the required directories to hold the data object will be created and the data object will be created with a NONEXISTENT value. As a result, it will become possible to define pointers between a monitor and the object being monitored. If a subsequent “touch” request is made to a data object, the value of the data object will no longer indicate NONEXISTENT.

In addition to storing the state of a directory or data object in the value field, the value_state field contains an enumerated type indicating the state of an object. This field is added to enable more efficient state comparisons.

Field	**Default**	**Description**
prg_name	"UNKNOWN"	The name of the client program interacting with the Status Server. For clients using the C-API, this should correspond to argv[0]. The program name is provided through the "register" command.
pid	0	The UNIX Process ID (PID) of the client program interacting with the Status Server. This will automatically be provided to the Status Server for clients using the C-API. The PID is provided through the "register" command.
ip_address	Client IP address	The remote IP address of the client.
hostname	Client Hostname	The hostname of the client.
login_ts	Connect Time	Time a client connection was established with the Status Server. This is expressed as the number of seconds elapsed since 00:00:00 hours, GMT, January 1, 1970.
current_path	"/"	The current path prepended to relative path references of objects or directories. The value of this field can be changed by using the "CD" command.
is_mbox_empty	TRUE	Boolean flag indicating whether there any monitoring notifications to be sent to this client. TRUE if a monitoring notifications must be sent. FALSE if a monitoring notification does not need to be sent.
is_client_notified	FALSE	Boolean flag indicating whether the Status Server has notified the client that it has monitor updates to retrieve. TRUE if the client has been nofified. FALSE if the client has not been notified.
is_monitor_in_prog	FALSE	Boolean flag indicating whether the Status Server is currently sending monitor updates to this client. TRUE if monitor updates are in progress. FALSE if monitor updates are not in progress.
monitor_ptr	NULL	Pointer to the next monitor object within the monitor_list which must be checked to determine if it is eligible to be sent to the client in a monitor update message. If the client is not currently retrieving monitor updates, this field will be set to NULL.
is_ls_in_prog	FALSE	Boolean flag indicating whether the Status Server is currently sending directory contents to this client. TRUE if the directory contents are being sent. FALSE if directory contents are not being sent.
ls_ptr	NULL	Pointer to the next directory listing response within the ls_list which must be sent to the client. If the client is not currently retrieving directory listing data, this field will be set to NULL.
touch_list	NULL	Linked list of directories and data objects which this client has performed a "touchdir" or "touch" on. If the client has not performed a "touchdir" or "touch" on any directories or data objects, the touch_list pointer will be NULL.
ls_list	NULL	Linked list of directory listing responses to be sent to the client during a directory listing request.
monitor_list	NULL	Linked list of monitor objects. If this client is not monitoring any Status Server Objects, the monitor_list pointer will be NULL.
is_protocol_error	FALSE	This flag will be set if the client ever sends an out-of-sequence command violating the Status Server protocol. At this point, the Status Server expects the client to close its connection. However, if the connection is not closed and another request is received, the Status Server will close the connection. TRUE if a protocol error has occured. FALSE if a protocol error has not occured.

Figure 14: Description of Fields within Client Data Structure

When the Status Server is initialized, it will contain only one directory node, which is that of the root directory. Much like the UNIX file system, the root directory is described by a single forward slash '/'. Additional directories will be created when the Status Server receives client "touch", "touchdir", or "monitor" requests.

Data object nodes are created when a client either initiates a "touch" or "monitor" request. As previously mentioned, a data object will use the value, lifetime, and monitor_list fields within the node structure.

In the case of either a directory or object data, a node will contain a pointer to its parent. In the case of a directory, this will always be the parent directory. The root node of the directory is a special case and will have the parent pointer set to itself. Data objects will always point to the directory which has the linked list containing a pointer to the data object itself.

```
typedef struct {
   char *name;
   char *comment;
   char value[MAX_VALUE_SIZE];
   node_state_t value_state;
   char *full_name;
   node_info_t *parent;
   time_t creation_ts;
   time_t update_ts;
   time_t lifetime;
   linked_list *touch_list;
   linked_list *node_list;
   linked_list *monitor_list;
   linked_list *ls_list;
} node_info_t;


typedef enum {
   SS_NONEXISTENT,
   SS_NOTDEFINED,
   SS_EXPIRED,
   SS_VALID
} node_state_t;
```

More details regarding each of the fields in the node_info_t structure can be found in figure 15.

6.1.3 Monitoring Data (mon_info_t)

This section outlines the data structure used to store monitoring information for a client based on modifications to an object. The information within the monitoring data structure is essentially a bridge between a client and data object. Whenever a client requests a monitor to be placed on an object, the object will be created if it didn't already exist, before the monitor object is created. Once the monitor object is created, a pointer to the object will be stored within the monitor lists of both the client object and data object being monitored. If a client disconnects from the Status Server, the monitoring objects and monitor object reference will be removed before the client object is removed.

```
typedef struct {
   node_info_t *object;
   client_info_t *client;
   char prev_sent_value[MAX_VALUE_SIZE];
   time_t prev_sent_ts;
   double deadband;
```

Field	**Default**	**Description**
name	directory or object name	The name of the directory or data object represented as a relative name with respect to its parent directory.
comment	Comment or NULL	Descriptive comment of this directory or object. If a comment is not supplied by the client, it will be set to NULL.
value	NONEXISTENT or NOT-DEFINED for a data object. A valid value for a directory	The value of a data object or directory. This field will either contain valid data which is enclosed with double quotes or a string indicating why the field is not valid. For a directory object this field will always contain a valid value enclosed in double quotes. For a data object this field will either contain valid data or NONEXISTENT, NOTDEFINED, or EXPIRED.
value_state	SS_NONEXISTENT, SS_VALID, or SS_NOTDEFINED	An enumerated type which defines the state of the object or directory. This information can also be determined from the value field, but the enumerated type provides a more efficient means of identifying the state of an object. Directories will always be created with a value_state of SS_VALID. Data objects will be created as either SS_NONEXISTENT or SS_NOTDEFINED.
full_name	absolute path	Fully qualified path of the directory or data object. This will always be expressed in absolute terms starting at the root directory (“/”).
parent	parent node or itself	In the case of a directory or data object, this field will point to the parent directory object. In the case of the root directory object, this field will point back to itself.
creation_ts	creation time	Time the directory or data object is created in the Status Server. This is expressed as the number of seconds elapsed since 00:00:00 hours, GMT, January 1, 1970.
update_ts	update time	Time the directory or data object is last updated in the Status Server. This field is only updated when the state or value of an object changes. The first update_ts will be set to the time the object is created. This is expressed as the number of seconds elapsed since 00:00:00 hours, GMT, January 1, 1970.
lifetime	lifetime or 0	Number of seconds following each update which the object will be considered valid. If a lifetime was not provided by the client, it will be set to 0.
touch_list	empty list	Linked list of clients which have performed a “touch” on this directory or data object.
node_list	NULL or empty list	A linked list of pointers to directory and data objects. This list will only be used for directory objects. With data objects, this field will be NULL.
monitor_list	empty list	A linked list of pointers to monitor objects placed on a data object. This list will only be used for data objects.
ls_list	empty list	A linked list of pointers to monitor objects used to return directory and data object information.

Figure 15: Description of Fields within the Directory and Data Object Structure

```
        time_t creation_ts;
     } mon_info_t;
```

More details regarding each of the fields in the mon_info_t structure can be found in figure 16.

Field	Default	Description
object	pointer to object	A pointer to the data object which is being monitored (node_info_t).
client	pointer to client	A pointer to the client data structure (client_info_t).
prev_sent_value	UNSPECIFIED	The previous value of the object sent to the client. When the monitoring object is initially defined, it will be set to UNSPECIFIED.
prev_sent_ts	0	Time when the client was last sent a monitoring update. When the monitoring object is initially defined, it will be set to 0. This is expressed as the number of seconds elapsed since 00:00:00 hours, GMT, January 1, 1970.
deadband	deadband or 0.0	Deadband value specified by the client. If a deadband is not specified, it will be set to 0.0.
creation_ts	creation time	Time the monitoring object was created in the Status Server. This is expressed as the number of seconds elapsed since 00:00:00 hours, GMT, January 1, 1970.

Figure 16: Description of Fields within Monitoring Data Structure

6.2 Status Server Software Components

The Status Server is broken in a set of components illustrated in figure 17. A brief description of each of the components follows with a more in depth description available in the following sections.

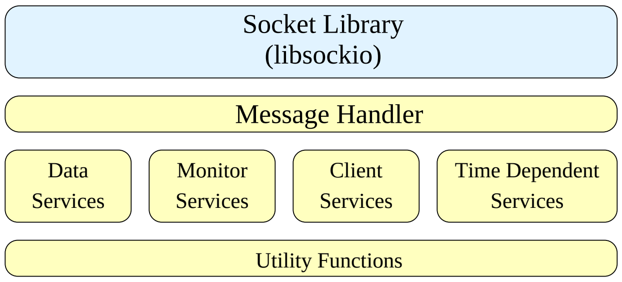

Figure 17: Status Server Software Components

- **Socket Library** - The socket I/O library to handle the low-level socket details.
- **Message Handler** - Processes the messages sent from the Client and handles any response which must be sent back to the client.
- **Data Services** - Provides functions to manage the directory and data object structure (node_info_t).
- **Monitor Services** - Provides functions to manage the monitoring data structure (mon_info_t).
- **Client Services** - Provides functions to manage the client data structure (client_info_t).
- **Time Dependent Services** - Provides functions to manage a list of data objects which have a time dependency based on a non-zero lifetime.
- **Utility Functions** - Provides a set of utility functions shared among the Status Server components.

6.2.1 Message Handling Services

This component interacts closely with libsockio, and will set up the following callback routines to be called by libsockio.

- client_add_hook() - Called whenever a new client is connecting to the Status Server. This routine will check to make sure the client is connecting from a valid IP address. The IP address must be 127.*, 128.171.80.*, or 128.171.83.*. If a client attempts to connect from an invalid address, the connection will be refused. If the connection is successful, the message handler will call the createClient() routine in the Client Services component to create a client data structure and return this structure to libsockio to associate with the socket connection.
- client_del_hook() - Called whenever an end-of-file or error condition occurs with a client. The message handler will call the removeClient() routine in the Client Services component to clean up monitors associated with this client and all other client specific data.
- client_recv_hook() - Called whenever the client has sent a message to be processed. The message handler will parse the message and will call the appropriate service components to process the message. Once the message is processed, the message handler will send the appropriate response back to the client.
- client_send_hook() - Called whenever the output buffers are empty in case the client wishes to send an asynchronous message to the client. The message handler, via service routines, will check if a mailbox message, monitor information, or directory listing data is ready to be sent to the client.

This is the main component of the Status Server and it initiates all the message processing done by the Status Server. In addition, it contains main() and sets up the sockio library calls to initiate and manage the server socket connection. The message handler also handles a reload of the Status Server from serialized data. The Status Server will save it's internal state via a set of messages similar to the messages sent by a client. As a result, restoring the Status Server is simply a process of replaying a set of previously sent transactions.

The main() function within the message handler will set up the commands which manage the interaction with the sockio library. An example of a code segment which could be used to manage the socket communications via the sockio library is as follows:

```
int main(int argc, const char *argv[])
{
   sockserv_t *statserv = sockserv_create(port #);

   /* Check if the socket could be created successfully */
   if (!statserv)
      exit(EXIT_FAILURE);

   /* Set up the callback functions */
   statserv->client_recv_hook = client_recv_hook;
   statserv->client_del_hook = client_del_hook;
   statserv->client_add_hook = client_add_hook;
   statserv->client_send_hook = client_send_hook;

   /* Set up an infinite loop to manage the socket communication */
   for (;;) {

      /* Service all time dependent objects */
      serviceTimeDepObjects(void);

      /* Continue calling sockserv_run with a timeout  */
      /* value of zero if it returns a positive return */
      /* value.  This indicates that something was     */
      /* received or sent during the last iteration.   */
      while (sockserv_run(statserv, 0) > 0) { ; }
```

```
         /* Call sockserv_run with a timeout interval     */
         /* corresponding to when the next time dependent */
         /* object should expire.                         */
         sockserv_run(statserv, getMinimumTimoutInterval() * 100);
      }
      exit(0);
   }
```

The message handler will handle each Client request and is responsible for writing a response to sockio buffer provided by the client_recv_hook() and client_send_hook() callback functions.

When a request is received from a client, it will be checked to make sure it is a valid URL formatted string. If so, the first argument will parsed and checked against the set of known commands. If it is recognized as a legal command, it will be processed according to the type of command it is. Each of the following sections provides a brief overview of the processing performed for each command and the utility functions within the other software components which will be called. Most of the functionality has been described as the pseudocode processing which occurs when each command is received by the Status Server.

6.2.1.1 Register Client with the Status Server

The pseudocode processing for how a client “register” request is handled is as follows:

```
if (register message not valid) {
   write a syntax error message to the sockio buffer
   return
}

update the client object with updateClient()
write a positive response to the sockio buffer
```

6.2.1.2 Disconnect from the Status Server

The pseudocode processing for how a client “logoff” request is handled is as follows:

```
call the client\_del\_hook callback function
write an empty string (buf[0] = ’\0’) to the sockio buffer
```

6.2.1.3 Create an object or register the intent to modify an object

The pseudocode processing for how a client “touch” request is handled is as follows:

```
if (touch message not valid) {
   write a syntax error message to the sockio buffer
   return
}

/* Try to create/retrieve the object from the hierarchy.   */
/* The create parameter to getObject() must be set to TRUE */
/* and the create_valid parameter must be set to TRUE.     */
if (getObject() == NULL) {
   Status Server internal error
```

```
      write a negative response to the sockio buffer
      return
   }
   else {
      if a lifetime was specified, set it with setObjectLifetime()
      if a comment was specified, set it with setNodeComment()
   }
   add the data object to the client touch list using addTouchNode()
   write a positive response to the sockio buffer
```

6.2.1.4 Update an object

The pseudocode processing for how a client “put” request is handled is as follows:

```
   if (put message not valid) {
      write a syntax error message to the sockio buffer
      return
   }

   /* Try to retrieve the object from the hierarchy.  Both  */
   /* The create parameter and create_valid parameters must */
   /* be set to false in the call to getObject().           */
   if (getObject() == NULL) {
      write object does not exist error response to the sockio buffer
      return
   }

   /* Check if the object was previously created, but has a */
   /* value of NONEXISTENT */
   if (isNodeNonexistent() == TRUE) {
      write object does not exist error response to the sockio buffer
      return
   }

   /* Check to see if this object is within the list of */
   /* objects the client has performed a ‘‘touch’’ on   */
   if (touchNodePerformed() == FALSE) {
      write permission denied error response to the sockio buffer
      return
   }

   Update the value of the object with setObjectValue()
   write a positive response to the sockio buffer
```

The pseudocode processing is slightly different than the flow diagram for an update in the current Functional Specification document. In this design, the Status Server will always traverse the directory and data object hierarchy in an attempt to find the data object and from there determine if the client has performed a touch. This is a simpler and more efficient implementation than checking the touch_list associated with the client object first to see if the object has been touched by the client. This would not be the case if all client requests were expressed in terms of an absolute path. However, by traversing the directory and data object hierarchy it is possible to resolve the path of the object on the fly instead of converting the path string to absolute path, finding a string match on the absolute path name in the client object touch_list and then referencing the data object. As a result, the first response that a client may see is that the data object does not exist. Only if the object exists is it possible for the client to receive a notification that it has not performed a touch on the data object.

6.2.1.5 Retrieve an object

The pseudocode processing for how a client “get” request is handled is as follows:

```
if (get message not valid) {
   write a syntax error message to the sockio buffer
   return
}

/* Try to retrieve the object from the hierarchy.  Both  */
/* The create parameter and create_valid parameters must */
/* be set to false in the call to getObject().           */
if (getObject() == NULL) {
   write object does not exist error response to the sockio buffer
}
else {
   write positive data object value response to the sockio buffer
}
```

6.2.1.6 Initiate a monitor on an object

The pseudocode processing for how a client monitoring request is handled is as follows:

```
if (monitor message not valid) {
   write a syntax error message to the sockio buffer
   return
}

/* Try to create/retrieve the object from the hierarchy.   */
/* The create parameter to getObject() must be set to TRUE */
/* and the create_valid parameter must be set to FALSE.    */
if (getObject() == NULL) {
   Status Server internal error
   write a negative response to the sockio buffer
   return
}

/* Check to see if a monitor already exists on that */
/* object for the requesting client */
if (getMonitorByClient() == NULL) {

   /* Check to make sure the monitoring object could */
   /* be successfully created.                       */
   if (createMonitor() == PASS) {
      write a positive response to the sockio buffer.
      return
   }
   else {
      Status Server internal error
```

```
            write a negative response to the sockio buffer.
            return
         }
      }
      else {
         Update the deadband for a monitor with updateMonitorDeadband()
      }
      write a positive response to the sockio buffer.
```

6.2.1.7 Remove a monitor from an object

The pseudocode processing for how a monitor will be removed from an object:

```
      if (unmonitor message not valid) {
         write a syntax error message to the sockio buffer
         return
      }

      /* Try to retrieve the object from the hierarchy.  Both  */
      /* the create parameter and create_valid parameters must */
      /* be set to false in the call to getObject().  If the   */
      /* object does not exist, the monitor also won't exist   */
      /* since an object can not be removed while it is being  */
      /* monitored.  */
      if (getObject() == NULL) {
         write a monitor does not exist error response to the sockio buffer.
         return
      }

      /* Check to see if a monitor already exists on that */
      /* object for the requesting client */
      if (getMonitorByClient() == NULL) {
         write a monitor does not exist error response to the sockio buffer.
         return
      }

      /* Remove the monitoring record. */
      if (removeMonitor() == FAIL) {
         write an internal error response to the sockio buffer
      }
      else {
         write a positive response to the sockio buffer.
      }
```

6.2.1.8 Retrieve monitor updates

There are two stages where monitor retrievals occur. First is the processing of the monitoring request, and second is the handling of the monitor responses. In the first stage, the processing is triggered by a call to the client_recv_hook(). In the second stage, the processing is triggered by repeated calls to the client_send_hook(). At most, only one message can be sent to the client with each client_send_hook() callback function call. As a result, if the client must be notified of many monitor updates, the client_send_hook() will be called multiple times.

The pseudocode processing for how a client “poll” request is handled is as follows:

```
  STAGE 1: Processing initiated when a ‘‘poll’’ request is received.

     /* Check to see if the client was sent a mailbox message. */
     /* If not, this would be considered a protocol error,     */
     /* since the client should not be sending a ‘‘poll’’      */
     /* request without first being informed with a mailbox    */
     /* message. */
     if (wasMailboxMsgSentToClient == FALSE) {
        write a protocol error message to the sockio buffer
        mark the client object to indicate that a protocol error was sent
        return
     }

     /* Set up the client object fields to indicate that a    */
     /* monitor request is in progress.  This function should */
     /* only fail if the client does not have any monitors    */
     /* defined. */
     if (setMonitorInProgress() == FAIL) {
        write a nothing monitored by client error message to the sockio buffer
        return
     }
     else {
        Call checkSendMonitor() to send monitoring data to the client
     }

  STAGE 2: Processing initiated each time the client receives a
          client_send_hook() callback function call.

     /* Check to see if monitoring data must be sent to the client */
     checkSendMonitor()
```

It is important to note that the processing of a directory listing and monitor retrieval is very similar since each case requires an iteration through a list of monitor objects. As a result, it is possible to reuse much of the same code.

6.2.1.9 Remove an object

The pseudocode processing for how a client object removal request is handled is as follows:

```
     if (rm message not valid) {
        write a syntax error message to the sockio buffer
        return
     }

     /* Try to retrieve the object from the hierarchy.  Both  */
     /* The create parameter and create_valid parameters must */
     /* be set to false in the call to getObject().           */
     if (getObject() == NULL) {
        write object does not exist error response to the sockio buffer
     }

     /* Check to see if the object exists but is marked */
     /* as ‘‘does not exist’’. */
```

```
if (isNodeNonexistent() == TRUE) {
   write object does not exist error response to the sockio buffer
   return
}

/* Check to see if this client is within the list of    */
/* clients who have performed a ‘‘touch’’ on the object */
if (touchNodePerformed() == FALSE) {
   write permission denied error response to the sockio buffer
   return
}

/* Check whether the object can be cleanly removed */
/* using the rmObject() function call */
if (rmObject() == PASS) {
   write a positive response to the sockio buffer
}
else {
   write an internal error response to the sockio buffer
}
```

6.2.1.10 Get the current directory path

The pseudocode processing for how a client “pwd” request is handled is as follows:

```
get the current path with getCurrentPath()
write current path to the sockio buffer
```

6.2.1.11 Change the current directory

The pseudocode processing for how a client “cd” request is handled is as follows:

```
/* Make a request to change the current directory path */
if (changeCurrentPath() == FAIL) {
   write directory does not exist error response to the sockio buffer
   return
}
else {
   write new directory path response to the sockio buffer
}
```

6.2.1.12 Create directory or register intent to remove a directory

The pseudocode processing for how a client “touchdir” request is handled is as follows:

```
if (touchdir message not valid) {
   write a syntax error message to the sockio buffer
   return
}
```

```
/* Use the directory path and try to retrieve the     */
/* directory.  The create parameter to getDir() must  */
/* be set to TRUE and the create_valid parameter must */
/* be set to TRUE */
if (getDir() == FAIL) {
   Status Server internal error
   write a negative response to the sockio buffer
   return
}
else {
   if a comment was specified, set it with setNodeComment()
}
add the directory to the client touch list using addTouchNode()
write a positive response to the sockio buffer
```

6.2.1.13 Remove a directory

The pseudocode processing for how a client “rm -r” request is handled is as follows:

```
if (rmdir message not valid) {
   write a syntax error message to the sockio buffer
   return
}

/* Use the directory path and try to retrieve the     */
/* directory.  The create parameter to getDir() must  */
/* be set to FALSE. */
if (getDir() == FAIL) {
   write a directory not found error response to the sockio buffer
   return
}

/* Check to see if this directory is within the list of */
/* directories the client has performed a ‘‘touch’’ on  */
if (touchNodePerformed() == FALSE) {
   write permission denied error response to the sockio buffer
   return
}

/* Try to remove the directory.  At this point, the function  */
/* should only fail if the directory has valid subdirectories */
if (rmDir() == PASS) {
   write a positive response to the sockio buffer
}
else {
   write directory contains subdirectories error response to the sockio buffer
}
```

As it is currently designed, directories will never be completely removed as long as they have data objects within them or subdirectories. In this case it is possible for a directory to have a NONEXISTENT data object within it for the case of a monitor added to a data object, so it will not be possible to remove the directory. Currently, only data objects have a value state of NONEXISTENT while directories are always visible. This should not be a problem with the envisioned model for creating and removing directories to hold FITS header information. If needed, directories could also be given a NONEXISTENT state and effectively hidden from view. This will add some additional complexity in the checks required when removing data objects and directories. Presumably when a data object or directory is removed, some recursive traversal of that tree would need to be performed in order to determine whether directory objects could then be removed more permanently.

6.2.1.14 Retrieve the contents of a directory

There are two stages where a retrieval of directory contents occurs. First is the processing of the directory retrieval request (“ls”), and second is the handling of the directory responses. In the first stage, the processing is triggered by a call to the client_recv_hook(). In the second stage, the processing is triggered by repeated calls to the client_send_hook(). At most, only one message can be sent to the client with each client_send_hook() callback function call. As a result, if the client must be sent the contents of a large directory, the client_send_hook() will be called multiple times.

The pseudocode processing for how a client “ls” request is handled is as follows:

```
STAGE 1: Processing initiated when a ‘‘ls’’ request is received.

   if (ls message not valid) {
      write a syntax error message to the sockio buffer
      return
   }

   /* Use the dir_path and try to retrieve the directory  */
   /* The create parameter to getDir must be set to FALSE */
   if (getDir() == FAIL) {
      write directory does not exist error response to the sockio buffer
      return
   }

   /* Use the directory object returned by getDir() as  */
   /* the base directory to perform the ls on.  Regular */
   /* expression rules will be applied to each data     */
   /* object name to determine if a match exists.  If   */
   /* so, the object name and value will be stored in a */
   /* linked list.  */
   set up the ls_list of ls monitor objects via setupDirectoryListing()

   /* Set up the client object fields to indicate that an ls */
   /* request is in progress. */
   setLSInProgress()

   write the directory header to the sockio buffer

STAGE 2: Processing initiated each time the client receives a
         client_send_hook() callback function call.

   /* Check to see if monitoring data must be sent to the client */
   checkSendLS()
```

As previously mentioned, the processing of a directory listing and monitor retrieval is very similar since each case requires an iteration through a list of monitor objects. As a result, it is possible to reuse much of the same code.

6.2.1.15 Initiate a trace

The pseudocode processing for how a client “trace on” request is handled is as follows:

```
set the global flag indicating that tracing is enabled
write a positive response to the sockio buffer
```

6.2.1.16 Stop a trace

The pseudocode processing for how a client “trace off” request is handled is as follows:

```
set the global flag indicating that tracing is disabled
write a positive response to the sockio buffer
```

6.2.1.17 Serialize Status Server data to a file

The pseudocode processing for how a client “autosave” request is handled is as follows:

```
inform client that it has successfully receive the request
execute the serialize() function with the fork flag set to TRUE
```

6.2.1.18 Shutdown the Status Server

The pseudocode processing for how a client “shutdown” request is handled is as follows:

```
execute the serialize() function with the fork flag set to FALSE
initiate an exit()
```

6.2.2 Client Services

This component is responsible for managing the data closely associated with a client connection. Most of the functions manipulate information contained within the client_info_t data structure.

6.2.2.1 Create a client object

This function creates the client object. Figure 14 contains the default values within the client_info_t structure when the client object is created. The function will return the newly created client_info_t structure for the client.

```
client_info_t *createClient(void)
```

6.2.2.2 Update a client object

When a client sends the “register” command with the program name and UNIX Process ID (PID), this function will be called to add this information to the client_info_t structure associated with the client connection.

```
void updateClient(client_info_t *client,
                  const char *prg_name,
                  int pid)
```

6.2.2.3 Check if a mailbox message was sent to a client

The client should only retrieve monitor information once it has been informed with a mailbox message that there is monitored data available for retrieval. The following function will return a boolean indicator of whether the client was sent the mailbox message. TRUE indicates that a mailbox message was sent to the client. FALSE indicates that a mailbox message was not sent to the client.

```
BOOLEAN wasMailboxMsgSentToClient(const client_info_t *client)
```

6.2.2.4 Check if monitor information must be sent to the client

When the client_send_hook() within the Message Handling Services component is called, a call will be made to this function to determine whether monitoring data must be sent to the client. If so, it will either send out the next monitoring update or the EOT message if all monitoring information has been sent to the client.

This function will check whether the is_monitor_in_prog field within the client_info_t structure is set to TRUE. If so, it will start at the monitor_ptr position within the monitor_list and send the next qualifying monitor update to the client. If the monitor_ptr is set to NULL, or there is no more monitoring information to be sent to the client, an EOT message will be sent. Once the EOT message has been sent, the is_monitor_in_prog field will be set to FALSE and the monitor_ptr will be set to NULL.

If the is_monitor_in_prog field is set to FALSE, the is_mbox_empty flag is set to FALSE, and the is_client_notified flag is set to FALSE the client will be sent a mailbox message indicating that there is monitored information ready for retrieval. Once this message has been sent, the is_client_notified flag will be set to TRUE.

```
void checkSendMonitor(client_info_t *client)
```

6.2.2.5 Set flag indicating monitoring is in progress

When the client_recv_hook() within the Message Handling Services component receives a "poll" request, a call will be made to this function to set the appropriate fields to indicate that a monitoring update retrieval command is now in progress.

This function will perform the following actions:

- Set the is_mbox_empty flag to TRUE
- Set the is_client_notified flag to FALSE
- Set the is_monitor_in_prog flag to TRUE
- Set the monitor_ptr flag to the first object in the linked list of monitoring objects for this client. If for some reason, this client received a "poll" request, but it does not have any monitors in its monitor_list, the function will return FAIL.

```
PASSFAIL setMonitorInProgress(client_info_t *client)
```

6.2.2.6 Set flag indicating a directory listing is in progress

When the client_recv_hook() within the Message Handling Services component receives an "ls" request, a call will be made to this function to set the appropriate fields to indicate that a directory listing is now in progress.

This function will perform the following actions:

- Build a linked list of monitor objects hold directory listing information by calling setupDirectoryListing().
- Set the is_ls_in_prog flag to TRUE
- Set the ls_ptr flag to the first object in the linked list of monitor objects used to hold directory listing information. If the linked list is either NULL or empty, ls_ptr will be set to NULL.

```
void setLSInProgress(client_info_t *client)
```

6.2.2.7 Check if directory listing information must be sent to the client

When the client_send_hook() within the Message Handling Services component is called, a call will be made to the this function to determine whether directory listing data must be sent to the client. If so, it will either send out the next directory listing message or the EOT message if all directory listing information has been sent to the client.

This function will check whether the is_ls_in_prog field within the client_info_t structure is set to TRUE. If so, it will start at the ls_ptr position within the ls_list and send the next directory listing to the client. If the ls_ptr is set to NULL, or there are no more directory contents to be sent to the client, an EOT message will be sent. Once the EOT message has been sent, the is_ls_in_prog field will be set to FALSE, and the ls_ptr will be set to NULL.

```
void checkSendLS(client_info_t *client)
```

6.2.2.8 Add a directory object or data object to the client touch list

This function will add a directory object or data object to the linked list used to store objects which this client has performed a touch on. The directory object or data object will only be added to the list if it doesn't already exist in the list.

```
void addTouchNode(client_info_t *client,
                  const node_info_t *node)
```

6.2.2.9 Check whether a data object or directory object is part of the client touch list

This function will take a directory object or data object and check whether a "touch" or "touchdir" was previously performed on the object by a client. The function will return TRUE if the object is contained within the linked list of "touched" objects. Otherwise, the function will return FALSE.

```
BOOLEAN touchNodePerformed(const client_info_t *client,
                           const node_info_t *node)
```

6.2.2.10 Add a monitoring object to the client monitor list

This function will add a monitoring object to the linked list used to store monitor objects.

```
PASSFAIL addMonitorToClient(client_info_t *client,
                            const mon_info_t *mon)
```

6.2.2.11 Remove a monitoring object from the client monitor list

This function will remove a monitoring object from the linked list used to store monitor objects. If this monitoring object can not be found the function will return FAIL.

```
PASSFAIL removeMonitorFromClient(client_info_t *client,
                                 const mon_info_t *mon)
```

6.2.2.12 Add a monitoring object to the client ls list

This function will add a monitoring object to the linked list used to store monitor objects for directory listings.

```
PASSFAIL addMonitorLS(client_info_t *client,
                      const mon_info_t *mon)
```

6.2.2.13 Remove a monitoring object from the client monitor list

This function will remove a monitoring object from the linked list used to store monitor objects for directory listings. If a monitoring object can not be found the function will return FAIL.

```
PASSFAIL removeMonitorLS(client_info_t *client,
                         const mon_info_t *mon)
```

6.2.2.14 Set the mailbox flag indicating that monitors are available

During object updates or removals, if the monitor criteria has been met to indicate that a client must need to be notified, this function will be called. This function causes the is_mbox_empty flag to be set to FALSE. As a result, a mailbox message will be sent if the is_client_notified flag is also set to FALSE.

```
void setMailAvailable(client_info_t *client)
```

6.2.2.15 Get the current path associated with a client

This function will return the current path used by the Status Server for relative path references associated with a client connection. The returned path will be represented as an absolute path.

```
char *getCurrentPath(const client_info_t *client)
```

6.2.2.16 Change the current path associated with a client

This function will change the current directory path associated with a client. The path specified as a parameter can be specified in either relative or absolute path format. In order to determine whether the newly specified directory path is valid, this function will call getDir() in order to identify whether a directory referenced with the new directory path exists. If so, the full_name field within the directory object will be stored in the client object as the new default path for the client. This function will return FAIL if the requested directory path does not point to a directory within the Status Server.

```
PASSFAIL changeCurrentPath(const char *new_path)
```

6.2.2.17 Remove a client and all of its relevant data

When a client disconnects from the Status Server, the client object and all of its related information must be freed up and removed. This requires that any client objects referenced in the touch_list of a directory object or data object be removed. In addition, all monitor objects associated with the client for both directory listings and data object monitors must be removed. As part of the monitor cleanup, this will require that monitor object references in the monitor_list and ls_list for directory and data objects also be removed. This is the processing which is performed by the removeMonitor() function.

```
PASSFAIL removeClient(client_info_t *client)
```

6.2.3 Data Services

This component is responsible for managing the data associated with the hierarchical structure within which Status Server directory objects and data objects are stored. Most of the functions manipulate information contained within the node_info_t data structure.

6.2.3.1 Create a directory or register the intent to remove a directory

This function is responsible for creating a directory if it doesn't already exist. If it does exist, the comment associated with the directory will be modified if the comment passed in to the function is not NULL. In order to remove a comment, an empty string must be passed to this function. Finally, a reference to the client object will be stored in the touch_list if it doesn't already exist.

```
void touchDir(const node_info_t *base,
              const char *path,
              const char *comment,
              const client_info_t *client,
              node_info_t **dir)
```

6.2.3.2 Retrieve a directory

This function will retrieve a directory object from the hierarchical Status Server directory structure. If the "create" parameter is set to TRUE, the directory will be created if it doesn't already exist. As part of the directory creation process, intermediate directories may also be created as part of the process. For example, if the Status Server only has a root directory "/" and this function is called to create /fits/455346o, both a fits directory and 455346o directory will be created. All directories will be created with a value of "DIRECTORY".

The address of the directory pointer will be returned from this function if it was successful. If the function returns FAIL, more details regarding the failure will be available in cfht_errno.

```
PASSFAIL getDir(const node_info_t *base,
                const char *path,
                const BOOLEAN create,
                dir_info_t **dir_p)
```

6.2.3.3 Remove a directory

This function will attempt to remove a directory object from the Status Server directory structure. The removal of a directory object is permitted if the client requesting the removal has initiated a "touchdir" request for the directory. Otherwise the function will return FAIL. The function will also return fail if the directory object contains any subdirectories. Detailed information regarding a failure will be available in cfht_errno.

If the operation is valid, an attempt will be made to remove each data object from the node_list associated with the directory object. This will be done by calling the rmObject() function with the check_touch parameter set to FALSE.

A check will then be made to determine if the directory object currently has a touch associated with it from another client, or if the size of the node_list is non-zero. If so, the directory object itself can not be completely removed. Otherwise, the directory object will be removed from the parent directory object node_list and deallocated.

```
PASSFAIL rmDir(node_info_t *dir,
               const client_info_t *client)
```

6.2.3.4 Add a data object to a directory object

This function adds an object to the linked list of objects associated with the directory object. This operation should not fail unless an object with the same object name already exists in the object list, or an attempt is made to add a data object to another data object instead of a directory object. Detailed information regarding a failure will be available in cfht_errno.

```
PASSFAIL addObject(node_info_t *dir,
                   const node_info_t *obj)
```

6.2.3.5 Remove a data object from a directory

This function will attempt to remove a data object from the linked list of objects associated with a directory object. If the data object currently has a touch associated with it from another client, or if the data object currently has any monitor objects within its monitor_list or ls_list, the data object will have its value changed to NONEXISTENT. Otherwise, this object does not have any external dependencies and will be removed from the parent directory object node_list and deallocated.

This function should not fail unless it is called on a directory object or if the client object is not contained within the linked list of clients who have performed a "touch" on this data object if the check_touch parameter is set to TRUE. Detailed information regarding a failure will be available in cfht_errno.

```
PASSFAIL rmObject(node_info_t *obj,
                  const client_info_t *client,
                  BOOLEAN check_touch)
```

6.2.3.6 Retrieve a data object

This function will retrieve a data object with a given object_name from the Status Server hierarchy. If a data object with the requested name can not be found, the function will return NULL. Otherwise a pointer to the data object will be returned. This function will iterate through the Status Server hierarchy as needed to find the target object.

If the "create" parameter is set to TRUE, the data object will be created if it doesn't already exist along with any required directories. If the "create_valid" parameter is set to TRUE and a data object is created, it will be created with a value of UNDEFINED. In addition, the client object will be added to the touch_list associated with the data object and the data object will be added to the touch_list associated with the client object. Otherwise, the object will be created with a value of NONEXISTENT.

```
node_info_t *getObject(const char *object_name,
                       const client_info_t *client,
                       const BOOLEAN create,
                       const BOOLEAN create_valid)
```

6.2.3.7 Set a directory comment

Convenience function to set the comment associated with a directory or data object.

```
void setNodeComment(node_info_t *node,
                    const char *comment)
```

6.2.3.8 Serialize Status Server contents to a file

This function will initialize the serialization process for the entire Status Server hierarchical directory structure and all the objects contained within the Status Server. Serialization will be performed by saving a series of transactions which can replayed by the message handler in order to reconstruct the Status Server directory structure and the object information contained within it. The messages will be identical to the "touch" and "touchdir" requests sent by a client with the exception that they will contain some additional parameters such as object value, value state, update timestamp, and creation timestamps for the objects.

During normal operations, the contents of the Status Server will be serialized at a predefined interval (probably 10 minutes). The Status Server contents will also be serialized is when a client requests a serialization. In both cases, a forked process will be used to save the contents of the Status Server to disk. The only time a fork will not be performed is when a client request a shutdown of the Status Server, or an error condition occurs requiring the Status Server to exit.

This function will initiate recursive processing to traverse the directory structure and initiate serialization for each directory object and data object within the directory structure.

Whenever the Status Server forks a child process to initiate a serialization, the PID of the child process will be saved. If a subsequent request for the Status Server to serialize itself is received before the previously spawned child process has completed, the previous child process will be "killed" and a new serialization will be initiated.

Once the serialization contents are successfully written to a file, the UNIX "mv" command will be performed to transfer the file contents to the target location. This step is used to prevent the creation of a corrupted or incomplete file from being written to the standard file serialization target location.

If a function should return FAIL, details will be available in cfht_errno.

```
PASSFAIL serialize(const char *path, BOOLEAN fork)
```

6.2.3.9 Populate a serialized touch string for a directory or data object

This component will populate a pre-allocated string buffer with a serialized "touchdir" or "touch" command used to restore the directory object or data object upon a restore request. If the function should return FAIL, details will be available in cfht_errno.

```
PASSFAIL getSerializeTouch(const node_info_t *dir,
                           char **buffer)
```

6.2.3.10 Create a data object

This function will create a data object and associate with it the name, comment and lifetime passed in to the function. Figure 15 contains the default values within the node_info_t structure when the data object is created. The create_valid parameter indicates whether the object should be created with a state of UNDEFINED in the case of TRUE or NONEXISTENT in the case of FALSE.

```
void createObject(const char *name,
                  const char *comment,
                  const time_t lifetime,
                  const BOOLEAN create_valid,
                  node_info_t **obj)
```

6.2.3.11 Add a monitor to a directory or data object

This function will add a monitor to the linked list of monitor objects (either monitor_list or ls_list). The is_node_monitor parameter will identify which monitor list the monitor object must be added to. This function should not return FAIL unless the monitor already exists in the linked list of monitor objects.

```
PASSFAIL addMonitorToNode(node_info_t *node,
                          const mon_info_t *mon,
                          const BOOLEAN is_node_monitor)
```

6.2.3.12 Remove a monitor from a directory or data object

This function will remove a monitor from the linked list of monitor objects (either monitor_list or ls_list) within a directory object or data object. The is_node_monitor parameter will identify which monitor list the monitor object must removed from. This operation should not return FAIL unless the monitor object does not exist in the linked list of monitor objects.

```
PASSFAIL removeMonitorFromNode(node_info_t *node,
                               const mon_info_t *mon,
                               const BOOLEAN is_node_monitor)
```

6.2.3.13 Retrieve a monitor from a directory or data object

This function will return a monitoring object from the linked list used to store monitors which have been placed on a data object. Each monitor object within the linked list will be checked to determine if it is associated with the client object passed as a parameter. If it is found, a pointer to the monitoring object will be returned. If not, the function will return NULL.

```
mon_info_t *getMonitorByClient(const node_info_t *node,
                               const client_info_t *client)
```

6.2.3.14 Inform monitoring clients of object changes

This function will go through the linked list of monitoring objects associated with the data object and determine whether the monitoring criteria has been met to cause a mailbox message to be sent out to the client. This function should not return FAIL unless it was executed on a directory node.

```
PASSFAIL informMonitorClients(const node_info_t *obj)
```

6.2.3.15 Setup a monitor listing for a directory

This function will set up a linked list of monitor objects to handle an “ls” command request from the client. This function will initiate calls to the createMonitor() function in the Monitor Services component to create monitors for each subdirectory and data object contained within the current directory object. This function should not return FAIL unless it was executed on an data object node.

```
PASSFAIL setupDirectoryListing(node_info_t *dir,
                               const client_info_t *client)
```

6.2.3.16 Refresh object state

Since some data objects may have a lifetime associated with them, it is important to update the state of the data object whenever the object becomes expired. This is critical if the object is being monitored. This function will check and, if necessary, update the value of a data object to EXPIRED. If the state is updated, the informMonitorClients() function will be called to handle potential client monitor updates. This function should not return fail unless it was executed on a directory node.

When an object goes from valid to expired, it must be removed from the time dependent object list by calling remove-TimeDepObject(). More details regarding the time dependent object list can be found in section 6.2.5.

```
PASSFAIL refreshObjectState(node_info_t *obj)
```

6.2.3.17 Check whether a directory or data object is valid

This function will return a boolean value indicating whether the value of a directory or data object is valid. This function will return TRUE if the value_state field is set to SS_VALID. Otherwise the function will return FALSE.

```
BOOLEAN isNodeValid(const node_info_t *obj)
```

6.2.3.18 Check whether a data object does not exist

For monitor purposes, it is necessary to create directory objects and data objects to support monitoring pointer integrity even if the data objects or directories did not previously exist. This function will return TRUE if the value_state field is set to SS_NONEXISTENT. Otherwise the function will return FALSE.

```
BOOLEAN isNodeNonexistent(const node_info_t *obj)
```

6.2.3.19 Retrieve the value of a directory or data object

This function will return the value of a directory or data object. This value may or may not be valid. In the case of a directory, this will be a string indicating the change count of the directory.

```
char *getNodeValue(const node_info_t *node)
```

6.2.3.20 Set the value of a directory or data object

This function will set a new value for a directory or data object. If the passed in value is set to NULL, the value of the node will be incremented if it is a directory and UNDEFINED if it is a data object.

In the case of a data object, informMonitorClients() will be called to determine if clients must be notified of a monitor change due to a change in the data object value. In addition, a check is made whether a non-zero lifetime was defined for the data object. If so, the repositionTimeDepObject() function within the Time Dependent Services component will be called to handle the positioning of this object within the linked list of time dependent objects.

If the value of a data object is changing from EXPIRED to a valid value, the addTimeDepObject() function will be called to add this object to the time dependent object list.

```
void setNodeValue(node_info_t *node,
                  const char *value)
```

6.2.3.21 Set the lifetime of a data object

Convenience function to set the lifetime of a data object. If the lifetime of an object is unlimited, it will be set to 0. This function should not return FAIL unless it was executed on a directory node. If the lifetime of the object changes from unlimited to a valid lifetime, the addTimeDepObject() function must be called in the Time Dependent Services component to add this data object to the linked list of time dependent objects. If the lifetime of the object changes from a valid lifetime to unlimited, it must be removed from the list of time dependent objects with the removeTimeDependentObject() function.

```
PASSFAIL setObjectLifetime(node_info_t *obj,
                           const time_t lifetime)
```

6.2.3.22 Get the lifetime of a data object

Convenience function to retrieve the lifetime of an object. If this function is called on a directory node, a value of 0 will be returned.

```
time_t getObjectLifetime(const node_info_t *obj)
```

6.2.3.23 Get the comment associated with a directory or data object

Convenience function to retrieve the comment associated with a directory or data object. This may be NULL if a comment has not been associated with the directory or data object.

```
char *getNodeComment(const node_info_t *node)
```

6.2.3.24 Set the full path of a directory or data object

This is a convenience function to set the fully-qualified absolute path of a directory or data object. Once an directory or data object is created, the path will never change, since it is not possible to move objects within the Status Server.

```
void setFullPath(const node_info_t *node,
                 const char *fullpath)
```

6.2.3.25 Get the full path of a directory or data object

This function returns the fully-qualified absolute path of a directory or data object. The path associated with a directory or data object should never be NULL.

```
char *getFullPath(const node_info_t *node)
```

6.2.3.26 Remove a node

This function will free up all the resources associated a directory or data object and remove it.

```
PASSFAIL removeNode(const node_info_t *node)
```

6.2.4 Monitor Services

This component is responsible for managing the data associated with the Status Server monitors. Most of the functions manipulate information contained within the mon_info_t data structure.

6.2.4.1 Create a monitor object

This function will create a monitor object. Figure 16 contains the default values within the mon_info_t structure when the monitor object is created.

The type of monitor object which is being created will be identified with the is_data_monitor. Monitor objects can be created to support either client requested object monitors or directory listings. Once a monitor object is created, the client services component and data services components will be used to ensure that the monitoring object is properly associated with the client object and data object. If the return value of the function indicates that the call failed, more details will be available in cfht_errno.

```
PASSFAIL createMonitor(node_info_t *object,
                       client_info_t *client,
                       const double deadband,
                       const boolean is_data_monitor)
```

6.2.4.2 Update the deadband threshold for a monitor object

This function will update the deadband threshold for an existing monitor object.

```
void updateMonitorDeadband(mon_info_t *mon,
                           const double deadband)
```

6.2.4.3 Record the object value which has been sent to a client

This function will be called whenever a monitor update is sent to the client. It will set the value sent to the client as well as the prev_sent_ts timestamp indicating when the client was sent the monitoring information.

```
void setSentData(mon_info_t *mon,
                 const char *value)
```

6.2.4.4 Check whether a client must be notified of a monitoring change

This function will return a boolean value indicating whether the current value of an object has changed enough to warrant a client notification.

```
BOOLEAN notifyClient(const mon_info_t *mon)
```

6.2.4.5 Remove a monitoring object

This function will free up all the resources associated a monitoring object and remove it. Before a monitor object is removed, references to the monitor object must be removed from the monitor_list and ls_list in the node_info_t object and ls_list and monitor_list in the client_info_t object. This operation should not fail, but if it does, more details regarding the failure are available in cfht_errno.

Although the functionality is not contained within this function call, after a monitor is removed, a check should be made whether the data object being monitored can be removed. Once a monitor is removed, the data object can also be removed if its value is NONEXISTENT and if the size of its touch_list, monitor_list and ls_list are all zero. In this case, the object is already marked as non-existent and does not have any other external references to it.

```
PASSFAIL removeMonitor(mon_info_t *mon,
                       const BOOLEAN is_data_monitor)
```

6.2.5 Time Dependent Services

This component is responsible for managing the the data associated with time dependent objects. A time dependent object is defined as a data object, whose state will change depending upon time. Any data object which has a specified lifetime is considered a time dependent object. The functions in this component manipulate the linked list of time dependent objects.

6.2.5.1 Add a data object to the list of time dependent objects

This function will take a data object and try to add it to the linked list of time dependent objects. A check will be made whether it is a data object and not a directory and whether the lifetime associated with the data object is not set to zero. Both cases must be true. Also, this function will make sure the object does not already exist in the linked list. If the object already exists, or the previous checks failed, this function will return FAIL. When the object is inserted in the list, it will be added in descending order based on the expiration time of the object.

```
PASSFAIL addTimeDepObject(const node_info_t *obj)
```

6.2.5.2 Remove a time dependent object from the list of time dependent objects

This function will take a time dependent object and perform a search to see if the data object already exists in the linked list of time dependent objects. If so, it will be removed from the list. If not, the function will return FAIL.

```
PASSFAIL removeTimeDepObject(const node_info_t *obj)
```

6.2.5.3 Service time dependent objects

This function will iterate through each node in the linked list of time dependent objects which calculations show should have changed to an expired state. The state of the object will be refreshed by calling refreshObjectState(). This function will handle calling removeTimeDepObject() if the object has changed to an expired state as well as triggering client monitor notification processing if needed. Since the list should be maintained in a sorted order by objects which should be the first to expire at the top, it will only be necessary to iterate through the list until an object has time remaining before it expires.

```
void serviceTimeDepObjects(void)
```

6.2.5.4 Reposition a time dependent object

Whenever a time dependent object is modified, there is a good chance that it must be repositioned within the linked list. This is because based on its lifetime, the time to the next possible expiration of the object may require a different positioning within the list which is sorted in descending order based on the expiration time of the object. This function should not fail unless it is called on a directory object or if the data object is not contained within the list of time dependent objects.

```
PASSFAIL repositionTimeDepObject(const node_info_t *obj)
```

6.2.5.5 Retrieve the minimum time to next update

This function will return the number of seconds before another item within the list of time dependent objects must be serviced. This time interval will be used as part of the calculation to identify the timeout value to call the sockserv_run() function in the sockio library. This time interval becomes the timeout used to call the underlying socket select function call. This time value can be calculated by determining the time to expiration of the first item in the linked list. If the list does not contain any time dependent objects, a predefined constant time value will be returned.

```
int getMinimumTimeoutInterval(void)
```

6.2.6 Utility Functions

This component contains utility functions which may be used by one or more Status Server software components. Some of the functionality which will be provided by this component include:

6.2.6.1 Memory Allocation Wrapper Functions

Memory allocation in the Status Server will be handled in such a way that a call to allocate memory will never fail. Wrapper functions will be provided around malloc() and realloc() to prevent a calling function from receiving a NULL pointer indicating that memory could not be allocated. The wrapper functions, ssMalloc() and ssRealloc() will perform an underlying call to malloc() and realloc(). However, if either of these system calls returns a NULL value, the function will sleep for a predefined time interval (maybe a second or so) and retry. The function will not return back a pointer until memory is available. Since the Status Server is single-threaded, this means the server will block until memory becomes available.

Some analysis regarding some of the alternative approaches to handling out-of-memory conditions is included in the Design Analysis section at the end of this document.

6.2.6.2 Linked List Functions

The node_info_t and client_info_t data structures each contain fields identified as linked lists. A complete set of functions will be provided to manipulate the linked lists. The linked lists in the Status Server will be doubly linked and optionally sorted upon insert.

6.2.6.3 Argument Parsing and Validation Functions

When a client request is received across the socket interface, it must be checked to determine if it is a valid URL encoded string and, if so, it must be parsed according to its commands and arguments. Parsing functions already exist within libcli to handle some of the basic argument parsing required.

6.2.6.4 Logging Functions

Error and debug logging will be handled through the logging functions within libcfht. Convenience functions to handle the formatting of debug and error information before cfht_logv is called may be included here.

7 Design Analysis

Throughout the design process, there were alternative approaches which were analyzed in order to decide the final design choice. This section addresses some of the alternatives and the final decision.

7.1 Out-of-Memory Handling in the Status Server

Several of the components and functions in the Status Server will require memory to be allocated or reallocated as part of its functionality. It is possible for calls to malloc() or realloc() to fail in the unlikely event that memory is not available on the machine. While this event hopefully will never happen, it is an error condition which must be managed. This section addresses the issue and some possible alternatives. A decision must still be made on the approach to be implemented in the Status Server. The API calls and pseudocode documented in the software design don't take into account error conditions caused by the inability to allocate enough memory.

7.1.1 Alternatives for Handling Out-of-Memory Condition

In many cases, the memory allocation will occur as a result of a client request. In this case, it could be possible to send a return message to the client indicating that the request could not be processed. Other times, the memory allocation could occur when a client must be informed of a monitoring event and space can not be allocated in the client monitoring object to hold the new value. At this point, it could be possible to defer the monitor update by treating this a the same case as a full network buffer. Using the error message approach to handling "out-of-memory" errors, the Status Server would continue to operate in a somewhat degraded mode until additional memory becomes available. This would require the client to handle additional failed return values. A basic flow diagram for this approach is shown in figure 18.

Another alternative would be to have the Status Server add retry logic to both the malloc() and realloc() function calls. In this case, if memory was not available, it would sleep for some period of time before retrying. Since the Status Server is single-threaded, this means the entire Server process will block while memory is not available. Also, any client request currently in progress will block causing the client socket connection to remain tied up until the client-side socket timeout threshold is reached. This approach would eliminate the need to check for out-of-memory conditions simplifying the Client API and Status Server along with eliminating one possible return value the client may need to check for. The disadvantage is that the connection may remain tied up or time out. Hopefully, the process gobbling

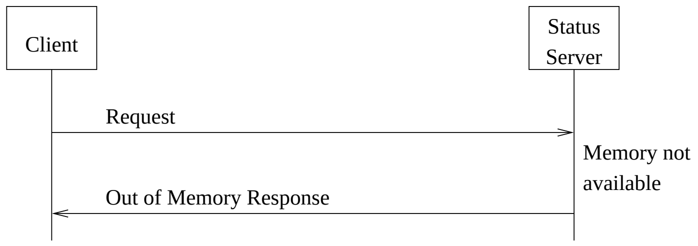

Figure 18: Status Server Memory Allocation Failure Error Message Flow

up the memory is not the Status Server itself. If so, this approach would cause the Status Server to slowly consume more and more memory. If the memory is being consumed by another process, it is possible that the process will soon die and normal operation can be quickly resumed. If a core dump of the process is triggered, it could take some time before the memory again becomes available. A basic flow diagram for this approach is shown in figure 19.

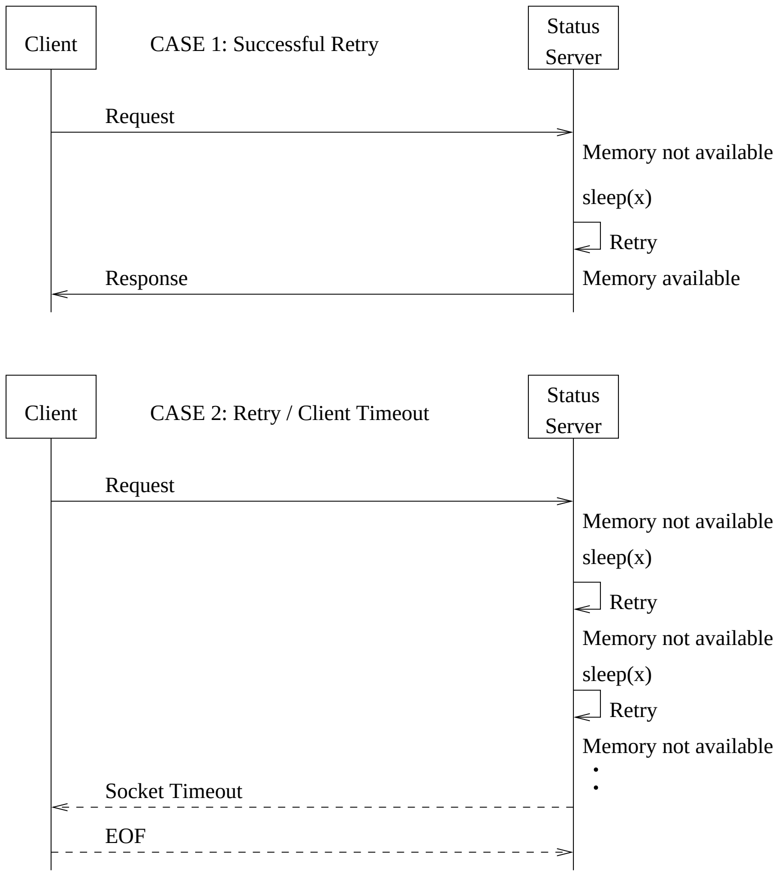

Figure 19: Status Server Memory Allocation Failure Retry Message Flow

A final alternative could be to initiate a predefined number of retries and if the memory can not be allocated when retries are exhausted, serialize the contents of the Status Server and trigger an exit. This approach would require that buffers required for serialization be preallocated and that the IO processes used to write the data to a file do not require additional memory allocation. The total time available to retries must be set below the default socket timeout defined for socket connections. In this case, the client would not need to implement any additional error checking for individual command responses, and it will be guaranteed not to block if it has connected to the Status Server without the autoretry option. For clients connecting to the Status Server with the autoretry option, they will block until the

Status Server is restarted. A basic flow diagram for this approach is shown in figure 20.

There is a good chance that this alternative would not work, since there is a good possibility that the IO processes used to write the data to a file will require an allocation of memory.

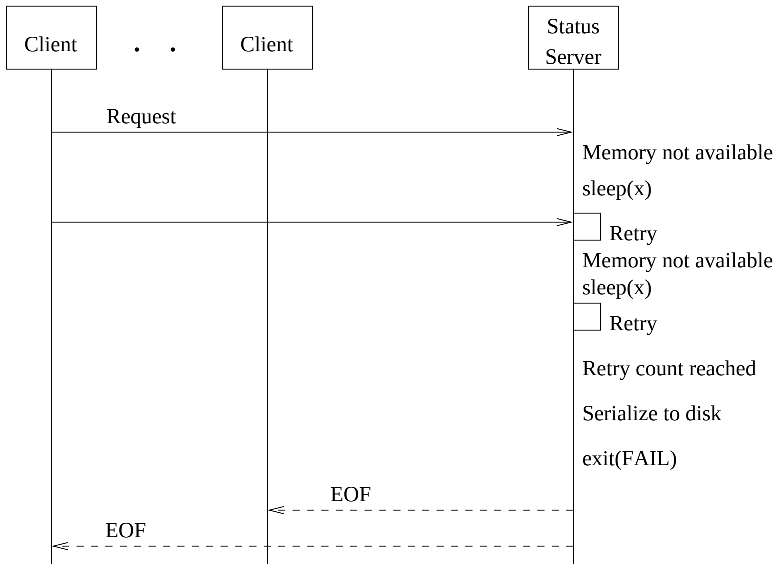

Figure 20: Status Server Memory Allocation Failure Retry and Exit Message Flow

7.1.2 Preferred Approach to Handling an Out-of-Memory Condition

The Status Server will implement the second approach, which requires the Status Server to add retry logic to both the malloc() and realloc() function calls. In this case, if malloc() or realloc() function calls fail, the Status Server will sleep for a predefined time before retrying the memory allocation.

8 Document Change Log

Version	Date	Comments
1.0	May 10, 2002	First release for review.
1.1	May 28, 2002	Revised document based on comments from S. Isani and J. Thomas.