NORDUGRID-TECH-17
30/4/2009

Chelonia - Self-healing distributed storage system

Documentation and developer's guide

Zsombor Nagy*
Jon Nilsen†
Salman Zubair Toor ‡

*zsombor@niif.hu
†j.k.nilsen@usit.uio.no
‡salman.toor@it.uu.se

Contents

Chapter 1

Design Overview

Chelonia is a distributed system for storing replicated *file*s on several file storage nodes and managing them in a global namespace. The files can be grouped into *collection*s (a concept very similar to directories in the common file systems). A collection can contain sub-collections and sub-sub-collections in any depth. There is a dedicated *root collection* to gather all collections to the global namespace. This hierarchy of collections and files can be referenced using *Logical Name*s (*LN*'s). The users can use this global namespace as if they were using a local filesystem. Files can be transfered by several different transfer protocols. The client side tools hide this from the user. The replicas of the files are stored on different storage nodes. A storage node here is a network-accessible computer having storage space to share, and a running storage element service (e.g. HTTP(S), FTP(S), GridFTP, ByteIO[1], etc.). On each storage node a Chelonia service is needed to be installed to manage and integrate it into the system. The client side provides access to third-party storage solutions through the namespace of Chelonia. The main services of the storage system are the following (see Figure 1.1):

- the **A-Hash** service, which is a replicated database which is used by the Librarian to store metadata;
- the **Librarian** service, which handles the metadata and hierarchy of collections and files, the location of replicas, and health data of the Shepherd services, using the A-Hash as database;
- the **Bartender** service, which provides a high-level interface for the users and for other services;
- the **Shepherd** service, which manages storage element services, and provides a simple interface for storing files on storage nodes.

1.1 Files and collections

The storage system is capable of storing files which can be grouped in collections and sub-collections, etc. Every file and collection has a unique ID in the system called the *GUID*. Compared to the well-known structure of local file systems, these GUID's are very similar to the concept of *inode*s. And as a directory on a local filesystem is basically just a list of name and inode pairs, a collection in Chelonia is just a list of name and GUID pairs. There is a dedicated collection which is the *root collection*. This makes the namespace of Chelonia a hierarchical namespace where you can start at the root collection, and go to sub-collections and sub-sub-collections to get to a file. This path is called the *Logical Name* (LN). For example if there is a sub-collection called `saturn` in the root collection, and there is a file called `rings` in this sub-collection, then the LN of this file is `/saturn/rings`.

In addition to the Logical Names a file or collection can be refered to simply by its GUID, or you can use GUID's and Logical Names together, as seen in Figure 1.2.

The full syntax of Logical Names is `/[path]` or `<GUID>[/<path>]` where [...] indicates optional parts.

An example of the hierarchy of the global namespace is shown in Figure 1.2: if you have a collection with GUID `1234`, and there is a collection called `green` in it, and in `green` there is another collection called

[1] OGSA ByteIO Working Group (BYTEIO-WG), `https://forge.gridforum.org/projects/byteio-wg/`

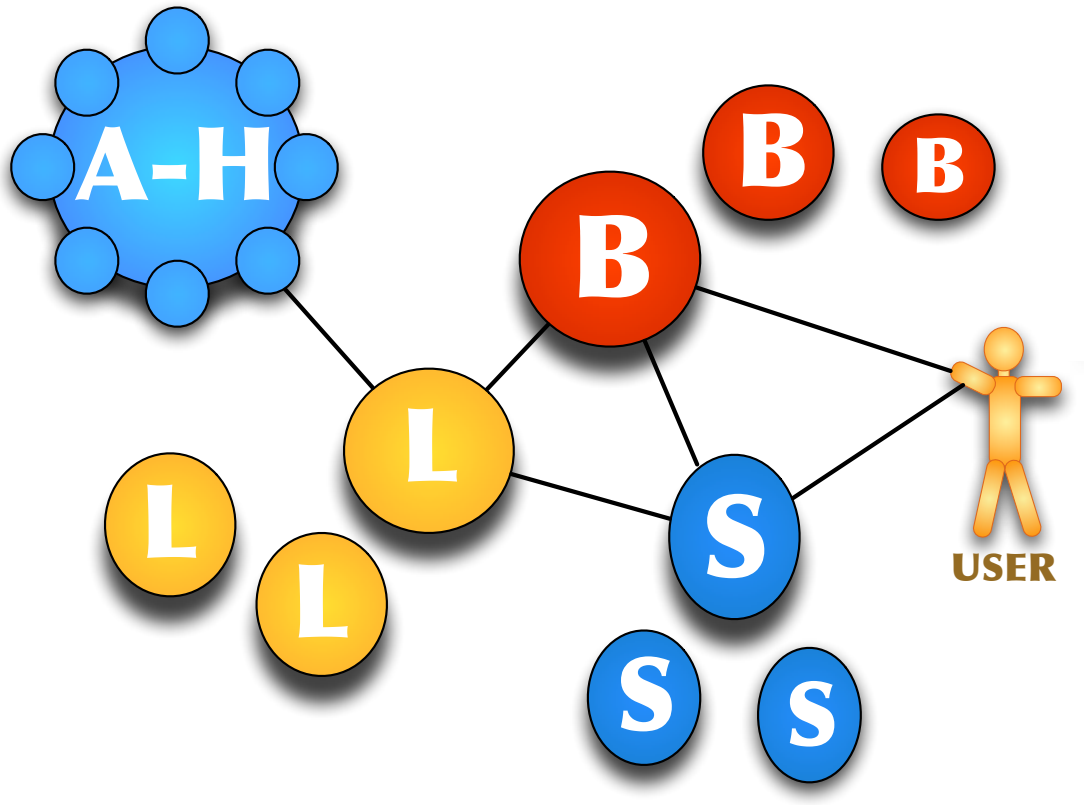

Figure 1.1: The components of Chelonia: the **A-H**ash service, the **L**ibrarian service, the **B**artender service and the **S**hepherd service.

`orange`, and in `orange` there is a file called `huge`, then you can refer to this file with the Logical Name `1234/green/orange/huge`, which means that from the collection called `1234` you have to follow along the path: `green`, `orange`, `huge`.

There is a dedicated root collection (which has the GUID `0`), and if a LN starts without a GUID prefix, it is implicitly prefixed with the GUID of the root collection, e.g. `/why/blue` means `0/why/blue`. If a user wants to find the file called `/why/blue`, the system knows where to start the search: the GUID of the root collection. The root collection knows the GUID of `why`, and the (sub-)collection `why` knows the GUID of `blue`. If the GUID of this file is `5678`, and somebody makes another entry in collection `/why` (= `0/why`) with name `red` and GUID `5678`, then the `/why/red` LN points to the same file as `/why/blue`, a concept very similar to a hardlink in a regular local (POSIX-like) file system.

1.2 Storage nodes and replicas

The collections in Chelonia are logical entities. The content of a collection is stored as metadata of the collection, which means that a collection actually has no physical data. A file, however, has both metadata and real physical data (the actual content of the file). The metadata of a file is stored in the same database where the collections are stored, but the physical data of a file is stored on storage nodes as multiple replicated copies.

A storage node consists of two things: a storage element service which is capable of storing and serving files through a specific protocol (e.g. a web server, an FTP server, a GridFTP server, etc.) and a Shepherd service which provides a simple interface to access the storage node, and which can initiate and manage file transfers through the storage element service. The Shepherd has different backends for the supported storage element services which made it possible to communicate with them.

A logical file, which are part of the hierarchical namespace, has a GUID and other metadata, and one or

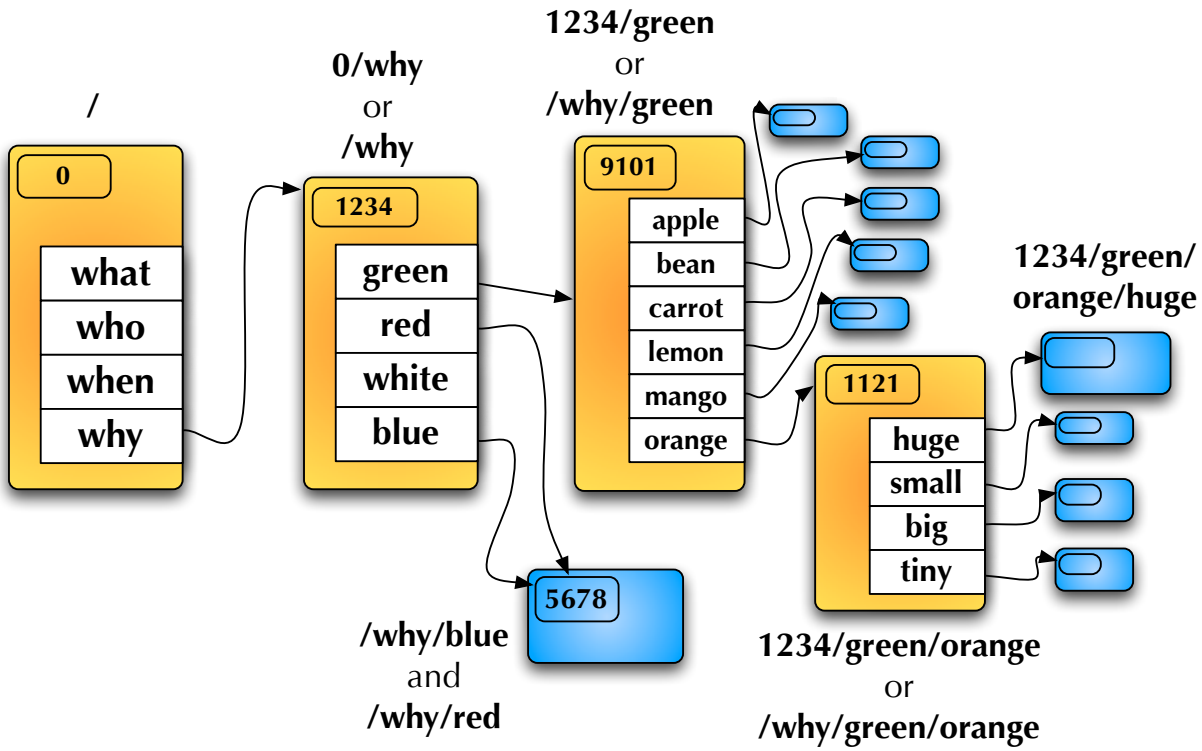

Figure 1.2: Example of the hierarchy of the global namespace

more physical replicas. The physical replicas are stored on separate storage nodes. In order to connect the logical file to its replicas, the system needs to have global pointers. Each storage node has a URL and each replica has a unique ID within the storage node called *referenceID*, the URL and the referenceID together is called a *Location*, a Location unambiguously points to one specific replica. To connect the logical files to the physical ones, each logical file has a list of Locations.

The user can specify how many replicas are required for each file. The storage nodes periodically check the number of replicas for each file they store, and automatically creates new replicas if there are fewer than required, and removes replicas if there are more. (Currently if a storage node stores a great amount of files, then the checking period could be longer.)

The file replicas can be in the following states: the replica can be valid and alive or just in the process of creation or it may be corrupt or a whole storage node may be offline. This state is always stored as metadata of the given replica. For each file there is a checksum calculated, and this checksum is used to detect if a replica gets corrupted. If a storage node (more precisely: the Shepherd service on the storage node) detects that a file is invalid, it reports this so the metadata will be in sync with the real state. The storage nodes sends heartbeat messages periodically, and if a storage node goes offline, the missing heartbeat also triggers the modification of metadata.

1.3 The A-Hash

The A-Hash is the metadata store of the storage system. The A-Hash may be deployed as either a centralized or a replicated service. It consistently stores 'objects' which contain property-value pairs organized in sections. All metadata about files and collections in the storage system are stored in the A-Hash. Additional system configutation information (e.g. about A-Hash replication, about Shepherd services, etc.) is stored in it as well. The A-Hash itself does not interpret any data.

1.4 The Librarians

The Librarian manages the storage system hierarchy of files and collections including their metadata. It can traverse Logical Names and return the corresponding metadata. The Shepherds (see Sect. 1.6) send their heartbeat messages to the Librarians and, if needed, the Librarian responds by modifying the states of files stored on a certain storage node. The Librarian itself is a stateless service, it uses the A-Hash to store and retrieve the metadata and thus there can be any number of independent Librarian services (all using the same A-Hashes). This ensures high-availability and load-balancing of the storage system.

1.5 The Bartenders

The Bartender service provides the high-level interface of the storage system to clients. Every interaction between a client and the storage system starts by sending a request to a Bartender. Using the Bartender it is possible to create and remove collections, create, get and remove files, move files and collections within the namespace using Logical Names, etc. The Bartender authorizes users and applies access policies of files and collections. It communicates with the Librarian and Shepherd services to execute the requests of the clients. The file data itself does not go through the Bartender; file transfers are performed directly between the storage nodes and the clients. Any number of independent Bartender services may be deployed in the system to ensure high-availability and load-balancing. The Bartender also provides access to files in third-party storage solutions through its interface by mounting the namespace of the third-party storage into the namespace of Chelonia (this is accomplished by so-called 'gateway' modules).

1.6 The Shepherds

The Shepherd service manages the storage element service on the storage node. It reports its health state to a Librarian and provides the interface for initiating file transfers. For each kind of storage element service (e.g. an HTTP server, an FTP server, a storage solution with a GridFTP interface, etc.) a Shepherd backend is needed which is capable of managing the given storage element service. The Shepherd service periodically checks the health of the replicas using their checksums, and if a replica is deleted or corrupted, the Shepherd tries to recover it by downloading a valid copy from another storage node. The Shepherd also periodically checks if a file has the correct number of replicas in the system. If too few or too many replicas are found, new replicas will be created or redundant replicas removed.

1.7 Security

Chelonia consists of several services (both different kinds of services and multiple instances of the same kind). The A-Hash, Librarian and Shepherd services are 'internal' services in the sense that the end-user of the storage system never communicates with them directly. But these internal services are communicating with each other, thus they have to know which other services to trust. This aspect of the security architecture of Chelonia is called 'inter-service authorization'.

The end users always connect to one of the Bartender services, which will decide which permissions the user has. This is the 'high-level authorization' part of the security architecture.

For transferring the actual file data, the users have to connect to storage element services running on storage nodes. These services have their own authentication and authorization methods. Managing these aspects is the 'transfer-level authorization' part of the security architecture of Chelonia.

1.7.1 Inter-service authorization

In a deployment of Chelonia, it is possible to have several A-Hash, Librarian, Shepherd and Bartender services. The Bartenders send requests to the Librarians and the Shepherds, the Shepherds communicate with the Librarians and the Librarians with the A-Hashes. If any of these services gets compromised or a new rogue service is inserted in the system, security is lost. That's why it is vital for each service to

authorize other services before sending or accepting requests. The services communicate via the HTTPS protocol, which means that they should provide an X.509 certificate for each connection, and that they can examine each other's certificates. Each service is required to have an X.509 certificate and is thus identified by its Distinguish Name (DN). A list of trusted DN's can be configured into each service, or a list of trusted services can be stored on a remote location. The services will only accept connections if the DN of the other end is listed in this list of trusted DN's. However, the Bartender services will accept any incoming connection, which are from the users, because the users are authenticated differently.

1.7.2 High-level authorization

The Librarian component of Chelonia manages all the metadata about files and collections. For each file and collection there are access policies in the form of access control rules, and these are stored among the metadata. The users are identified by their DN's, and an access control rule specifies the rights of the given user. A rule can be represented like this:

```
DN +action +action -action
```

This contains a list of actions, each prefixed with a `+` or `-` character which indicates that the given action is allowed or not allowed for the given DN.

Besides specifying only one user with a DN, there are other types of access control rules. For example, you can have a rule for a whole VO (Virtual Organization) or for all users, like this:

```
VOMS:knowarc.eu +action -action
ALL +action
```

These are the actions which can be used for access control:

- *read*: user can get the list of entries in the collection; user can download the file
- *addEntry*: user can add a new entry to the collection;
- *removeEntry*: user can remove any entry from the collection
- *delete*: user can delete the collection if it is empty; user can delete a file
- *modifyPolicy*: user can modify the policy of the file/collection
- *modifyStates*: user can modify some special metadata of the file/collection (close the collection, change the number of needed replica of the file)
- *modifyMetadata*: user can modify the arbitrary metadata section of the file/collection (these are property-value pairs)

Additionally, each file and collection has an 'owner' which is a user who always can modify the access control rules.

1.7.3 Transfer-level authorization

Currently the transfer-level authorization is very simple. When the Bartender decides that a user has permission to download a file, then the Bartender chooses a replica, and initiates the transfer. The result of this initiation is the transfer URL (TURL). This TURL is unique for each request, even for requests for the same replica, and this TURL is only valid for one download. Currently the storage element services are configured to not do any authorization, and these one-time TURL's are used to ensure that only authorized users can access the contents of the storage elements.

Chapter 2

Implementation of use cases

This chapter contains some examples to demonstrate the internal mechanism of Chelonia.

2.1 Listing the contents of a collection

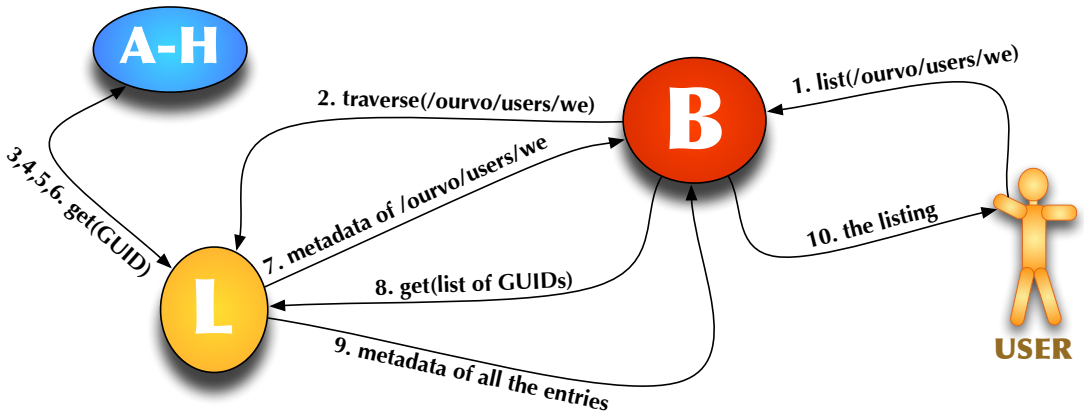

Figure 2.1: Listing the contents of a collection

A user wants to list the contents of a collection, which has a Logical Name of `/ourvo/users/user`. In order to do this, the client tool has to contact a Bartender, and send a request to it containing the Logical Name the user wants to list, and the response from the Bartender will contain the list of entries. The steps are represented in Figure 2.1.

1. The client tool needs to know the URL of a Bartender. This can be preconfigured on the client side or, in future releases it may be acquired from an information system. When the client tool has the URL, it sends a 'list' request which contains the Logical Name `/ourvo/users/user`.

2. The Bartender tries to get the metadata of the given LN by sending a 'traverseLN' request to a Librarian.

3. The Librarian service starts the traversing by asking an A-Hash service about the first part of the LN, which is the `/` root collection. The A-Hash service only knows about GUID's and not about LN's, but the GUID of the root collection is well-known, so the A-Hash can return the metadata of the list of files and sub-collections in the root collection.

4. Hopefully, the `ourvo` collection can be found in the root collection, which means that the Librarian knows its GUID, and can ask for its metadata from the A-Hash.

5. After the A-Hash returns the metadata of the `/ourvo` collection, the Librarian finds the GUID of `users` in it, then gets its metadata.

6. The A-Hash returns the metadata of `/ourvo/users` which contains the GUID of `user`, so the Librarian can ask for its metadata.

7. At last the A-Hash returns the metadata of `/ourvo/users/user` to the Librarian, and the Librarian returns it to the Bartender. This metadata contains the list of entries within this collection, and it also contains the access policies for this collection.

8. The Bartender first checks, based on the user's DN (or his VO membership) and the access policies of this collection, if the user has permission to get the contents of this collection or not. If the user is approved, then because the 'list' request should return additional metadata about each entry in the collection, the Bartender sends a 'get' message to a Librarian which requests the metadata of all the entries in this collection.

9. The Librarian gets the metadata from the A-Hash and returns it to the Bartender.

10. Now the Bartender has all the needed information and returns the proper response to the client tool which formats and prints the results nicely for the user.

2.2 Downloading a file

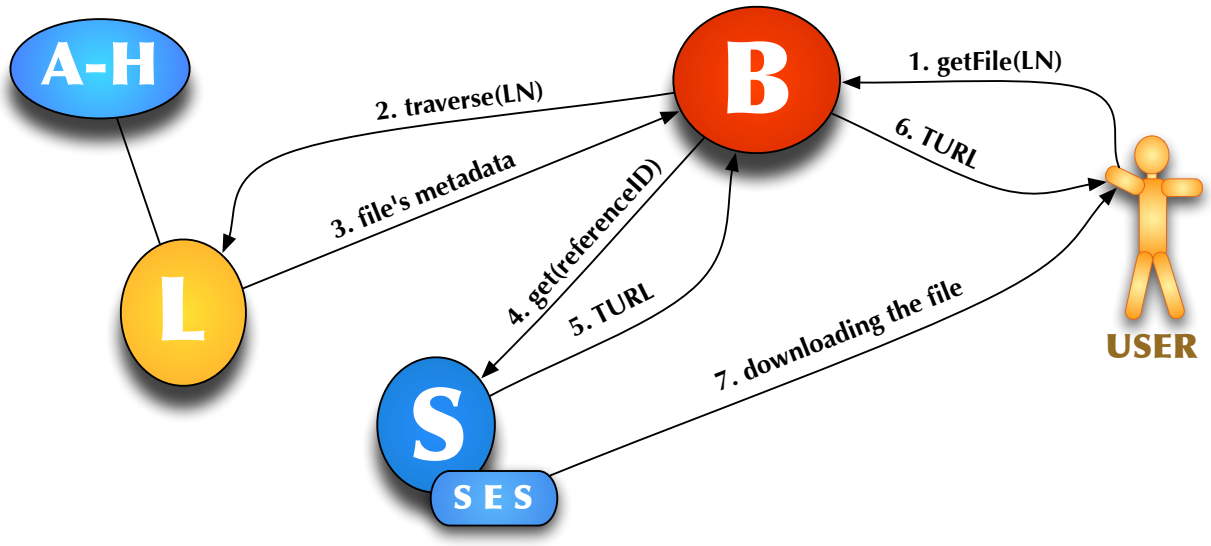

Figure 2.2: Downloading a file

In this use case a user wants to download a file which has a Logical Name of `/ourvo/users/user/thefilehewants` (see Figure 2.2).

1. The client tool connects to a Bartender and sends a 'getFile' request with the LN of the file.

2. The Bartender contacts a Librarian to traverse the Logical Name and to get the metadata of our file.

3. The Librarian does the step-by-step traversing of the name space and gets all the metadata from the A-Hash and returns the metadata of our file to the Bartender. This metadata contains the location of the file's replicas and the access policies of this file.

4. The Bartender checks, based on the access policies and the user's identity, if he is allowed to get the file, and if allowed, it chooses a replica location (currently randomly). A location consists of the URL of a Shepherd service, and the ID of the replica within that Shepherd (which is called a 'referenceID'). The Bartender sends a 'get' request to the chosen Shepherd.

5. The Shepherd prepares the file transfer by asking the storage element service to create a new one-time TURL for this replica. The Shepherd returns the TURL to the Bartender.

6. The Bartender returns the TURL to the client tool.

7. The client tool use this TURL to get the file directly from the storage element service (SES) on the storage node.

2.3 Creating a collection

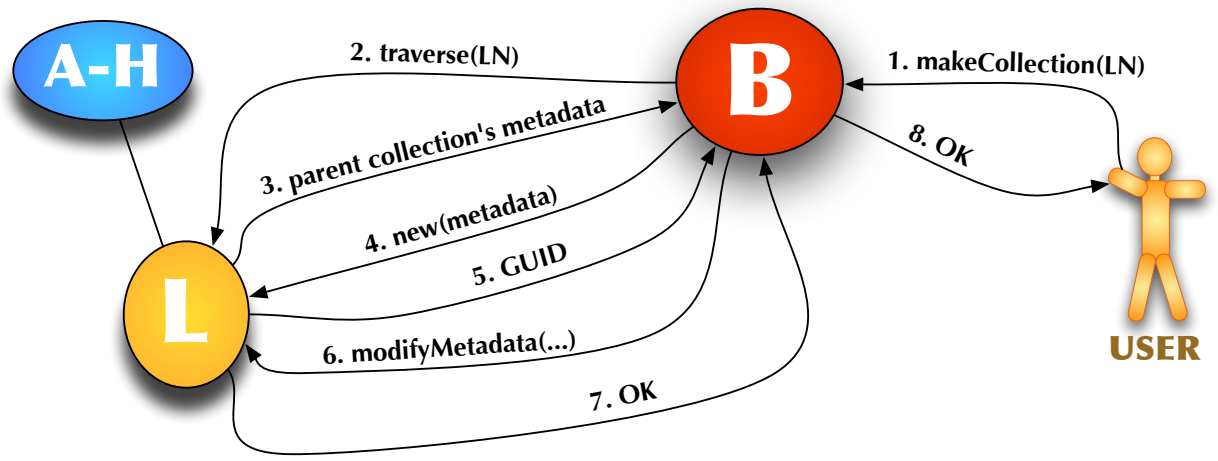

Figure 2.3: Creating a collection

Here, the user wants to create a new (empty) collection as sub-collection of `/ourvo/common`, and he wants to call it `docs` (see Figure 2.3)

1. The client tool contacts a Bartender and send a 'makeCollection' request with the LN `/ourvo/common/docs`.

2. The Bartender asks a Librarian to traverse this LN.

3. The Librarian tries to traverse the Logical Name and it stops at the last possible point and returns the metadata of the last element. Because the user wants to put his collection in a new path but into an existing collection, only the `/ourvo/common` part of the LN is expected to be traversed. If the Librarian can traverse the whole LN it means that there is already an existing file or collection by that name. If LN does not exist but the the parent collection does, then the parent collection's metadata is returned to the Bartender.

4. The Bartender checks the access policies to decide if the user has permissions to put something into this collection. Then it asks the Librarian to create a new collection.

5. The Librarian creates the collection, and returns its GUID. At this point this new collection has no real Logical Name, it only has a GUID, but it is not yet put into its parent collection.

6. The Bartender now asks the Librarian to add this new entry into the parent collection, which means that the new GUID and the name `docs` are added as a pair.

7. The Librarian returns with a status message.

8. Finally the Bartender tells the client tool if everything went OK or not.

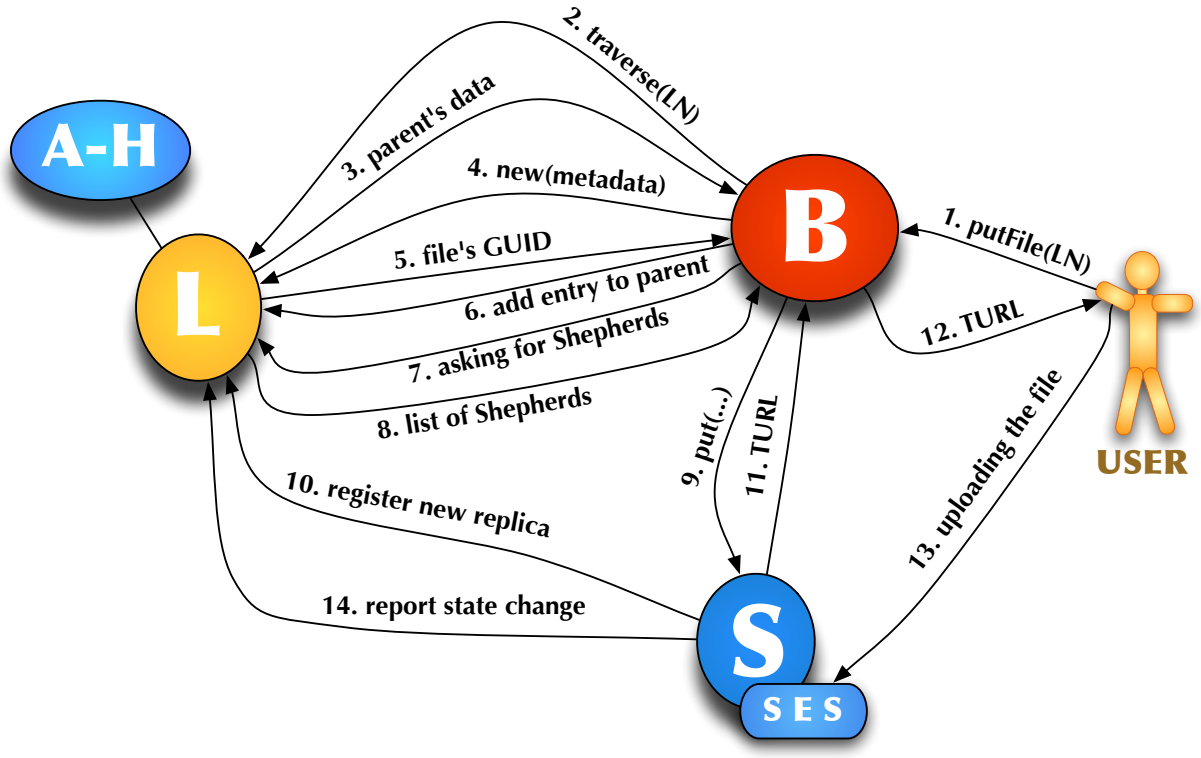

Figure 2.4: Uploading a file

2.4 Uploading a file

The user has a file on his local disk and wants to upload the file to a collection called `/ourvo/common/docs`. (See Figure 2.4.)

1. The client tool contact a Bartender to upload the file, providing the file's metadata, including the the Logical Name (LN), which in this case will be `/ourvo/common/docs/proposal.pdf`
2. The Bartender asks a Librarian to traverse this LN.
3. The Librarian traverses the Logical Name, and stopping at the `/ourvo/common/docs` part of the LN, which means that the name is available and the parent collection exists. If everything is OK, the metadata of the parent collection is returned to the Bartender.
4. The Bartender checks the access policies to decide if the user has permissions to put anything into this collection. Then it asks the Librarian to create a new file entry.
5. The Librarian creates the entry and returns its GUID.
6. Next, the Bartender adds the name `proposal.pdf` and the new GUID to the collection `/ourvo/common/docs` and from now on there will be a valid LN `/ourvo/common/docs/proposal.pdf`. However this LN points to a file which has currently no replica at all. If someone tried to download the file called `/ourvo/common/docs/proposal.pdf` now, they would get an error message.
7. The Bartender asks the Librarian about Shepherd services (which are sitting on storage nodes).
8. The Librarian returns a list of Shepherd services.
9. The Bartender randomly chooses a Shepherd service and sends it a 'put' request to initiate the file upload.
10. The Shepherd communicates with the storage element service (SES) on the same node to create a new transfer URL (TURL). Then it creates a 'referenceID' for this file and reports to the Librarian that

there is a new replica in the `creating` state. The Librarian gets the message from the Shepherd and adds the new replica to the new file. Now the file has one replica, which is not uploaded yet into the system. If someone tries to download this file now, they still get an error message.

11. The Shepherd returns the the TURL to the Bartender.
12. The Bartender returns the TURL to the client tool.
13. Then client tool can upload the file to this TURL (to the storage element service, SES)
14. The Shepherd detects that the file has been uploaded. In order to do this the storage element service does post processing (e.g. remove a hardlink, do a SOAP call, etc.) after it got the file. The Shepherd checks the checksum of the file, and if it is OK, then it reports to the Librarian, that this replica is now `alive`. The Librarian alters the state of this location, and now the file has one valid replica.

2.5 Removing a file

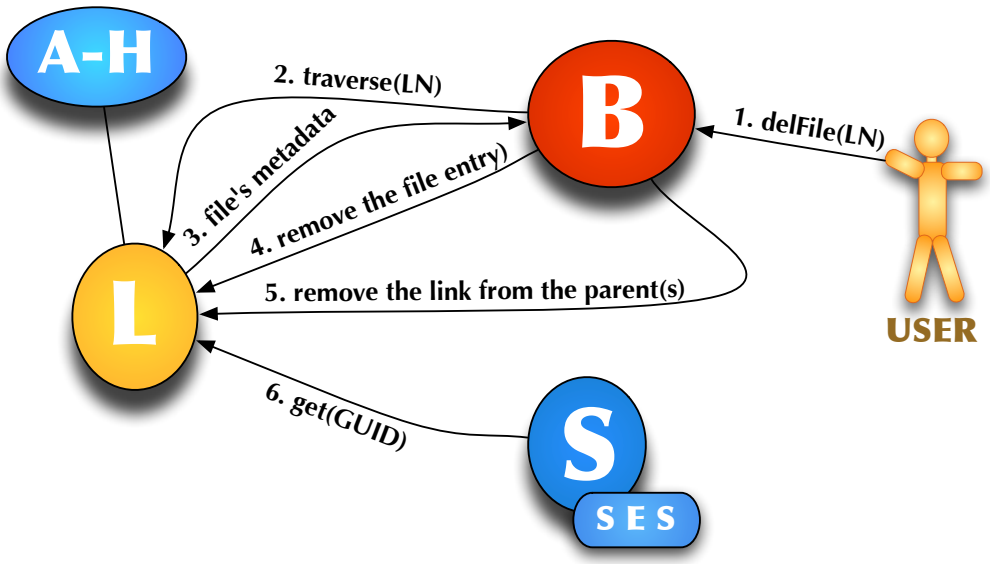

Figure 2.5: Removing a file

In Chelonia, files are deleted through so called *lazy deletion*, meaning that at first only the Logcial Name (LN) is deleted, while the physical replicas are removed at a later stage.

1. If the user wants to remove a file, his client tool should connect to a Bartender with the LN of the file he wants to remove.
2. The Bartender asks the Librarian to traverse the LN.
3. The Librarian returns the metadata of the file. The metadata contains information about all the hardlinks which points to this file if there are more than one.
4. Now the Bartender asks the Librarian to remove the file.
5. After that, the Bartender asks the Librarian to remove the links to this file from all the parent collections.
6. The next time a Shepherd which has a replica of this file does its periodic check, it asks the Librarian about the file, and notices that the file does not exist anymore, so it removes the replica itself from the storage node.

Chapter 3

Technical description

3.1 Framework and language

The Chelonia services are written in Python and execute in the HED[1] hosting environment. The HED itself is written in C++, but there are language bindings which allow services to be written in other languages, e.g. in Python or Java. The source code of the storage services is in the NorduGrid Subversion repository[2].

Because the next-generation information system of ARC is currently under development, it cannot be used to discover services. Hence, currently the URL's of almost all the services are hard-coded in the configuration files of the service. There is one exception: the Shepherd services are reporting their URL's to the Librarians, so a Bartender can always get the most current list of accessible Shepherd services.

The HED has a security framework which the Bartenders use to make access policy decisions. The details of this is described in Section 3.3.

3.2 Data model

The storage system stores different kinds of metadata about files, collections, mount points, Shepherd services, etc. Each of these has a unique ID; a GUID.

The A-Hash services provide a functionality to store 'objects', where each object has a unique ID, and contains property-value pairs organized in sections. The properties, the values and the section names are simple character strings. The A-Hash services provide a simple interface to interact with these objects, e.g. to do conditional and atomic changes.

The Librarian services use the A-Hash services to store the metadata, using the GUID's as unique ID's. For this, the metadata have to be represented as property-value pairs organized in section. The files, the collections and the mount points have some common attributes: the GUID, the type and the owner which are in the *entry* section; the creation and modification timestamps in the *timestamps* section, a list of access control rules in the *policy* section; the list of parent collections where this entry has a 'hardlink' in the *parents* section; and optional, arbitrary property-value pairs in the *metadata* section.

entry section

- *type*: the type of the entry: 'collection', 'file' or 'mountpoint'
- *GUID*: the unique ID of the entry
- *owner*: the DN of the user who owns this entry

timestamps section

- *created*: timestamp of creation

[1] The ARC container - `https://www.knowarc.eu/documents/Knowarc_D1.2-2_07.pdf`
[2] `http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/services/storage`

- *modified*: timestamp of last modification (the support for this is not implemented yet)

policy section

- *(identity, action list) pairs*: a list of allowed and not allowed actions for different identities (users, VO's, etc., see Section 1.7.2 for details).

parents section

- list of GUID's of collection where this entry is located, and the name of this entry within the given collections

metadata section

- optional arbitrary property-value pairs

3.2.1 Files

A file in Chelonia has a logical entry in the namespace, and one or more physical replicas on the storage nodes. The locations of the replicas have to be stored in the *locations* section. A location contains the URL of the Shepherd service, the local ID of the file within the storage node and the state of the replica. This state is '**alive**' if the replica passed the checksum test, and the Shepherd reports that the storage node is healthy, '**invalid**' if the replica has wrong checksum, or the Shepherd claims it has no such file, '**offline**' if the Shepherd is not reachable, but may have a valid replica, '**creating**' if the replica is in the state of uploading, '**thirdwheel**' if the replica is marked for deletion because of too many replicas, or '**stalled**' if the the replica was in the state of uploading for a long time but hasn't arrived. The *states* section contains the size of the file, the number of required replicas, and a checksum of the file, which is created by some kind of checksumming algorithm (currently only md5 is supported).

locations section

- *(location, state) pairs*: where a location is a (*URL*, *referenceID*) pair serialized as a string, where *URL* is the address of the Shepherd service storing this replica, *referenceID* is the ID of the file within that Shepherd service.

states section

- *size*: the file size in bytes
- *checksum*: checksum of the file
- *checksumType*: the name of the checksum method
- *neededReplicas*: how many valid replicas should this file have

3.2.2 Collections

A *collection* is a list of files and other collections, which are in parent-children relationships forming a tree-like hierarchy (with possible interconnection due to hardlinks). Each entry has a unique name within a collection and a GUID which points to a corresponding file or collection, so a collection is basically a list of name-GUID pairs, which list is stored in the *entries* section (see below). The collections can be in a closed state, which means that its contents should not be changed. If you close a collection then it cannot be opened again. However it cannot be guaranteed that the contents of a closed collection remains the same (e.g. if a user include in his collection a file which he does not own, then the owner of the file can still remove it), but if a closed collection is changed, its state will be broken, and this state can never be changed again, which means you will always know if a closed collection is not intact anymore. This state is stored in the *states* section, and its possible values are '**no**' if the collection is not closed, '**yes**' if the collection is closed and '**broken**' if the collection was closed, but something happened, and its contents have been changed.

entries section

- *(name, GUID) pairs*: a collection is basically a list of name-GUID pairs.

states section

- *closed*: this indicates if the collection is closed or broken.

3.2.3 Mount Points

The mount point entry is a reference to an external third-party storage. Its metadata is basically just the URL of the external storage end point.

mountpoint section

- *externalURL*: the URL of the external storage end point.

3.2.4 Shepherds

The Librarian stores information about the registered Shepherd services. Each Shepherd reports its URL and the list of its files to a Librarian, and for each Shepherd a GUID is created. There is a special entry (with GUID '1' by default) which stores a list of the registered Shepherd services, with their GUID and the timestamp of the last heartbeat message from them.

nextHeartBeat section

- *(URL, timestamp) pairs*: contains when was the last heartbeat of this service

serviceGUID section

- *(URL, GUID) pairs*: connects the URL of the Shepherd to the GUID where the information is stored about the Shepherd's stored files

And for each Shepherd there is a separate entry with the list of all files stored on the given storage nodes:

entry section

- *type*: 'shepherd'

files section

- *(referenceID, GUID) pairs*: for each replica stored on the Shepherd

3.3 Security implementation

The ARC HED hosting environment has a security framework, and it has its own policy language for describing access policies and requests. The storage system currently use a different internal format to store the access policies, but when the time comes to make decisions, it converts the stored policies and the incoming requests to the ARC native policy format, then asks the security framework to decide.

The internal representation of the access policies is based on access rules. A rule contains an identity and a list of actions. Currently the identity can be one of the following:

- a Distinguish Name (DN) of a user, e.g. '`/DC=eu/DC=KnowARC/O=NIIFI/CN=james`'
- a name of a Virtual Organization (VO) prefixed with '`VOMS:`', e.g. '`VOMS:knowarc.eu`'
- '`ALL`' to specify that this rule will be applied to everyone
- '`ANONYMOUS`' for a rule which will be applied to unauthenticated users (e.g. when non-secure HTTP is used and there is no DN)

Then for each rule there is list of actions, each action is either allowed or denied. In the internal representation of an access rule the list of actions is one single string, where the name of actions are separated with a space, and each name is prefixed with a '`+`' or '`-`' sign indicating if the action is allowed or denied for the given identity. The list of actions can be found in section 1.7.2.

When there is an incoming connection to a Bartender, the security framework of the HED extracts the identity information of the connecting client (DN of client, DN of the issuer CA, extended attributes, etc.),

and all this information is accessible by the Bartender. When the given method of the Bartender gets to the point where the access rules are present, and it is clear that what kind of action the user wants to do, then both the request and the policy are converted to the native ARC policy language, and then the policy evaluator of the security framework is called to make the decision. The native ARC policy language defines attribute types for attributes within requests and policies. The XML representation of an ARC policy or request contains several 'Attribute' elements, each have a type and a value. The storage actions are also put into Attribute elements with the type '`http://www.nordugrid.org/schemas/policy-arc/types/storage/action`'.

For inter-service authorization the services can be configured to only accept connections from trusted services. A service is trusted if its DN is known. For each service the trusted DN's can be inserted to the configuration, or the list can periodically be retrieved from an A-Hash. This can be configured with a '*TrustManager*' section in the service configuration, which contains these entries:

DN is a trusted DN

CA is a DN of a trusted CA: all certificates issued by this CA are trusted

DNsFromAHash is a list of A-Hash URL's (in **AHashURL**) from where a list of trusted DN's are periodically obtained (there can be multiple A-Hash URL's listed here in case of one is offline). The following attributes are obtained: **CheckingInterval** specifies how frequently the list from the A-Hash should be updated (default: 600 seconds); **ID** specifies the GUID where the list is stored (default: '3').

The TrustManager section itself have the following optional attributes: **FromFile** can specify a filename if the whole TrustManager configuration should be taken from a separate file; and **Force** indicates (with 'yes' or 'no', default: 'yes') that you want to ensure that your service does not even get the request if the client of the incoming connection is not trusted.

Example TrustManager configuration:

```
<TrustManager Force="yes">
    <DN>/DC=eu/DC=KnowARC/O=NIIFI/CN=host/epsilon</DN>
    <DN>/DC=eu/DC=KnowARC/O=NIIFI/CN=host/phi</DN>
    <CA>/DC=eu/DC=KnowARC/CN=storage-1233659377.11</CA>
    <DNsFromAHash CheckingInterval="10">
        <AHashURL>https://localhost:60000/AHash</AHashURL>
    </DNsFromAHash>
</TrustManager>
```

All the services must have their own X.509 certificates, which they use when connecting to other services. Currently, it is needed to set the paths of the certificate and private key files (and the trusted CAs as well) in the configuration of each service, like this:

```
<ClientSSLConfig>
    <KeyPath>certs/hostkey-epsilon.pem</KeyPath>
    <CertificatePath>certs/hostcert-epsilon.pem</CertificatePath>
    <CACertificatesDir>certs/CA</CACertificatesDir>
</ClientSSLConfig>
```

This can be put in a separate file, and then use the **FromFile** attribute of the ClientSSLConfig section to specify where the file is.

3.4 A-Hash

3.4.1 Functionality

The A-Hash is a database for storing string tuples, and it provides conditional and atomic modification of them. The A-Hash can be used as a centralized service, or it can be deployed in a distributed way, using multiple nodes, where all the data is replicated on all the nodes.

The A-Hash stores *objects*, where each object has an arbitrary string *ID*, and contains any number of *property*-*value* pairs grouped in *section*s, where *property*, *value* and *section* are arbitrary strings. It can only be one *value* for a *property* in a *section*.

Given an ID, the *get* method returns property-value pairs of the corresponding object. By specifying a section or property, the *get* method returns the corresponding values. Using the *change* method, property-value pairs can be added and removed from an object, all occurrences of a property can be deleted and new objects can be created. By specifying conditions, the change is only applied if the given conditions are met.

3.4.2 Interface

get(IDs, neededMetadataList) returns all or some of the property-value pairs of the requested objects.

The *IDs* is a list of string *ID*'s, *neededMetadataList* is a list of (*section*, *property*) pairs. If the *neededMetadataList* is empty, then for each *ID* it returns all the *values* for all the *properties* in all the *section* in that given object. If there are sections and properties specified in *neededMetadataList*, then only those values are returned.

The response contains a list of *object*s, where an *object* is an (*ID*, *metadataList*) pair, where *metadataList* is a list of (*section*, *property*, *value*) tuples.

change(changeRequestList) modifies, removes or creates objects if certain conditions are met, and returns information about the success of the modification requests.

The *changeRequestList* is a list of *changeRequest*s, where a *changeRequest* is a tuple of (*changeID*, *ID*, *changeType*, *section*, *property*, *value*, *conditionList*):

- *changeID* is an arbitrary ID which is used in the response to refer to this part of the request
- *ID* points to the object to be change
- *changeType* can be '**set**' to set the property within the section to value, '**unset**' to remove the property from the section regardless of the value, '**delete**' to remove the whole object
- *conditionList* is a list of *condition*s, where a *condition* is a tuple of (*conditionID*, *conditionType*, *section*, *property*, *value*), where *conditionType* can be '**is**' which will be true if the property in the section is set to the value, '**isnot**' which will be true if the property in the section is not set to the value, '**isset**' which will be true if the property of the section is set regardless of the value or '**unset**' which will be true if the property of the section is not set at all.

If all conditions are met, the A-Hash tries to apply changes to the objects, and creates a new object if a previously non-existent ID is given.

The response contains the *changeResponseList* which is a list of (*changeID*, *success*, *failedConditionID*) tuples. The *success* of the change is either **set**, **unset**, **deleted**, **failed**, **condition not met** (in which case the ID of the failed condition is put into *failedConditionID*), **invalid change type** or **unknown**.

3.4.3 Implementation

The A-Hash service has a modular architecture which means that it can be deployed in a centralized or a distributed way by simply specifying different modules in the service's configuration. There are different low-level modules for storing the metadata on disk, e.g. serialized into the 'pickle' format. In addition there is a module which stores the metadata replicated on multiple nodes.

The replicated A-Hash module is built on Oracle Berkeley DB High Availability, an open source database library with a replication API. The replication is based on a single master, multiple clients framework where

all clients can read from the database, while only the master is allowed to write to the database. This ensures that all database transactions are ACID (atomic, consistent, isolated and durable). The replicated A-Hash module consists of three modules, `ReplicatedAHash`, `ReplicationStore` and `ReplicationManager`.

ReplicatedAHash is a subclass of the `CentralizedAHash` business logic class and implements the extra logic needed for communication between A-Hash instances. The `ReplicatedAHash` adds the method `sendMessage` to the A-Hash interface and has a method `processMessage` for processing messages from `sendMessage`. These two methods are needed for the communication framework of the `ReplicationManager`. The actual replication is managed by the `ReplicationStore` and `ReplicationManager`, meaning that the `ReplicatedAHash`, with the exception of the above-mentioned methods, is actually just a centralized A-Hash with a specific storage module.

ReplicationStore is a specialized replicated database. It is a subclass of `TransDBStore`[3] with some additional features for handling replication. The `ReplicationStore` periodically updates an A-Hash object which contains information about the current master A-Hash and available clients. In addition it contains a `ReplicationManager` object which governs the replication framework.

ReplicationManager handles all aspects of the Berkeley DB API and takes care of replicas, elections of the master replica and replication messages.

- The replicated A-Hash consists of a set of replicas; one master and multiple clients. This means that the entire database is fully replicated on all replicas. When the master receives a write message, it will broadcast this message to all the clients. To make sure that the database is consistent and that it will always be possible to elect a master, the master requires an acknowledgement message from a quorum of clients before continuing.
- In the replicated A-Hash there is one master and multiple clients. When the A-Hash starts up or if a master goes offline, a new master must be elected. This is solved by the clients holding an election, according to the Paxos algorithm[4]. Simply told, all clients send a vote to all other clients, and the eligible client that gets the most votes wins the election. An eligible client is a client which has received the latest updates. See the Berkeley DB documentation for more details. There are cases where some servers are more suitable to be master than others (better hardware, faster Ethernet, etc.). In such cases the election can be influenced by setting a higher priority on the best servers. While an election is being held, it is not possible to write to or read from the A-Hash, as consistency cannot be guaranteed without a master.
- As the replicas in the replicated A-Hash are services running within HED, the replication messages need to be sent through the communication framework of HED. To be able to use this framework, the Berkeley DB needs a callback function for sending messages and the `ReplicationManager` needs to call Berkeley DB to process incoming messages. The `ReplicationMessage` takes a send function as one of the initialization arguments, and calls this function in the callback method. By providing the `ReplicatedAHash.sendMessage` to the `ReplicationManager` messages will be sent through the HED communication services just as in regular inter service communication. When `ReplicatedAHash` receives a replication message from `sendMessage` it will call `ReplicationManager.processMessage` which in turn will call the Berkeley DB processMessage. This communication framework is both used by Berkeley DB and directly by the `ReplicationManager`, e.g., for bootstrapping the A-Hash replicas and to communicate newly discovered clients.

3.4.4 Configuration

The A-Hash has the following configuration variables:

AHashClass tells the A-Hash service the name of the business-logic A-Hash class. Currently there are two implementations, the `CentralAHash` is a centralized version, and the `ReplicatedAHash` is a replicated version.

[3] A module for storing data on disk transactionally using the Berkeley DB library.
[4] http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-paxos

StoreClass specifies the name of the store class, used to store the data on disk. Currently there are several store classes each has the same interface but uses different mechanisms (`PickelStore`, `CachedPickleStore`, `StringStore`, `ZODBStore`, `TransDBStore`), however only the `TransDBStore` class is suitable for use in a replicated deployment. All the others are only for a centralized deployement, so this variable is only configurable for the centralized deployement. Based on our tests, currently the `CachedPickleStore` has the best performance.

StoreCfg contains parameters for the different store classes, and it almost always contains a *DataDir* parameter which specifies a directory on the (local) filesystem where the A-Hash can save its files. This variable is only configurable for the centralized deployment.

For the replicated A-Hash there are some additional mandatory configuration variables:

LocalDir specifies a directory on the local filesystem where this A-Hash replica should be stored. This directory must be unique for this replica, i.e., no two A-Hash replicas can share the same `LocalDir` since doing so will most likely leave both replicas broken beyond repair.

MyURL is the URL to this A-Hash service and is used as an ID for this service in the replication group.

OtherURL is the URL of another A-Hash in the replication group. More than one URL can be specified here.

Even though the default settings are useable, there are some optional variables available for the replicated A-Hash:

Priority is related to the master election. If a master dies, there will be an election to decide which client will be the new master. Higher priority means a higher chance to be elected. A priority of 0 means that this replica can never be elected. The defaults is 10.

CheckPeriod specifies in seconds how frequently the A-Hash should update the replica list. The default is 10 seconds.

CacheSize configures the amount of the storage to be cached in memory. The default is 10MB

Example of a centralized A-Hash configuration:

```
<Service name="pythonservice" id="ahash">
    <ClassName>storage.ahash.ahash.AHashService</ClassName>
    <AHashClass>storage.ahash.ahash.CentralAHash</AHashClass>
    <StoreClass>storage.store.cachedpicklestore.CachedPickleStore</StoreClass>
    <StoreCfg>
        <DataDir>ahash_data</DataDir>
    </StoreCfg>
</Service>
```

Example of a replicated A-Hash configuration:

```
<Service name="pythonservice" id="ahash1">
    <ClassName>storage.ahash.ahash.AHashService</ClassName>
    <AHashClass>storage.ahash.replicatedahash.ReplicatedAHash</AHashClass>
    <LocalDir>ahash_data1</LocalDir>
    <MyURL>http://localhost:60000/AHash1</MyURL>
    <OtherURL>http://localhost:60001/AHash2</OtherURL>
    <Priority>50</Priority>
    <ClientSSLConfig FromFile='clientsslconfig.xml'/>
</Service>
```

3.5 Librarian

3.5.1 Functionality

The Librarian knows about the type of possible entries in the namespace of Chelonia: files, collections and mount points. It knows that these are organized in a tree-hierarchy with entries grouped into collections, and it knows about Logical Names, and how to traverse a Logical Name to find the metadata of the file or collection (or mount point) which it refers to. However, the Librarian does not maintain the namespace of Chelonia. It does not put new files into collections, nor remove the link from its parent collection when a file is removed - this is the job of the Bartender. The only Librarian method which deals with Logical Names is the *traverseLN* method, which traverses the Logical Name as far as possible, and returns the GUID and metadata of the last found entry in the path of the Logical Name. All the other methods work with GUID's, and do not care about the hierarchical namespace. In this regard the Librarian can be treated as a higher level application-specifc interface to the metadata store (the A-Hash), because while the A-Hash does not care about what it stores, the Librarian knows exactly that it stores metadata about files, collections, etc. With this interface new files, collections and mount points can be created with the *new* method, the metadata of any entry can be retrieved and modified with the *get* and *modifyMetadata* methods, and entries can be removed with the *remove* method.

Besides being an interface to the metadata store, the Librarian has another important functionality: it provides a way for Shepherds to register themselves and to periodically send reports about the state of their stored files. A Shepherd sends a *report* message to a Librarian if e.g. a new file arrived, or an old file gets corrupted, thus the Librarian can keep the states of the replicas up-to-date. The Librarian monitors the registered Shepherds, and if one of them stops sending reports, the Librarian will assume, that it is offline, and modify the states of all the replicas which are on the Shepherd's storage node. In order to do this, the Librarian stores for each Shepherd a list of files on the given storage nodes, this make it possible to know which files have replicas on a given storage nodes without checking all the files in the system. This list of files gets updated every time a Shepherd sends a report.

The end-user of Chelonia never communicates directly with a Librarian, the Bartenders will contact the Librarians in order to fulfil the user's requests.

3.5.2 Interface

get(getRequestList, neededMetadataList) returns all or some of the metadata of the requested GUID's.

The *getRequestList* is a list of string *GUID*'s, *neededMetadataList* is a list of (*section*, *property*) pairs. If *neededMetadataList* is empty, then for each *GUID* all the metadata values for all the properties in all the sections are returned. If there are sections and properties specified in *neededMetadataList*, then only those values are returned.

The response contains a list of *GUID, metadataList* pairs, where *metadataList* is a list of (*section*, *property*, *value*) tuples.

new(newRequestList) creates a new entry in the Librarian with the given metadata, and returns the GUID's of the new entries.

The *newRequestList* is a list of (*requestID*, *metadataList*) where *requestID* is an arbitrary ID used to identify this request in the list of responses; *metadataList* is a list of (*section*, *property*, *value*) tuples. This method generates a *GUID* for each request, and inserts the new entry (with the given metadata) into the A-Hash, then returns the GUID's of the newly created entries. In the metadata of the new entry the 'type' property in the 'entry' section defines whether it is a file, a collection or a mount point. The 'GUID' property can contain a GUID if you want to specify the GUID of the new entry, and don't want the Librarian to generate a random one.

The response contains a list of (*requestID*, *GUID*, *success*) tuples.

modifyMetadata(modifyMetadataRequestList) modifies the metadata of the given entries.

modifyMetadataRequestList is a list of (*changeID*, *GUID*, *changeType*, *section*, *property*, *value*) tuples where *changeType* can be '**set**' to set the property in the section to value, '**unset**' to remove the property-value pair from the section, '**add**' to set the property in the section to *value* only if it does

not exist already or '**setifvalue**=value' to only set the property in the section to value if it currently equals to the 'value' in the *changeType*.

The response is a list of (*changeID*, *success*) pairs where *success* can be '**set**', '**unset**', '**condition failed**', '**failed**: reason'.

remove(removeRequestList) removes the given entries.

The *removeRequestList* is a list of (*requestID*, *GUID*) pairs.

The response is a list of (*requestID*, *success*) pairs where *success* is either '**removed**' or '**failed**: reason'.

traverseLN(traverseRequestList) traverses the given Logical Names and returns extensive information about them.

The *traverseRequestList* is a list of (*requestID*, *LN*) pairs with the Logical Names to be traversed

The response is a list of (*requestID*, *traversedList*, *wasComplete*, *traversedLN*, *GUID*, *metadataList*, *restLN*) tuples where:

traversedList is a list of (*LNpart*, *GUID*) pairs, where *LNpart* is a part of the *LN*, *GUID* is the GUID of the Librarian-entry referenced by that part of the *LN*, the first element of this list is the shortest prefix of the *LN*, the last element is the *traversedLN* without its last part

wasComplete indicates whether the full *LN* was traversed

traversedLN is the part of the *LN* which was traversed, if *wasComplete* is true, this should be the full *LN*

GUID is the *GUID* of the *traversedLN*

metadataList is all the metadata of the of traversedLN in the form of (*section*, *property*, *value*) tuples

restLN is the remainders of the *LN* which was not traversed for some reason, if *wasComplete* is true, this should be an empty string

report(serviceID, filelist) is a report message from a Shepherd to a Librarian which contains the ID of the Shepherd, and a list of changed files on the Shepherd's storage node.

The *filelist* is a list of (*GUID*, *referenceID*, *state*) tuples containing the state of changed or new files, where referenceID is the Sheperd-local ID of the given replica and GUID refers to the logical file of this replica. The *state* is one of '**invalid**' (if the periodic self-check of the Shepherd found a non-matching checksum or missing file), '**creating**' (if this is a new file not uploaded yet), '**thirdwheel**' (if the replica is marked for deletion because of too many replicas), '**alive**' (if the file is uploaded and the checksum is OK), or '**stalled**' (if the file was in 'creating' state for a long time but hasn't arrived).

The response is *nextReportTime*, a number of seconds, which is the timeframe within which the Librarian expects to receive the next heartbeat from the Shepherd.

3.5.3 Implementation

The Librarian service uses the A-Hash to store all the metadata, and it uses the GUID's as ID's in the database. It has the URL of one or more A-Hash services in its configuration, and it can acquire more A-Hash URL's from these first A-Hashes in case of a replicated A-Hash deployment.

The replicated A-Hash deployment is based on a single master, multiple clients scenario (see Section 3.4 for details). This means that the Librarian can read from any A-Hash, while it can only write to the master A-Hash. A list of A-Hashes, master and clients, is stored in an A-Hash object. The Librarian periodically retrieves this list to maintain its own list of A-Hash services. If the Librarian tries to write to the replicated A-Hash and the master is down, the Librarian will try to get a new list from one of the clients and retry the write request once before giving up. In case of reading from the replicated A-Hash, the Librarian will randomly loop through the list of clients until the read request is successful or until there are no more clients available.

The Librarian also accepts report messages from Shepherds, and stores the contents of these messages in the A-Hash. There is one A-Hash object which contains the ID and the expected time of the next heartbeat of the registered Shepherds, and there is one A-Hash object for each Shepherd which stores the states and GUID's of the files on the Shepherd's storage node (see Section 3.2.4 for details). The Librarian checks

periodically if there is a Shepherd which is late with its heartbeat, and it changes the states of all the replicas if needed.

Because the Librarian services store everything in the A-Hash, the Librarian itself is a stateless service. Multiple Librarians may be deployed. If they are configured to use the same replicated group of A-Hashes (or the same central A-Hash) then the Bartenders can use any Librarian to achieve the same results, and the Shepherds can report to any of them. One Shepherd should only report to one Librarian, but if that Librarian is offline, it should find another. The reports will be registered in the A-Hash as well, so it does not matter which Librarian receives a report.

3.5.4 Configuration

The Librarian has the following configuration variables:

AHashURL is the URL of an A-Hash which the Librarian should use. Multiple A-Hashes may be specified.

HeartbeatTimeout specifies in seconds how frequently the Sheperds should send reports to the Librarian.

CheckPeriod specifies in seconds how frequently the Librarian should check for late heartbeats.

An example configuration:

```
<Service name="pythonservice" id="librarian">
    <ClassName>storage.librarian.librarian.LibrarianService</ClassName>
    <AHashURL>https://localhost:60000/AHash</AHashURL>
    <HeartbeatTimeout>30</HeartbeatTimeout>
    <CheckPeriod>20</CheckPeriod>
    <ClientSSLConfig FromFile="clientsslconfig.xml"/>
</Service>
```

3.6 Shepherd

3.6.1 Functionality

A Shepherd service is capable of managing a storage node. It keeps track of all the files it stores with their GUID's and checksums. It periodically checks each file to detect corruptions, and sends reports to a Librarian indicating that the storage node is up and running, and whether the state of any files have been changed. If a file goes missing or has a bad checksum then the Librarian is notified about the error[5]. It periodically asks the Librarian how many replicas its files have, and if a file has fewer replicas than needed, the Shepherd offers its copy for replication by calling the Bartender.

A Shepherd service is always connected to a storage element service (e.g., a web server). For each supported storage element service there is a backend module which makes the Shepherd capable of communicating with the storage element to initiate file transfers, to detect whether a transfer was successful or not, to generate local ID's and checksums and so forth.

A file in a storage node is identified with a *referenceID* which is unique within that node. The *location* of a file consists of the *serviceID* of the Shepherd and the *referenceID*. If the location is known, the Shepherds *get* method can be called with the *referenceID* and a list of transfer protocols, the Shepherd chooses a protocol from this list, and creates a transfer URL (*TURL*). This TURL is returned together with the *checksum* of the file. A client tool may then download the file from this *TURL*, and verify it with the *checksum*. The client tool does not need to call the *get* method. The client tool simply queries the Bartender, which in turn calls the *get* method and returns the TURL.

Storing a file starts with initiating the transfer with the *put* method of the Shepherd, providing the file *size*, the *checksum* of the file and its *GUID*. The client tool also specifies a list of transfer protocols it is able to use, and the Shepherd chooses a *protocol*, creates a *TURL* for uploading and generates a *referenceID*, so that the client tool can upload the file to the TURL. Again, the client tool just asks the Bartender, and gets the TURL; the client tool does not need to call the *put* method of the Shepherd directly.

These *TURL*'s are one-time URL's which means that after the client tool uploads or downloads the file these *TURL*'s cannot be used again to do the same. If the same file is to be downloaded twice, the transfer has to be initiated twice, yielding in two different *TURL*'s.

The *stat* method provides information about a replica, e.g., checksum, GUID, state, etc. The *delete* method removes the replica.

In normal operation the *put* and *get* calls are made by a Bartender but the actual upload and download is done by the client tool. In the case of replication a Shepherd with a valid replica initiates the replication. This Shepherd asks the Bartender to choose a new Shepherd, the Bartender initiates putting the new replica on a chosen Shepherd and receives the TURL, and the Bartender returns the TURL to the initiator Shepherd, which uploads its replica to the given TURL.

The replicas have a state, which is one of '**creating**' when the transfer is initiated but the file is not uploaded yet, '**alive**' if the file is uploaded and has a proper checksum, '**invalid**' if it does not exists anymore or has a bad checksum, '**thirdwheel**' if the replica is marked for deletion because the file has too many replicas, or '**stalled**' if it has been in the 'creating' state for a long time but hasn't arrived.

3.6.2 Interface

get(getRequestList) initiates a download and returns the TURL

The *getRequestList* is a list of (*requestID*, *getRequestData*) pairs where *requestID* is an arbitrary ID used in the reply, *getRequestData* is a list of (*property*, *value*) pairs, where mandatory properties are: '**referenceID**' which refers to the file to get and '**protocol**' indicates a protocol the client can use. getRequestData may contain multiple protocols.

The response is a list of (*requestID*, *getResponseData*), where *getResponseData* is a list of (*property*, *value*) pairs, where *property* may be one of the following:

- '**TURL**' is a transfer URL which can be used by the client to download the file

[5]Here the Shepherd refers to the file with its GUID, that's why it needs to store the GUID's of its files.

- '**protocol**' is the protocol of the TURL; '**checksum**' is the checksum of the replica
- '**checksumType**' is the name of the checksum method
- '**error**' contains an error message if there is one.

put(putRequestList) initiates an upload and returns the TURL.

putRequestList is a list of (*requestID*, *putRequestData*) pairs where *requestID* is an ID used for the response, *putRequestData* is a list of (*property*, *value*) pairs where property can be one of '**GUID**', '**checksum**', '**checksumType**', '**size**' (the size of the file in bytes), '**protocol**' (a protocol the client can use, can be multiple) and '**acl**' (which is currently not used).

The response is a list of (*requestID*, *putResponseData*) pairs, where *putResponseData* is a list of (*property*, *value*) pairs where *property* may be one of the following:

- '**TURL**' is the transfer URL where the client can upload the file
- '**protocol**' is the chosen protocol of the TURL
- '**referenceID**' is the generated ID for this new replica
- '**error**' contains an error message.

delete(deleteRequestList) removes a replica.

The *deleteRequestList* is a list of (*requestID*, *referenceID*) pairs containing the ID's of the files to remove.

The response is a list of (*requestID*, *status*) pairs, where *status* can be either '**deleted**' or '**nosuchfile**'.
.

stat(statRequestList) returns information about replicas.

statRequestList is a list of (*requestID*, *referenceID*) pairs where *referenceID* points to the file whose metadata are requested.

The response is a list of (*requestID*, *referenceID*, *state*, *checksumType*, *checksum*, *acl*, *size*, *GUID*, *localID*) tuples.

3.6.3 Implementation

The Shepherd communicates with storage element services via backend modules. Currently there are two backend modules implemented, one for the *Hopi* service[6] (which is a simple HED-based HTTP server), one for the *Apache* webserver[7].

In both cases the Shepherd and the transfer services should have access to the same local filesystem where the Shepherd creates two separate directories: one for storing all the files (e.g. `./store`) and one for the file transfers (e.g. `./transfer`). The store directory always contains all the files the Shepherd manages, the transfer directory is empty at the beginning.

If a client asks for a file called `file1`, and this file is in the store directory (`./store/file1`), the Shepherd service creates a hardlink into the transfer directory (`./transfer/abc`) and sets this file read-only. If the Hopi service is configured to serve the HTTP path `/prb` and it is serving files from the directory `./transfer` then after the hardlink is created, the transfer URL for this file reads `http://localhost:60000/prb/abc`. Now this URL is passed to the client. Then the client `GET`'s this URL and gets the file. The Hopi service removes (unlinks) this file immediately after the `GET` request arrived, which makes `http://localhost:60000/prb/abc` invalid (so this is a one-time URL). However, because of the hardlink the file is still in the store directory, it is just removed from the transfer directory. Now if some other user wants this file, the Shepherd creates an other hardlink, e.g., `./transfer/qwe` with the URL `http://localhost:60000/prb/qwe`.

If a client wants to upload a new file, the Shepherd creates an empty file in the store directory, e.g. `./store/file2` and creates a hardlink in the transfer directory, e.g., `./transfer/oiu` and makes it writable. The transfer URL is then `http://localhost:60000/prb/oiu`, and the client can do a `PUT` on this URL. When the client tool `PUT`s the file there, the Hopi service immediately removes the uploaded file from the transfer directory, but because it has a hardlink in the store directory, the file is stored there as `./store/file2`. The backend

[6] Note that the Hopi service needs to be configured in a special 'slave' mode to be used as a backend for the Shepherd.
[7] `http://www.apache.org/`

module for the Hopi service periodically checks whether a new file has two hardlinks or just. If it has only one hardlink that means that the file is uploaded, so it will notify the Shepherd that the file has arrived. In order to do that, the Shepherd needs to provide a callback method 'file_arrived' to the backend module.

All the backend modules should have the following common interface which the Shepherd can use to communicate with the storage element services:

prepareToGet(referenceID, localID, protocol) returns a one-time *TURL*.

Initialize transfer with *protocol* for the file identified by (*localID*, *referenceID*). The reason for including here the referenceID as well is that this information may be used by the backend module later, e.g., when the transfer has finished and the state of the file needs to be changed.

prepareToPut(referenceID, localID, protocol) returns a one-time *TURL*.

Initialize transfer with *protocol* for the file identified by (*localID*, *referenceID*).

copyTo(localID, turl, protocol) returns *success*.

Upload the file referenced by *localID* to the given *TURL* with the given *protocol*.

copyFrom(localID, turl, protocol) returns *success*.

Download the file from the given *TURL* with the given *protocol*, and store it as *localID*.

list() returns a list of *localID*'s currently in the store directory.

getAvailableSpace() returns the available disk space in bytes.

generateLocalID() returns a new unique *localID*.

matchProtocols(protocols) only leave those protocols in the list *protocols* which are supported by this file transfer service.

checksum(localID, checksumType) generates a *checksumType* checksum of the file referenced by *localID*.

3.6.4 Configuration

The Shepherd has the following configuration variables:

ServiceID is the ID of the service, currently it should be the URL of the service, because there is no information system where the services can be discovered by ID's.

CheckPeriod specifies in seconds how frequently the Shepherd should check the existence and checksum of all the files on the storage node.

MinCheckInterval specifies in seconds how much time the should Shepherd wait between two subsequent file checks. This ensures that the Shepherd does not use up all the resources of the hosting machine. After checking the checksum of a file, the Shepherd will always wait this long before checking the next file, even if this means that checking all the files takes longer than the 'CheckPeriod' (in other words 'MinCheckInterval' overrides 'CheckPeriod' if the number of stored file is greater.)

CreatingTimeout specifies in seconds the time before the Shepherd decides that a started file upload has failed. This is not fatal, if the file is uploaded eventually, it will be a valid replica, but after this much time the system starts creating new replicas just in case.

StoreClass specifies which type of store the Shepherd should use for storing its metadata on disk (see Section 3.4.4 for a list of choises).

StoreCfg specifies configuration parameters for the store class. These parameters depends on the StoreClass, but will in most cases include *DataDir*, i.e., the local directory to be used.

BackendClass specifes which backend the Shepherd should use to communicate with the storage element service.

BackendCfg specifies configuration parameters for the backend class: **DataDir** and **TransferDir** are the directories the backend will use (see Section 3.6.3), **TURLPrefix** is the URL of the storage element service ending with a '/'.

LibrarianURL is the URL of a Librarian where the Shepherd should send the reports about the health of the stored files.

BartenderURL is the URL of a Bartender to which the Shepherd should connect if it wants to offer a replica for replication or recover a corrupted replica.

An example configuration:

```
<Service name="pythonservice" id="shepherd">
    <ClassName>storage.shepherd.shepherd.ShepherdService</ClassName>
    <ServiceID>https://localhost:60000/Shepherd</ServiceID>
    <CheckPeriod>20</CheckPeriod>
    <MinCheckInterval>0.1</MinCheckInterval>
    <CreatingTimeout>600</CreatingTimeout>
    <StoreClass>storage.store.cachedpicklestore.CachedPickleStore</StoreClass>
    <StoreCfg>
        <DataDir>./shepherd_data1</DataDir>
    </StoreCfg>
    <BackendClass>storage.shepherd.hardlinkingbackend.HopiBackend</BackendClass>
    <BackendCfg>
        <DataDir>./shepherd_store</DataDir>
        <TransferDir>./shepherd_transfer</TransferDir>
        <TURLPrefix>https://localhost:60000/hopi/</TURLPrefix>
    </BackendCfg>
    <LibrarianURL>https://localhost:60000/Librarian</LibrarianURL>
    <BartenderURL>https://localhost:60000/Bartender</BartenderURL>
    <ClientSSLConfig FromFile="clientsslconfig.xml"/>
</Service>
```

3.7 Bartender

3.7.1 Functionality

The Bartender provides an easy to use interface of Chelonia to the users. The user can put, get and delete files using their logical names (*LN*'s) with the *putFile*, *getFile* and *delFile* methods, create, remove and list collections with *makeCollection*, *unmakeCollection* and *list*. The link to a file or sub-collection can be removed from its parent collection without removing the file or sub-collection itself with the *unlink* method. The metadata of a file or collection (e.g., whether the collection is closed, number of needed replicas, access policies) can be changed with *modify*. A *stat* call gives all the information about a file or collection, and collections and files can be moved (or hardlinks be created) within the namespace with *move*. An entirely new replica of a file can be uploaded (e.g., if the file lost all its replicas, or when a Shepherd service offers its replica for replications) with *addReplica*. Mount points can be created and removed with the *makeMountpoint* and *unmakeMountpoint* methods.

The Bartender communicates with Librarians to get and modify the metadata of files and collections, and it communicates with the Shepherds to initiate file transfers. The Bartender also has *gateway modules* which are capable of communicating with different kinds of third-party storage solutions. With these gateway modules, it is possible to create mount points within the namespace of Chelonia, and to access the namespace of the third-party storage element through these mount points, which means the user can use the Bartender to get listing of directories on the third-party storage element or to get files from the third-party storage element. Thus the user can use a single client tool to easily access different kind of storage elements.

The high-level authorization is done by the Bartender. It makes decisions based on the identity of the connecting client and the access policy rules of the files and collections. The Bartender uses the security framework of the HED to evaluate the requests and the policies which are generated by the Bartender from the internal representation of the access rules. For more details see Section 3.3.

3.7.2 Interface

putFile(putFileRequestList) creates new files at the requested Logical Names, chooses a Shepherd and initializes the file transfer, and returns a TURL for each new file. The actual file transfer is left to the client tool.

The *putFileRequestList* is a list of (*requestID*, *LN*, *metadata*, *protocols*) tuples, where *requestID* is an arbitrary ID which will be used in the response, *LN* is the requested Logical Name of the new file, *protocols* is a list of supported protocols for uploading and *metadata* is a list of (*section*, *property*, *value*) tuples which should contain the following mandatory properties in the '**states**' section: '**size**', '**checksum**', '**checksumType**', and '**neededReplicas**'.

The response is a list of (*requestID*, *success*, *TURL*, *protocol*) tuples, where *TURL* is a URL with a chosen *protocol* which can be used to upload the file, the *success* string can be one of '**done**', '**missing metadata**', '**parent does not exists**' and '**internal error:** reason'.

getFile(getFileRequestList) finds the files referenced by the given logical name, chooses a replica and initializes the download, and returns a TURL for each file.

The *getFileRequestList* is a list of (*requestID*, *LN*, *protocols*) tuples where *requestID* is used in the response, *LN* is the Logical Name referring to the requested file, *protocols* is a list of supported transfer protocols.

The response is a list of (*requestID*, *success*, *TURL*, *protocol*), where *TURL* is the transfer URL using *protocol*, with which the file can be downloaded, *success* is one of '**done**', '**not found**', '**is not a file**', '**file has no valid replica**' and '**error while getting TURL:** reason'.

delFile(delFileRequestList) removes the files references by the given logical names.

The *delFileRequestList* is a list of (*requestID*, *LN*) pairs where the *LN*'s are the logical names of the files to be deleted.

The response is a list of (*requestID*, *status*), where the *status* is either '**deleted**' or '**nosuchLN**'.

unlink(unlinkRequestList) remove a link from a collection without deleting the file or sub-collection itself.

The *unlinkRequestList* is a list of (*requestID*, *LN*) pairs with the logical names of the files and sub-collections to be unlinked.

The response is a list of (*requestID*, *success*) pairs where *success* is one of '**unset**', '**no such LN**', '**denied**' or '**nothing to unlink**'.

stat(statRequestList) returns all metadata of files, collections or mount points.

statRequestList is a list of (*requestID*, *LN*) pairs with the logical names of enntries whose metadata should be retrieved.

The response is a list of (*requestID*, *metadata*) pairs, where *metadata* is a list of (*section*, *property*, *value*) tuples (see the data model in Section 3.2).

makeCollection(makeCollectionRequestList) creates new collections.

makeCollectionRequestList is a list of (*requestID*, *LN*, *metadata*) tuples where *metadata* is a list of (*section*, *property*, *value*) tuples. The '**entries**' section may contain the initial content of the catalog in the form of name-GUID pairs (these entries will be hard links to the given GUID's with the given name) and in the '**states**' section there is a '**closed**' property (if it is 'yes' then no more files can be added or removed later).

The response is a list of (*requestID*, *success*) pairs, where *success* is one of '**done**', '**LN exists**', '**parent does not exist**', '**failed to create new catalog entry**', '**failed to add child to parent**' or '**internal error**'.

unmakeCollection(unmakeCollectionRequestList) deletes empty collections.

unmakeCollectionRequestList is a list of (*requestID*, *LN*) pairs with the logical names of the collections to be removed.

The response is a list of (*requestID*, *success*) pairs, where *success* is one of '**removed**', '**no such LN**', '**collection is not empty**' or '**failed**: reason'.

list(listRequestList, neededMetadata) returns the contents of the requested collections.

listRequestList is a list of (*requestID*, *LN*) pairs where *LN* is the logical mame of the collection to be listed, *neededMetadata* is a list of (*section*, *property*) pairs which filters the returned metadata.

The response is a a list of (*requestID*, *entries*, *status*) tuples where entries is a list of (*name*, *GUID*, *metadata*) where *metadata* is a list of (*section*, *property*, *value*) tuples (see the data model in Section 3.2). The *status* is one of '**found**', '**not found**' or '**is a file**'.

move(moveRequestList) moves file or collections within the namespace which changes the Logical Name of the file or collection. This method only alters metadata, it does not move real file data.

moveRequestList is a list of (*requestID*, *sourceLN*, *targetLN*, *preserveOriginal*) tuples where *sourceLN* is the logical name referring to the file or collection to be moved (renamed) and *targetLN* is the new path. If *preserveOriginal* is true the *sourceLN* is kept, so with *preserveOriginal* a hard link is in fact created.

The response is a list of (*requestID*, *status*) pairs, where status is one of '**moved**', '**nosuchLN**', '**targetexists**', '**invalidtarget**', '**failed adding child to parent**' or '**failed removing child from parent**'.

modify(modifyRequestList) modifies the metadata of files, collections and mount points. Only the 'states', the 'policy' and the 'metadata' sections may be modified, and there are separate access control actions for each of the three.

The *modifyRequestList* is a list of (*changeID*, *LN*, *changeType*, *section*, *property*, *value*) where *changeType* is one of '**set**', set the *property* in the *section* to *value*, '**unset**', remove the *property-value* pair from the *section* or '**add**', set the *property* in the *section* to *value* only if it is not exists already.

The response is a list of (*changeID*, *success*), where *success* is on of '**set**', '**unset**', '**entry exists**' (for an 'add' request), '**denied**', '**no such LN**' or '**failed**: reason'.

makeMountpoint(makeMountpointRequestList) creates a mountpoint which is used to provide access to a third-party storage system within the global namespace.

makeMountpointRequestList is a list of (*requestID*, *LN*, *URL*, *metadata*) tuples where *LN* is the requested logical name, *URL* is the URL to the third-party storage system to be accessed, and *metadata* is a list of (*section*, *property*, *value*) tuples which could contain additional metadata..

The response is a list of (*requestID*, *success*) pairs, where *success* is one of '**done**', '**LN exists**', '**parent does not exist**', '**failed to create new catalog entry**', '**failed to add child to parent**' or '**internal error**' or '**cannot create anything in mountpoint**'.

unmakeMountPoint(unmakeMountpointRequestList) deletes mountpoints.

unmakeMountpointRequestList is a list of (requestID, LN) pairs where *LN* is the logical names of the mountpoint.

The response is a list of (*requestID*, *status*), where status could be '**removed**', '**no such LN**', '**denied**'.

removeCredentials() removes a previously delegated proxy certificate.

This method has no arguments, it removes the proxy certificate which is previously delegated by the same user.

The response is a (*message*, *status*) pair, where *status* could be '**successful**' or '**failed**', and *message* contains additional details.

The Bartender implements the **DelegateCredentialsInit** and **UpdateCredentials** methods as well to be able to accept credentials delegation. These two operations are described in this WSDL document: http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/libs/delegation/delegation.wsdl

3.7.3 Implementation

To manage collections the Bartender only needs to communicate with a Librarian service. To manage files it needs to communicate with Shepherd services as well. When a Bartender wants to create a new replica of a file (or the first replica of a new file) it should find available Shepherd services. The Bartender uses the list of Shepherd services registered to the Librarian, and chooses the locations of the replicas randomly from this list according a uniform distribution.

The Bartender utilizes *Gateway* modules to access file listings and transfer URLs from third-party storage systems. The gateway module has four basic methods: *get*, *put*, *list* and *remove*. This module connects to the third-party storages on behalf of the user, in order to do this, the Bartender needs to have proxy certificates from the user. A proxy certificate can be passed to the Bartender through the process of credential delegation. Users can also remove their proxy certificate from the service side.

3.7.4 Configuration

The Bartender has the following configuration variables:

LibrarianURL specifies the URL of one or more Librarian services.

ProxyStore is a local directory where the Bartender will store the delegated proxies. This directory should be as secure as possible.

GatewayClass is the name of the gateway module to be used in this Bartender.

GatewayCfg contains the configuration parameters of the gateway class: **ProxyStore** is the directory where the gateway module can find delegated proxies, and **CACertificatesDir** is a directory with the certificates of the trusted CAs.

An example configuration:

```
<Service name="pythonservice" id="bartender">
    <ClassName>storage.bartender.bartender.BartenderService</ClassName>
    <LibrarianURL>https://sal1.uppmax.uu.se:60000/Librarian</LibrarianURL>
    <ProxyStore>/var/arc/spool/proxy_store</ProxyStore>
```

```
        <GatewayClass>storage.bartender.gateway.gateway.Gateway</GatewayClass>
        <GatewayCfg>
            <ProxyStore>/var/arc/spool/proxy_store</ProxyStore>
            <CACertificatesDir>/etc/grid-security/certificates</CACertificatesDir>
        </GatewayCfg>
        <ClientSSLConfig FromFile='clientsslconfig.xml'/>
    </Service>
```

3.8 Client tools

3.8.1 Prototype CLI tool

In the prototype release there is a command-line client tool called `arc_storage_cli`, which is written in Python, and only need a basic Python installation to run. It is capable of communicating with a given Bartender service, and can upload and download files via HTTP. However, in order to use proxy certificates or to transfer files with the GridFTP protocol, the ARC client libraries must also be installed.

The credentials of the user and the URL of the Bartender must be configured either with an XML file located at `~/.arc/client.xml` or with environment variables. (The environment variables overwrite the values from the XML file.)

An example of user credentials and Bartender configuration in the `~/.arc/client.xml` file:

```
<ArcConfig>
  <ProxyPath>/Users/zsombor/Development/arc/proxy</ProxyPath>
  <CACertificatesDir>/Users/zsombor/Development/arc/certs/CA</CACertificatesDir>
  <BartenderURL>https://localhost:60000/Bartender</BartenderURL>
</ArcConfig>
```

An example of user credentials and Bartender configuration with environment variables:

```
export ARC_BARTENDER_URL=https://localhost:60000/Bartender
export ARC_KEY_FILE=/Users/zsombor/Development/arc/certs/userkey-john.pem
export ARC_CERT_FILE=/Users/zsombor/Development/arc/certs/usercert-john.pem
export ARC_CA_DIR=/Users/zsombor/Development/arc/certs/CA
```

The number of needed replicas for new files can be specified with the `ARC_NEEDED_REPLICAS` environment variable. The default is one replica.

The `arc_storage_cli` has its built-in help. The methods can be listed by giving the command without arguments:

```
$ arc_storage_cli
Usage:
  arc_storage_cli <method> [<arguments>]
  Supported methods: stat, make[Collection], unmake[Collection], list, move,
    put[File], get[File], del[File], pol[icy], unlink,
    credentialsDelegation, removeCredenstials
```

Without arguments, each method prints its own help, e.g., for the move method:

```
$ arc_storage_cli move
Usage: move <sourceLN> <targetLN>
```

Here is an example of uploading, stating and downloading a file:

```
$ cat testfile
This is a testfile.
$ arc_storage_cli put testfile /tmp/
- The size of the file is 20 bytes
- The md5 checksum of the file is 9a9dffa22d227afe0f1959f936993a80
- Calling the Bartender's putFile method...
- done in 0.08 seconds.
- Got transfer URL: http://localhost:60000/hopi/d15900f5-34ee-4bba-bb10-73d60d1c0d75
- Uploading from 'testfile'
    to 'http://localhost:60000/hopi/d15900f5-34ee-4bba-bb10-73d60d1c0d75' with http...
```

```
Uploading 20 bytes... data sent, waiting... done.
- done in 0.0042 seconds.
'testfile' (20 bytes) uploaded as '/tmp/testfile'.
$ arc_storage_cli stat /tmp/testfile
- Calling the Bartender's stat method...
- done in 0.05 seconds.
'/tmp/testfile': found
  states
    checksumType: md5
    neededReplicas: 1
    size: 20
    checksum: 9a9dffa22d227afe0f1959f936993a80
  timestamps
    created: 1210232135.57
  parents
    51e12fab-fd3d-43ec-9bc5-17041da3f0b2/testfile: parent
  locations
    http://localhost:60000/Shepherd fc0d3d99-6406-4c43-b2eb-c7ec6d6ab7fe: alive
  entry
    type: file
$ arc_storage_cli get /tmp/testfile newfile
- Calling the Bartender's getFile method...
- done in 0.05 seconds.
- Got transfer URL: http://localhost:60000/hopi/dab911d0-110f-468e-b0c3-627af6e3af31
- Downloading from 'http://localhost:60000/hopi/dab911d0-110f-468e-b0c3-627af6e3af31'
    to 'newfile' with http...
Downloading 20 bytes... done.
- done in 0.0035 seconds.
'/tmp/testfile' (20 bytes) downloaded as 'newfile'.
$ cat newfile
This is a testfile.
```

3.8.2 FUSE module

ARCFS, a Filesystem in Userspace (FUSE) module, provides high-level access to the storage system. FUSE provides a simple library and a kernel-userspace interface. Using FUSE and the ARC Python interface, ARCFS allows users to mount the storage namespace into the local namespace, enabling the use of operating system commands and tools such as graphical file browsers. Delegation handling is not yet implemented in the fuse module, so that to change ownership, group access, etc., the `arc_storage_cli` tool must be used.

As for the `arc_storage_cli` the credentials of the user and the URL of the Bartender can be configured either with an XML file located at `~/.arc/client.xml` or with environment variables. See Section 3.8.1 for more details.

ARCFS consists of a Python script, `arcfs.py`. It requires the ARC Python interface, the FUSE library and pyFUSE, a Python wrapper to FUSE. `arcfs.py` takes only one input argument which is the mount point where Chelonia should be mounted.

An example of uploading, downloading and listing files in Chelonia mounted by FUSE:

```
$ python arcfs.py ./mnt
$ cat testfile
This is a test file
$ mkdir mnt/tmp
$ cp testfile mnt/tmp
$ ls -l mnt/tmp/testfile
-rw-r--r-- 1 jonkni jonkni 20 2009-03-21 23:27 mnt/tmp/testfile
$ cp mnt/tmp/testfile newfile
$ cat newfile
```

```
This is a test file
$ cat mnt/tmp/testfile
This is a test file
$ fusermount -u ./mnt
$ arc_storage_cli stat /tmp/testfile
- Calling the Bartender's stat method...
- done in 0.05 seconds.
'/tmp/testfile': found
  states
    checksumType: md5
    neededReplicas: 1
    size: 20
    checksum: 9a9dffa22d227afe0f1959f936993a80
  timestamps
    created: 1210232135.57
  parents
    51e12fab-fd3d-43ec-9bc5-17041da3f0b2/testfile: parent
  locations
    http://localhost:60000/Shepherd fc0d3d99-6406-4c43-b2eb-c7ec6d6ab7fe: alive
  entry
    type: file
```

3.9 Grid integration

To access data through the ARC middleware client tools, one needs to go through Data Manager Components (DMC's). These are protocol specific plugins to the client tools. For example, to access data from a HTTPS service, the HTTP DMC will be used with a URL starting with `https://`, to access data from an SRM service, the SRM DMC will be used with a URL starting with `srm://`. Similarly, to access Chelonia, the ARC DMC will be used with a URL starting with `arc://`.

As with the `arc_storage_cli` command the credentials of the user and the URL of the Bartender can be configured with an XML file located at `~/.arc/client.xml`. When the ARC DMC is initiated, it will create an instance of `DataPointARC` which provides functionality for listing collections and file IO. When downloading/uploading files from/to the storage system, the data point will use the provided Bartender URL to acquire a TURL from a Bartender. This TURL will be given to a `DataTransfer` handle. Calling the constructor of the DataHandle class with a URL will trigger the iGetDataPoint class of all registered DMC's to be called until a non-NULL pointer is returned. This means that as long as there exists a data point plugin for the protocol of the returned TURL, the transfer will proceed.

An example of uploading, downloading and listing a file with the ARC client tools:

```
$ arccp arc:///tmp/testfile fusefile
$ cat fusefile
This is a test file
$ arccp testfile arc:///tmp/testfile2
$ arcls -l arc:///tmp/testfile2
testfile file * * * *
$ arccp arc:///tmp/testfile2 newfile2
$ cat newfile2
This is a test file
```

Note here that long listing (`arcls -l`)! with the `arc` protocol not yet supports time stamps. Hence, only the type of the entry (here 'file') is shown.

The ARC DMC allows Grid jobs to access Chelonia directly. As long as A-REX, the job execution service of ARC, and ARC DMC are installed on a site, files can be both downloaded and uploaded by specifying the corresponding URL's in the job description. In this case, the Bartender URL needs to be embedded in the URL as a URL option. For example, if a job requires an input file `/user/me/input.dat`, and a Bartender `https://storage/Bartender` the URL specified in the job description will be as follows:

```
arc:///user/me/input.dat?BartenderURL=https://storage/Bartender
```