Volume 5

Hyperwave API Definition

Hyperwave Information Server

Version 4.1

HYPERWAVE
Organize your future

Hyperwave HW API Definition

Hyperwave
Information Management Inc.
2350 Mission College Blvd.
Santa Clara, CA 95054
USA
Tel: 1-408-982-8228
Fax: 1-408-982-8229
Email: info@hyperwave.com
http://www.hyperwave.com

Hyperwave
Information Management GmbH
Stefan-George-Ring 19
D-81929 Munich
Germany
Tel.: ++ 49 89 993074-0
Fax: ++ 49 89 993074-99
Email: info@hyperwave.de
http://www.hyperwave.de

Document Version 1.0

TABLE OF CONTENTS

1 OVERVIEW

1.1 WHAT IS THE HYPERWAVE INFORMATION SERVER API?

The Hyperwave Information Server API is targeted at experienced users with programming skills who want to add their own set of functionality to the server. It provides access to the server's operation by providing a set of functions which cover the server's capabilities at a relatively high level. Additionally, it provides a set of specific Hyperwave Information Server data types along with their common access methods. An example is the function "children", which is used to retrieve the descendants of a collection and return a list of Hyperwave objects. The list, as well as the Hyperwave objects and the access methods, are defined by the API.

The API definition itself is as language-independent as possible and is object-oriented in a simple manner. In a program written in terms of the Hyperwave Information Server API there will most likely exist one central object containing and implementing the Hyperwave functionality. This object will be surrounded by many data objects with well-defined access functions, initialization and cleanup routines.

This API definition serves as the basis for future language mappings. A language mapping is a concrete one-to-one implementation of the formal definition in terms of the given language. Thus the recommended way to get started with the API is to first read through the formal API definition to become familiar with the concepts, and then to jump into detail concerning the language.

1.2 OVERALL STRUCTURE

The API is conceptually divided into several modules, the most central being the Hyperwave Information Server access module. It provides several methods of interoperation with the server, such as:

- opening a session (the object constructor, to stick with the terminology)
- identification
- getting a Hyperwave object by its unique identifier
- getting the children of a collection
- inserting a document

just to mention a few.

Another module is the collection of data types used by the Hyperwave Information Server access module. It consists of about a dozen data structures and utilities needed to interoperate with the server and to manipulate requests and results. Some of the data structures are almost trivial and have native implementations in the host language; nevertheless, it is necessary to define them since their interface has to be uniform across different language mappings. However, language mappings are free to leave out one type or the other if their native counterparts are comparable.

The most basic type defined by the API is a string type. This type is most likely to be omitted by a particular language mapping. Built upon the string type, there is a Hyperwave Information Server object representation which is a set of name-value pairs. Also, there is a data type "set of Hyperwave objects", which can be operated on using, for example, a "sort order specification" object.

A more abstract concept is the representation of document content. The content object is designed as a stream of data (its core operation is "read") and hides from the user its internals which can be anything from a network connection to in-core memory. Since it is necessary to pass

document content from the user to the server as well, this type requires a certain amount of extensibility.

Last but not least, there is an error type which is (hopefully not) returned by every call to the Hyperwave Information Server access module. This type implements a rather sophisticated error tracing scheme and will be explained in detail later.

The sole purpose of another module is to pass parameters to the Hyperwave Information Server access module and back out. The Hyperwave Information Server calls have the following uniform form:

```
out = request (in)
```

where `in` is an object containing the input parameters and `out` is an object containing the result(s). An in/out pair is defined for every function in the Hyperwave Information Server module. This parameter passing method was chosen because of its flexibility (parameters can be added without changing the interface) and to avoid long parameter lists.

1.3 LIST OF MODULES

More formally, the list of modules that participate in the Hyperwave Information Server API are as follows. Once again, the definitions of the structures and their operations are not necessarily binding on an implementation in a particular language. Rather, if the language supports better mechanisms to achieve a given functionality, the language mapping is free to prefer these mechanisms.

1.3.1 DATA TYPES

The data types defined and used by the Hyperwave Information Server API are

String

A string class.

String Array

A collection of **String** objects. Supports merging, iteration.

Attribute

A name-value pair of strings. The value is a string array. An **Attribute** corresponds to a name-value pair of a Hyperwave object, such as "`Name=xxx/yyy`". Multiple name-value pairs with the same name result in multiple entries in the value string array.

Object

The more or less direct counterpart of a Hyperwave object. A set of **Attributes**.

Object Array

A set of **Objects**. Supports roughly the same operations as a string array, plus has a stub for sort order specification objects.

Error

Represents an event recording scheme. The events recorded are errors, warnings and messages. It is returned by every API function.

Content

Represents any kind of data that is passed from the user to the Hyperwave Information Server or from the server to the user. It has a stream-like interface which supports reading, checking for eof and error. It also records the amount of data which has been read already. **Note:** unlike the other data types, copying a **Content** object does not

actually copy all the data. In other words, if one copies a **Content** object which has been read from already, a subsequent read operation on the copied object starts at the part which has not yet been read.

1.3.2 OPERATORS

The Hyperwave Information Server API defines some "operators" that are used to manipulate data objects or customize the behavior of certain API functions.

Attribute Selector

This is not a real type, but rather a special use of a string array. However, it is listed here because it is an important parameter to most of the API functions. A string array is used as an input parameter to most API functions to request additional information the output objects should contain in the form of dynamic attributes. This information is not generated by default for server performance reasons.

Sort Parameters

Object Array objects are unordered by default, i.e., their sort order is unspecified (this is not true for object arrays that are API function return values; they get a default sort order, but more on that later). Sort order specification objects are used to control the sorting operation which is provided by the object array.

1.3.3 API FUNCTION PARAMETERS

Many of the API functions described below accept a great number of parameters, many of which are optional. Also, the API definition is expected to be extended a great deal in the future. Furthermore, the return parameters are not just a single item per API function, but rather comprise an error (the one from the data types section) as well as regular result data members (such as a set of **Objects**).

To avoid long parameter lists and to remain extensible, we decided to define one input parameter set and one output parameter set per API function. An API function thus accepts as input parameter an object which contains the input to the function, and passes as result an object which contains the output.

The concrete interface definition is provided below in the Interface Definition chapter.

1.3.4 API OBJECT/FUNCTIONS

Depending on the language and the environment the API functionality is implemented in, there are several implementations of the API. For example, one API implementation sits directly on top of the HG-CSP, the TCP based protocol that is used by server-side client programs such as the WWW gateway. Another possible implementation uses HTTP as a transport medium. No matter what the implementation is, all support the same set of API functions.

2 NAMING SCHEME

In order to make the Hyperwave Information Server API look uniform and to make it easily portable across different language mappings, a naming scheme is necessary.

A general rule in the Hyperwave Information Server API naming scheme is that everything is prefixed with the string `HW_API_`.

The rest of the rules are explained below.

2.1 DATA TYPES

The names of the Hyperwave Information Server API data types are capitalized and prefixed as stated above. If the type name is composed of two words, then the first letter of each of these words is capitalized. For example, the **String** type is called `HW_API_String`. The **String Array** type is called `HW_API_StringArray`.

Another data type rule is that the name of a method of a data type is written in lowercase letters, prefixed with the name of the data type itself, postfixed with the parameter type names without their `HW_API_` prefix, separated by underlines ('_'). For example, the **String** `append` method that is used to append another **String** object would read `HW_API_String_append_String`. The **String** assignment method would read `HW_API_String_copy`.

Note that this is only a guideline. Object-oriented languages provide implicit scoping and assignment where all that prefixing and postfixing is unnecessary. However, in a C API implementation such a scheme would be essential.

2.2 OPERATORS

The API operator naming scheme is exactly the same as for the API data types.

2.3 API FUNCTION PARAMETERS

The API function parameter naming scheme is similar to the data type naming scheme in that the methods are named the same as there.

There are two parameter types for each API function - an input parameter and an output parameter. These are named after the API function, with the obligatory `HW_API_` prefix. The postfix is `_In` for the input parameter, and `_Out` for the output parameter.

For example, the parameters for the API `children` function are called `HW_API_children_In` and `HW_API_children_Out`, respectively.

2.4 API OBJECT/FUNCTIONS

Depending on the language used, there may be multiple choices for implementing an API object. These choices need not be made visible to the user as separate types. Rather, the choice of implementation should result in a parameter to an API object creation function. Thus, in a proper language mapping, the API object type will simply be called `HW_API.`

The API functions (the functionality the `HW_API` type provides) are prefixed with "`HW_API_`" and then continue with their own names, starting lowercase. For example, the "identify" function would read "`HW_API_identify`".

Prefixing the API functions with "`HW_API_`" is omitted in object-oriented languages.

3 INTERFACE DEFINITION

3.1 DATA TYPES

Following is the list of data types defined by the Hyperwave Information Server API. The members of the types are left as language-independent as possible: we are limiting ourselves to describing the concepts. As a consequence, for example, iterator functions which accept index arguments as their parameters do not specify at which position the index starts. It is left to the implementation to define the start index (in C one would expect indexes to start at 0, while in Pascal (a rather hypothetical example) it is expected to start at 1).

3.1.1 STRING

A **String** is supposed to be an opaque data type (an anonymous block of bytes). Its length can easily be determined. Assignment is supposed to be cheap, just as is the appending operation.

One of the major uses of the string data type is to pass it as object identifier input parameter to many of the API functions. There it serves as either the global object ID (the "GOid" attribute) of the per-server-unique object name (the "Name" attribute). For a detailed explanation of the semantics of `GOid` and `Name` see 3.3.

assign

The **String** copy function. A particular language mapping should also provide its variant to copy its native strings onto a **String** object.

length

Returns the length of the string.

string

Returns the native representation of the string.

append

Appends to the **String** object. A particular language mapping should also provide its variant to append its native strings to a **String** object. Also, appending a single byte should be supported.

concat

Returns a concatenation of two **String** parameters. Native variants which return a **String** should also be provided.

3.1.2 STRING ARRAY

A string array is a collection of strings. The elements are unique, that is, there cannot be two instances of the same string in a string array. The elements can be enumerated with an integer value.

assign

The **String Array** copy function.

count

Returns the number of elements (strings) in the array.

string

The **String Array** iterator function. This function accepts as parameter an integer value, the iterator, which enumerates the elements in the array. It returns a string if there is a corresponding string in the array, and an error otherwise (the iteration is finished).

insert

Inserts a string into the array. Returns an error if the string is already an element of the array.

remove

Removes a string (the only function parameter) from the array. Returns an error if the string was not found in the array.

merge

Returns a string array produced by the union of two string array parameters.

3.1.3 ATTRIBUTE

An attribute is a name-value pair, with the name (key) being a string and the value being a string array.

assign

The **Attribute** copy function.

key

Returns the key of the **Attribute**.

values

Returns the value part of the Attribute, a string array.

3.1.4 OBJECT

An **Object** is a set of **Attributes**. It can be queried for an **Attribute** matching a given key. Also it is possible to iterate over the attributes.

assign

The **Object** copy function.

count

Returns the number of attributes in the object.

insert

Inserts the parameter, of type **Attribute**, into the object. If an attribute with the same key is already a member of the object, its values are added to the list of values corresponding to that key.

remove

Accepts as parameter a string which is the key of the attribute to be removed. Returns an error if the attribute is not found.

attribute

Used to retrieve attributes. You can retrieve attributes in two ways. First, you can iterate over them much like you do with the string array. Second, you can retrieve them by their keys. Thus there are two forms of this function:

- **iterator**: This function accepts as parameter an integer value, the iterator, which enumerates the elements in the object. It returns an attribute if there is a corresponding attribute in the array, and an error otherwise (if the iteration ends without finding the corresponding object).
- **key**: This function accepts as parameter a string, which acts as the key of the attribute to be retrieved. If there is an attribute with that key in the object, it is returned, otherwise an error is returned.

3.1.5 OBJECT ARRAY

An **Object Array** is a collection of objects as defined above. Its interface is quite similar to the **String Array** interface, with the exception that the array is not concerned about object equality. This means that you are allowed to insert the same object twice (though this is of course not recommended).

assign

The **Object Array** copy function.

count

Returns the number of elements (**Objects**) in the array.

object

The object array iterator function. This function accepts as parameter an integer value, the iterator, which enumerates the elements in the array. It returns an object if there is a corresponding object in the array, and an error otherwise (if the iteration ends without finding the corresponding object).

insert

Inserts an **Object** into the **Object Array**.

sort

This operation accepts as parameter a sort order specification object (see 3.2.1 for a detailed description of the sort order string syntax and the other members of that operator).

If an object array is the result of a Hyperwave Information Server API function (that is, its origin is the Hyperwave Information Server we are talking to), it always has a well-defined sort order, which is referred to here as the default sort order. In the context of this API, this sort order is either of a “weak” or a “strong” type. For example, the **Object Array** might be the result of the “children” function, i.e. it consists of the children of a collection. This collection listing might stem from a collection that has a **SortOrder** attribute, in which case the default sort order is strong. However, if the collection does not have the **SortOrder** attribute, it has the default sort order "#T", and this sort order is weak. Note that at this time there is no default language string set, so if you want to sort language-dependent attributes such as **Title**, you have to specify the language priorities in a sort parameters object you pass to this operation.

The sort order contained in the sort order specification object can also be specified as being weak or strong. This affects the sort order imposed on the **Object Array** as follows:

- If the sort order in the sort order specification object is strong, then this sort order is imposed on the **Object Array** whether its default sort order is weak or strong.
- If the sort order in the sort order specification object is weak, then
  - this sort order is imposed on the **Object Array** if the default sort order is weak

- it is overridden if the **Object Array** has strong default sort order

- If you pass an empty sort order in the sort parameter object, the default sort order of the **Object Array** is always used whether it is weak or strong.

If you make an object array from scratch by appending objects it will have an empty default sort order, and thus its sort order will be undefined.

3.1.6 REASON

A Reason represents a message (error/warning/message) from a module which is involved in a call to the API.

A Reason object is identified by the following characteristics:

- **type**: This can be either `Error`, `Warning` or `Message`.
- **source**: This is the unique identifier of the module that created the reason. This module is located inside the Hyperwave Information Server or the Hyperwave Information Server API. Every module is registered in a database and has a well-defined set of possible errors/warnings/messages which are registered as well.
- **code**: This is the error/warning/message code local to the module which created the Reason.

Its operations are

assign

The reason copy function.

type

Returns the type of the reason, i.e., `Error`, `Warning` or `Message`.

source

Returns the identifier of the module that created the reason.

code

Returns the code of the reason.

description

Accepts as parameter a two-character string which is the language identifier as used by the Hyperwave Information Server (ISO 639). Returns a human readable description of the reason object.

3.1.7 ERROR

An error is an ordered list of reasons that were collected from the different places/modules a HW API call passed through. The name "Error" is a bit misleading in that an error contains reasons which are not necessarily of type `Error`. Rather, if an error occurred, the last element in the list of reasons is guaranteed to be of type `Error`. All other reasons, if any, are **not** errors.

The error type is returned by every API function. The user should check if the function really failed by checking if the error's last element is of type `Error`.

count

Returns the number of reasons that were recorded.

error

Returns the error if one occurred, else false. In other words, if the last element in the reason list is of type error, this element is returned, otherwise the function returns false.

reason

The error's iterator function. The function accepts as parameter an integer value, the iterator, which enumerates the elements in the list. It returns a reason if there is a corresponding reason in the list, else an error (if the iteration ends without finding the corresponding object).

3.1.8 CONTENT

A content is a rather abstract concept. It represents document content in a general manner. Its interface is stream-like. That is, one is only allowed to read blocks of data from it. No positioning (searching) operation is permitted because the underlying medium can be anything which can hold document content, such as in-core memory buffers, disk files, network connections (which do not support random positioning) and concatenations of any of these.

An important thing to note is the content's copying semantics. As opposed to the other Hyperwave Information Server API data types, copying a content is defined as only passing around the handle to the content. The content itself is not copied. This means that if one copies a content object, then reads from the original, a subsequent read from the copied content object will **not** get the same data again, but the data after the first read. In other words, the content is reference counted.

assign

The content copy function.

read

Reads a block of data. Accepts as parameters the number of bytes to be read, and, depending on the language we are using, a location where to store the data. The number of bytes to be read must be greater than zero. Implementations may choose different ways to define the interface of the read function and its return values/types. However, it is good practice for an API program to check for eof or error after a read.

length

Returns the total length of the content, or -1 if the length is unknown.

consumed

Returns the number of bytes that were consumed (read) already.

eof

Is the end-of-file reached already?

error

This returns an error type as defined above.

3.2 OPERATORS

3.2.1 SORT PARAMETERS

A sort parameter is used in conjunction with the object array data type. A sort parameter object has three members, each of which has a "strength" value associated with it. For a detailed description of its use with the object array data type see there, also for the semantics of the strength of the particular members.

assign

The sort parameter copy function.

setSortOrder

A string containing a sort order as described below.

setLanguages

A string containing language specifiers separated by any non-alpha characters.

setTypeInfo

An object which is used as a lookup. It maps attribute names to type specifiers. The type specifiers are described below.

The sort order string can have two possible formats. The first one is simply a concatenation of characters as follows:

A	Author
C	TimeCreated
E	TimeExpire
O	TimeOpen
N	Name
S	Score
t	Type (order: collection, cluster, text, image, film, audio, scene, PostScript, generic, annotation, remote)
T	Title
#	Sequence number
-	Use the reverse of the next stated sorting criterion

With this sort order type the sorting operation is rather hardcoded. The type information passed to the sort parameter object is ignored; the different characters given are converted as expected (**Author** is a string with the usual lexical comparisons, as is **Name**). The languages passed are applied only to the **Title** attribute.

Example: `#-T` causes objects to be sorted primarily by sequence number. Those objects which have the same sequence number are sorted by title. Because the title is language dependent (this decision is **not** taken from the type information object which is described below, but is hardcoded), the title's two-character language prefix is compared using the order of the languages in the language string (the order is undefined if the language string is empty). If there are objects which have titles with the same language, these are sorted by the rest of the title alphabetically.

The second format of the sort order string is somewhat more difficult to understand, albeit more powerful. It consists of one or more groups of three values, separated by colons and surrounded by parentheses:

'('<AttributeName> ':' <SortType> ':' <Order>')'

The AttributeName is the name of the attribute by which you want to sort the object array. SortType should be left blank because type information is extracted from the type information object. The only exception is the single character 'E' as a SortType, which stands for "enumeration". Order is '+' or '-' for all types which are extracted from the type information object, meaning "ascending" and "descending", respectively. For the enumeration type it is any sequence of strings, separated by a semicolon (';'), interpreted as prefix of the attribute values to be sorted.

The values of the type information object are currently interpreted as follows (to be extended):

String	Values are compared using the usual lexical comparisons
Integer	Values are compared as integer numbers
Float	Values are compared as floating point numbers
Time	Values are assumed to be in Hyperwave Information Server time format, and are compared that way
LangKeyword	Values are supposed to have a language prefix, and are compared primarily using the language string parameter. Two values are compared as follows. First, their language prefix is compared using the order given in the language string. Note that it is not enforced by the Hyperwave Information Server that a LangKeyword have a language prefix; if one does not, the order is undefined. If the language prefixes are equal, the order is determined by the alphabetical order of the rest of the title.

The sorting algorithm interpreting this format is best explained using an example.

The prerequisites are

- a type information object with the following information:

  Title is of LangKeyword type

  Score is of Integer type

- a language string "en,ge,fr"
- a sort order string "(Score::+)(Title::+)(YesNo:E:y;n)"

With this sort order string the sort algorithm proceeds as follows. The groups surrounded by braces are evaluated sequentially left to right, each refining the "buckets" that result from evaluating the previous group.

"(Score::+)" divides the input objects into buckets according to their Score attributes, with the objects that have the same Score residing in the same bucket. (Score values are compared using integer comparison, as has been looked up in the type information object.) All the buckets are then sorted in ascending order.

"(Title::+)" iterates over the Score buckets from the previous step. It dives into each of them and sorts its contents by the Title attribute in ascending order. The Title attribute is interpreted as a LangKeyword, thereby consulting the language string. If an object has a "en" title, that one is taken for comparison. Else, if it has a "ge" title, that one is taken. Else, if it has a "fr" title, that one is taken. If one object contains titles, but none of them has one of the languages of the language string, an arbitrary title is chosen (it is undefined which one). If an object contains no title, the title comparison operation of that particular object is undefined (this is, its final position within the current bucket is unpredictable).

Finally, if there are still ambiguous objects ("buckets with more than one element"), the sort order string component "(YesNo:E:y;n)" orders them according to the YesNo attribute which has an enumeration type with the two possible values "y" and "n". As stated above, these values are considered prefixes of the attribute's value, so the YesNo value can as well be "yes" or "no".

3.3 API FUNCTIONS AND PARAMETERS

The following is a list of the API functions together with their input and output parameters. The input and output parameters are containers of the actual function parameters. Their content is listed with the member name and the member type, plus an optional explanation.

One word about the object identifier semantics. Most of the functions described below accept as their primary input parameter an object identifier which is basically a string. This identifier is twofold:

- It can be the `GOid` attribute of a Hyperwave Information Server object. This is the preferred case since it addresses the object directly. Moreover, a *G*lobal *O*bject *id*entifier is guaranteed to be unique across server boundaries, which is relevant to if you intend to talk to a Hyperwave Information Server which is part of a Hyperwave server pool.
- It can be the `Name` attribute of a Hyperwave Information Server object. Most API functions, if they get passed an object's name, will perform an index query, possibly resulting in inefficiency. Also, as stated above, a Name attribute is not guaranteed to be unique across server boundaries.

The essence of these semantics is that you should use the global object identifier wherever you can get it, and use the object's name only as an entry point. For example, if you are writing a program that traverses a collection tree beginning at a given starting point, you will most likely have no global object identifier as the starting point identifier, simply because users tend to prefer to remember names rather than numbers. You pass the name of the starting point to the first `children` call, and for all further calls you use the GOid attributes of the descendants of the starting point.

3.3.1 IDENTIFY

Input

- **username**: string.
- **password**: string.

Output

- **error**: error.

Identifies according to username and password. The output is an error object.

3.3.2 USER

Input

- **attribute selector**: string array.

Output

- **error**: error.
- **object**: object.

Retrieves the user object (if available) of the current user. If no object is available for whatever reason, a synthetic object is generated (check for eventual warnings or messages).

3.3.3 CHILDREN

Input

- **object identifier**: string.
- **attribute selector**: string array.
- **object query**: string.

Output

- **error**: error.
- **objects**: object array.

Retrieves the children of a collection, as well as optionally applying an object query on the returned objects.

3.3.4 PARENTS

Input

- **object identifier**: string.
- **attribute selector**: string array.
- **object query**: string.

Output

- **error**: error.
- **objects**: object array.

Retrieves the parents of an object, as well as optionally applying an object query on the returned objects.

3.3.5 OBJECT

Input

- **object identifier**: string.
- **version**: string.
- **attribute selector**: string array.

Output

- **error**: error.
- **object**: object.

Retrieves an object. If a version is given, and if the object is version controlled, the given version of the object is retrieved.

3.3.6 MOVE

Input

- **object identifier**: string.
- **source parent identifier**: string.
- **destination parent identifier**: string.

Output

- **error**: error.

Moves an object from the source parent collection to the destination parent collection.

3.3.7 LINK

Input

- **object identifier**: string.
- **destination parent identifier**: string.

Output

- **error**: error.

Makes an existing object a child of the destination parent collection. This operation only concerns the parent-child relation. The object itself is not touched.

3.3.8 COPY

Input

- **object identifier**: string.
- **destination parent identifier**: string.
- **attribute selector**: string array.

Output

- **error**: error.
- **object**: object.

Physically copies an object (together with its content, if any) into the destination parent collection. If successful, returns the newly created object.

3.3.9 REMOVE

Input

- **object identifier**: string.
- **parent identifier**: string.

- **mode**: a combinataion of the values below.
- **object query**: string.

Output

- **error**: error.

Removes an object from the specified parent. If the object is a collection, the collection is removed recursively if mode and object query allow that. If an object query is given, only those objects are removed that match the query.

Mode is combination of:

- `normal`: the default; it can be combined with removelinks, and is overridden by physical. A child object which is to be removed is only removed from the parent collection over which the recursion hit the object. If the object is also a child of another collection, it will not be removed but rather the parent-child relation is split up.
- `physical`: quite the opposite of `normal`. That is, objects that have multiple parents are removed from all parents.
- `removelinks`: all references to an object are deleted as well. That is, if a source anchor refers to the object it is deleted (if the access rights permit that). If the object has a source anchor which points to a destination anchor which will not be referenced anymore after the deletion of the object's source anchor, it is deleted (again, if the access rights permit that).

Due to the relative complexity of the remove function it is recommended to check for warnings. If nothing really bad happens, the call will return ok, although not all objects were deleted. See, for example, the access rights situation when using the removelinks call, which of course applies to recursive removals in general (you will receive a warning for every object on the way down which refuses to be removed). One more example of a warning is when you remove recursively and together with an object query. This may leave some objects in a collection which should be removed, but cannot be emptied first.

3.3.10 SRCANCHORS

Input

- **object identifier**: string.
- **attribute selector**: string array.
- **object query**: string.

Output

- **error**: error.
- **objects**: object array.

Retrieves the source anchors of an object, optionally filtering them through an object query.

3.3.11 DSTANCHORS

Input

- **object identifier**: string.
- **attribute selector**: string array.
- **object query**: string.

Output

- **error**: error.
- **objects**: object array.

Retrieves the destination anchors of an object, optionally filtering them through an object query.

3.3.12 INSERT

This is the generic insert function. It can be used to insert any kind of object into the Hyperwave Information Server. For this reason, it provides a relatively flat interface in that some of the input parameters can be left blank for particular types of objects. For example, inserting a collection certainly does not require a content object to be passed.

Input

- **object**: object.
- **parameters**: object.
- **content**: content.
- **attribute selector**: string array.

The `object` parameter serves as a template for the Hyperwave Information Server object to be inserted. It specifies the type of the object to be inserted, as well as holding additional fields the user wants to pass. The rules of constructing the template are as follows.

The most important characteristic of a Hyperwave Information Server object is its `Type`. Depending on the value of the `Type` field, some more fields may be required, following a more or less hierarchical scheme.

The `parameter` parameter is used to pass information to the Hyperwave Information Server about where the object should go and what its relations to other Hyperwave Information Server objects (existing or non-existent) will be. (For the sake of correctness: although the `parameter` is of type "object" which is defined as a set of attributes which in turn are name-value pairs with (in general) more than one value, a `parameter` is expected to be a set of name-value pairs with one value each.)

The `parameter` scheme is relatively simple:

- An object of type `Type=Document` requires a `Parent` parameter whose value is the unique object identifier (the `Name` or `GOid`) of the collection where it is supposed to go.
- An object of type `Type=Anchor` requires a `Doc` parameter whose value is the unique object identifier (the `Name` or `GOid`) of the document to which it belongs. Additionally, if the type of the anchor (`TAnchor`) is `Src` (a source anchor), the following parameters are accepted (*and mutually exclusive*):
  - `GDest` is the `GOid` (*no* ***`Name`*** *allowed*) of the destination object the anchor is supposed to be connected with.
  - `Hint` is a more sophisticated concept (and reflects one of the major Hyperwave Information Server features). It represents the name of the destination object to which a link should be created automatically. The destination object need not be present at the time the source anchor is inserted (in this case the hint would be more or less equivalent to the `GDest`, except for the difference in the type of the value). Rather, if the destination object is not present at the time of insertion, the hint is recorded inside the Hyperwave Information Server database, waiting to be resolved automatically when an object matching the hint's value comes along.

The `content` parameter must only be given if the template object is of type `Type=Document` and the document type is not `DocumentType=collection` (i.e., a document with a body).

The `attributeSelector` affects the fields of the returned object (as with the other calls that accept an attribute selector as input parameter), not the fields of the template object or the object that is created.

Output

- **error**: error.
- **object**: object.

The `object` output parameter is the representation of the object that has been inserted (if any).

3.3.13 INSERTDOCUMENT

Input

- **object**: object. The object template.
- **parent identifier**: string. The `Parent` parameter of the insert call.
- **content**: content. The document content.
- **attribute selector**: string array.

Output

- **error**: error.
- **object**: object.

This call is a shortcut of the insert call explained above. It is the recommended way of inserting a document.

3.3.14 INSERTCOLLECTION

Input

- **object**: object. The object template.
- **parent identifier**: string. The `Parent` parameter of the insert call.
- **attribute selector**: string array.

Output

- **error**: error.
- **object**: object.

This call is a shortcut of the insert call explained above. It is the recommended way to insert collections.

Note that if you want to insert a different collection type, such as a cluster or sequence, you must put the attribute **CollectionType** in the object template as follows:

Clusters: `CollectionType=Cluster`

MultiClusters: `CollectionType=MultiCluster`

AlternativeClusters: `CollectionType=AlternativeCluster`

Sequences: `CollectionType=Sequence`

3.3.15 INSERTANCHOR

Input

- **object**: object. The object template.
- **document identifier**: string. The `Doc` parameter of the insert call.
- **destination identifier**: string. The `GDest` parameter of the insert call.
- **hint**: string. The `Hint` parameter of the insert call.
- **attribute selector**: string array.

Output

- **error**: error.
- **object**: object.

This call is a shortcut of the insert call explained above. It is the recommended way of inserting an anchor.

3.3.16 REPLACE

The replace function is quite powerful in that it allows the user to change a given object and, if appropriate, its content in one single step.

Input

- **object identifier**: string.
- **object**: object.
- **content**: content.
- **attribute selector**: string array.

As always, the object identifier designates the object to be changed.

If given, the content parameter contains the content of the document that is the replacement of the document associated with the object identifier. If the object identifier refers to a Hyperwave Information Server object which is not able to have a content (for example, a collection), a warning is issued (as always, it is recommended to check for warnings anyhow).

The object parameter requires a bit more explanation. You pass a more or less complete Hyperwave Information Server object. The replace function computes the differences between the passed object and the object referred to by the object identifier. While doing so, it ignores dynamically generated attributes which accidentally slipped into the passed object, as well as some other attributes (for example, the `GOid` attribute is surely not subject to replacement). This enables the user to take the attributes of an existing Hyperwave Information Server object as a template for the object parameter. These differences are then executed by the API in order to make the object look like the template.

For example, if you want to remove the `Name=my/hwis/object` attribute from the following object

```
Author=someone
DocumentType=collection
GOid=0xc0a89919 0x000c83c7
Name=my/hwis/object
Name=some/other/name
Rights=W:u someone
TimeCreated=1998/02/17 14:00:25
Title=en:My Hyperwave Information Server Object
Type=Document
```

take this object with the Name=my/hwis/object attribute removed as the object parameter for the replace call:

```
Author=someone
DocumentType=collection
GOid=0xc0a89919 0x000c83c7
Name=some/other/name
Rights=W:u someone
TimeCreated=1998/02/17 14:00:25
Title=en:My Hyperwave Information Server Object
Type=Document
```

Likewise, if you intend to add an attribute, add it to the original and pass the object as the object parameter.

Output

- **error**: error.
- **object**: object.

If no error occurred, the object output parameter is the Hyperwave Information Server object as it is after the modification.

3.3.17 CONTENT

Input

- **object identifier**: string.

Output

- **error**: error.
- **content**: content.

This call retrieves documents from the Hyperwave Information Server. It returns a content object as described above.

3.3.18 LOCK

Input

- **object identifier**: string.
- **mode**: see below.

Output

- **error**: error.

Locks an object. If the object is a collection, the collection and its descendants are locked recursively if the mode allows it.

Mode is one of:

- `normal`: (the default) locks only the given object, no descendants are considered. If this is not possible, an error is returned.
- `recursive`: if the object in question is a collection, an attempt is made to lock all of its descendants. Where this is not possible, a warning is added to the error return parameter (*Note: not an error*). Be sure to check the error return parameter after a recursive lock call.

3.3.19 UNLOCK

Input

- **object identifier**: string.
- **mode**: see below.

Output

- **error**: error.

Unlocks an object. If the object is a collection, the collection and its descendants are unlocked recursively if the mode allows it.

Mode is one of:

`normal`: (the default) unlocks only the given object, no descendants are considered. If this is not possible, an error is returned.

`recursive`: if the object in question is a collection, an attempt is made to unlock all of its descendants. Where this is not possible, a warning is added to the error return parameter (*Note: not an error*). Be sure to check the error return parameter after a recursive unlock call.

3.3.20 CHECKIN

Input

- **object identifier**: string.
- **version**: string.
- **comment**: string.
- **mode**: see below.
- **object query**: string.

Output

- **error**: error.

Version control. Checks in an object. If the object is a collection, its descendants are checked in recursively, presuming they are documents with a body. This occurs if the mode and the permissions allow it.

`version` is the version identifier that will be given to the document. It must be of the form <integer>.<integer>. These two values are referred to as the major and minor version numbers.

`comment` is a string that describes the changes that were made since the previous version. It can be multiline, but should be restricted to a few lines.

`mode` is a combination of the following values:

- `normal`: (the default) checks in the object in question. If the object is not a document with a body, an error is returned.
- `force version control`: checks in even if the object in question is not yet under version control.
- `recursive`: if the object in question is a collection, an attempt is made to check in all of its descendants. The same procedure is applied to every descendant of the collection, with one exception: no error is returned if the descendant is collection.
- `revert if not changed`: the object is examined on checkin. If changes have been made, the object is checked in as normal. If no changes have been made, the server reverts to the last committed version and the experimental version is deleted.

`object query`: it is only applied if the `mode` is `recursive`. If this is the case, it is only attempted to check out objects that match the object query.

3.3.21 CHECKOUT

Input

- **object identifier**: string.
- **mode**: see below.
- **object query**: string.

Output

- **error**: error.

Version control. Checks out an object. If the object is a collection, its descendants are checked out recursively, presuming they are documents with a body. This occurs if the mode and the permissions allow it.

`mode` is one of the following:

- `normal`: (the default) checks out only the given object, no descendants are considered. If this is not possible, an error is returned.
- `recursive`: if the object in question is a collection, an attempt is made to check out all of its descendants. The same procedure is applied to every descendant of the collection, with one exception: no error is returned if the descendant is collection.

`object query`: it is only applied if the `mode` is `recursive`. If this is the case, it is only attempted to check out objects that match the object query.

3.3.22 REVERT

Input

- **object identifier**: string.
- **version**: string.
- **mode**: see below.
- **object query**: string.

Output

- **error**: error.

Version control. Deletes all versions of an object backwards, up to the given version. If the object is a collection, its descendants are reverted recursively, presuming they are documents with a body. This occurs if the mode and the permissions allow it.

`version`: the version identifier that will be reverted to.

`mode` is one of the following:

- `normal`: (the default) reverts only the given object, no descendants are considered. If this is not possible, an error is returned.
- `recursive`: if the object in question is a collection, an attempt is made to revert all of its descendants. The same procedure is applied to every descendant of the collection, with one exception: no error is returned if the descendant is collection.

Caution: the recursive mode is somewhat dangerous in that it is the user's responsibility to make sure the version identifiers in the tree are assigned consistently. Therefore it is recommended not to intermix recursive and non-recursive modes of the version control calls on the same tree.

3.3.23 HISTORY

Input

- **object identifier**: string.
- **attribute selector**: string array.
- **object query**: string.

Output

- **error**: error.
- **objects**: object array.

Version control. Retrieves all versions of the object in question from the Hyperwave Information Server.

3.3.24 FIND

Input

- **keyquery**: string.
- **fulltextquery**: string.
- **object query**: string.
- **scope**: string array.
- **languages**: string array.

Output

- **objects**: object array.
- **error**: error.

Performs a search on the Hyperwave Information Server. Either a keyquery or a fulltextquery must be present. If both are present, then they will be executed in sequence. The found objects will be then filtered by the objectquery. If no scope is given the entire server is searched. The languages in which the search will be carried out must be given in any case.

3.3.25 HWSTAT

Input

- **attribute selector**: string array.

Output

- **error**: error.
- **object**: object.

Retrieves statistics of the "hgserver" Hyperwave Information Server component. Currently, this is an object containing only one attribute, namely `Version=<String>`.

3.3.26 DCSTAT

Input

- **attribute selector**: string array.

Output

- **error**: error.
- **object**: object.

Retrieves statistics of the "dcserver" Hyperwave Information Server component. Currently, this is an object containing the following attributes.

- `Version=<a string>`
- `UpSince=<Hyperwave Information Server time string>` a string in Hyperwave Information Server time format representing the time dcserver was last started.
- `MegsRetrieved=<number>` where number is a floating point number in decimal notation representing the number of megabytes retrieved since startup.
- `NumDocuments=0xdeadbeef` the number of documents stored in hexadecimal.
- `MaxDocuments=0xdeadbeef` the maximum number of documents allowed.

3.3.27 DBSTAT

Input

- **attribute selector**: string array.

Output

- **error**: error.
- **object**: object.

Retrieves statistics of the “dbserver” Hyperwave Information Server component. Currently, this is an object containing the following attributes.

- `UpSince=<Hyperwave Information Server time string>` a string in Hyperwave Information Server time format representing the time dcserver was last started.
- `LocalTime=<Hyperwave Information Server time string>` the server’s local time.
- `UserSess=0xdeadbeef` the current number of dbserver sessions.
- `CustomIndex=<field name>:<type>` for every custom index that is configured, there is one such entry. <field name> is the field in the Hyperwave Information Server object which is configured to be indexed, <type> is the type of the field’s value.

3.3.28 FTSTAT

Input

- **attribute selector**: string array.

Output

- **error**: error.
- **object**: object.

Retrieves statistics of the "dbserver" Hyperwave Information Server component. Currently this is an object containing the following attributes.

- `Version=<a string>`
- `UpSince=<Hyperwave Information Server time string>` a string in Hyperwave Information Server time format representing the time dcserver was last started.
- `SearchEngine=<string>` the search engine currently used; one of "native" and "verity".
- `SearchEngineVersion=<a string>` the version of the search engine currently used.

3.3.29 USERLIST

Input

- **attribute selector**: string array.

Output

- **error**: error.
- **objects**: object array.

Retrieves a list of users that are currently logged in to the Hyperwave Information Server. Every object in the list consists of the following attributes:

- `Id=0xdeadbeef` the id of the user's session.
- `Username=<string>` the login name of the user.
- `Hostname=<string>` the host the user logged in from.
- `Logintime=<string>` the absolute time the user was logged in in a string in Hyperwave Information Server time format
- `Idletime=` the absolute time the user was idle (HH:MM:SS)
- `Client=<string>` the user's client program.
- `ClientVersion=<string>` the client program's version.

3.3.30 DSTOFSRC

Input

- **object identifier**: string.

- **attribute selector**: string array.

Output

- **error**: error.
- **object**: object.

Retrieves the destination object of a source anchor.

3.3.31 SRCSOFDST

Input

- **object identifier**: string.
- **attribute selector**: string array.
- **object query**: string.

Output

- **error**: error.
- **objects**: object array.

Retrieves the source anchor that points to a given destination (anchor), optionally applying an object query.

3.3.32 OBJECTBYANCHOR

Input

- **object identifier**: string.
- **attribute selector**: string array.

Output

- **error**: error.
- **object**: object.

Retrieves the object an anchor belongs to.

4 APPENDIX A

4.1 NETSCAPE COPYRIGHT STATEMENT