NORDUGRID-TECH-17
2/3/2009

Documentation of the ARC storage system

First prototype status and plans

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

The ARC storage system is a distributed system for storing replicated *file*s on several file storage nodes and manage them in a global namespace. The files can be grouped into *collection*s (a concept very similar to directories in the common file systems), and 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 they were using a local filesystem. Files can be transfered by multiple transfer protocols, and 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 storage element service running (e.g. HTTP(S), FTP(S), GridFTP, ByteIO[1], etc.). For each storage node one of the services of the ARC storage system is needed to manage it and to integrate it into the system. There is a way on the client side to access third-party storage solutions through the namespace of the ARC storage system. 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 sysem called the *GUID*. Compared to the well-known structure of local file systems, these GUIDs 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 on the ARC storage is just a list of name and GUID pairs. There is a dedicated collection which is the *root collection*. This makes the namespace of the ARC storage system 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`.

Besides the Logical Names we can refer to a file or collection by simply its GUID, or we can use GUIDs and Logical Names together, as seen on Figure 1.2.

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

Example on Figure 1.2: if we have a collection with GUID `1234`, and there is a collection called `green` in it, and in `green` there is another collection called `orange`, and in `orange` there is a file called `huge`, then we

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

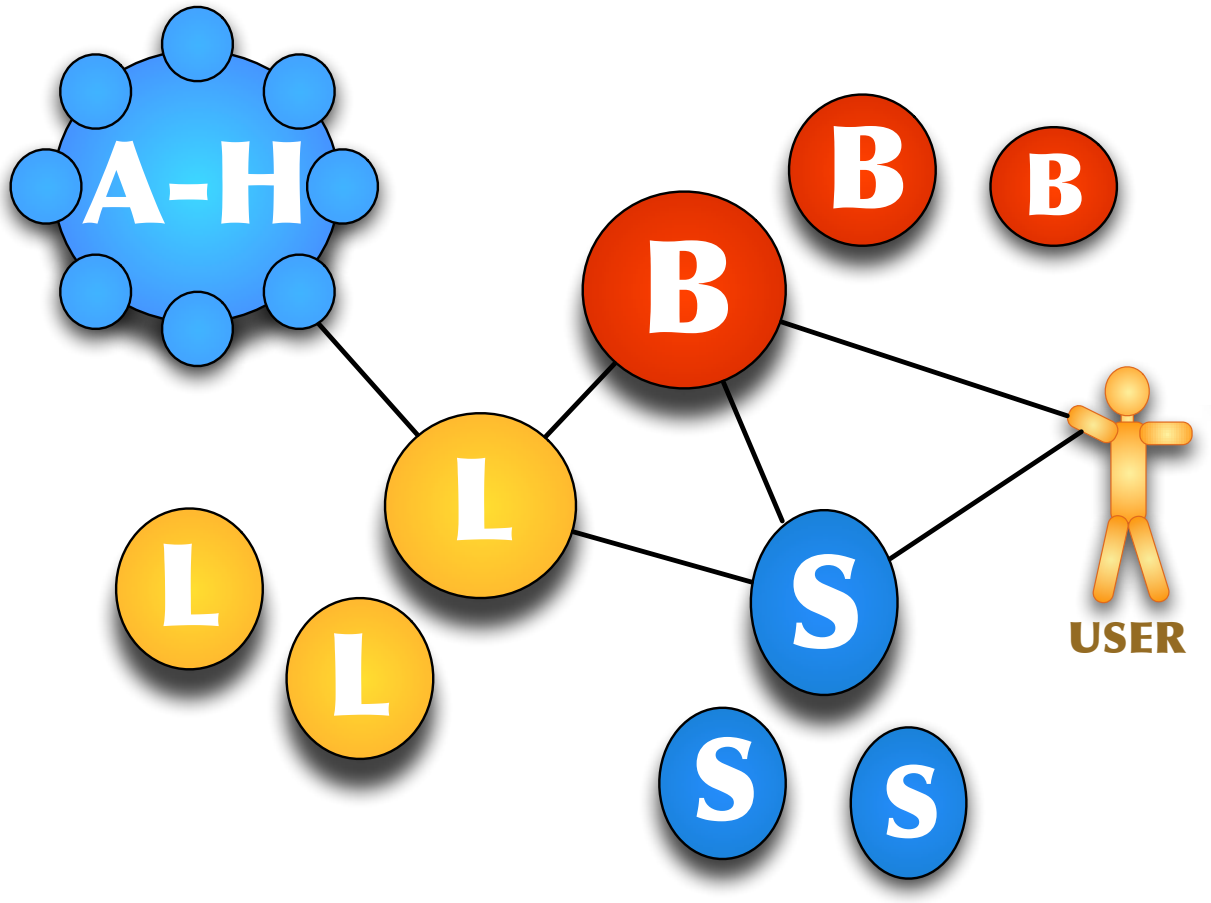

Figure 1.1: The components of the ARC storage: the **A-H**ash service, the **L**ibrarian service, the **B**artender service and the **S**hepherd service.

can refer to this file with the Logical Name `1234/green/orange/huge`, which means that from the collection called `1234` we 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 with no GUID prefix, it is implicitly prefixed with the GUID of this well-known 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`, which concept is very similar to a hardlink in a regular local file system.

1.2 Storage nodes and replicas

The collections in the ARC storage are logical entities, the content of a collection is stored as metadata of the collection, which means the a collection actually has no physical data. A file however has both metadata and real physicial data (the actual bytes 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

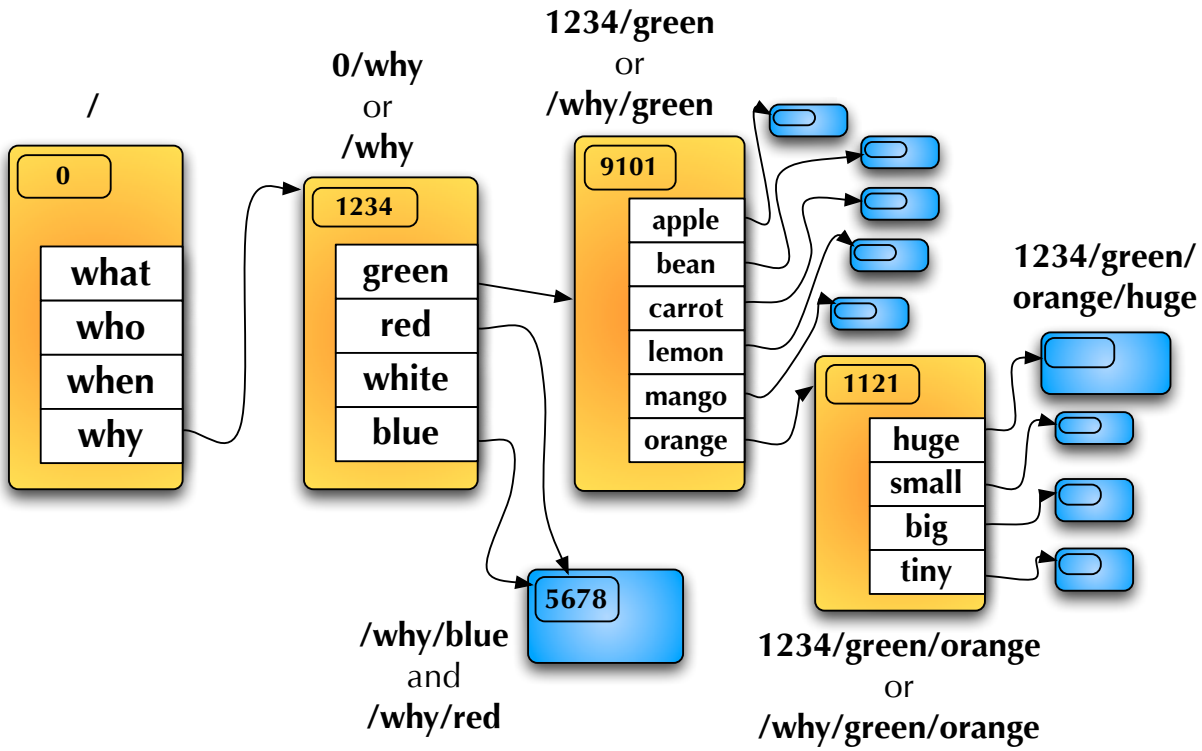

Figure 1.2: Example of the hierarchy of the global namespace

storage element services which made it possible the communicate with them.

So we have logical files, which are part of the hierarchical namespace and have a GUID and other metadata, and a logical file has one or more physical replicas. The physical replicas stored on seperate storage nodes. In order to connect the logical file to its replicas, we need to have some 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. So to connect the logical files to the physical ones, each logical file has a list of Locations.

The user can specify for each file how many replicas are needed. The ARC storage system periodically checks the number of replicas, and automatically creates new replicas if there are fewer then needed, and removes replicas if there are more.

The different replicas of a file could be in different states, e.g. the replica could be valid and alive or just in the process of creation or it could be corrupt or a whole storage node could be offline. This state is always stored as metadata next to the Location 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. And the storage nodes sends heartbeat messages periodically, and if a storage node goes offline, the missing heartbeat triggers the modification of metadata as well.

1.3 The A-Hash

The A-Hash is a replicated metadata store, which is capable of consistently storing 'objects' where an object contains property-value pairs organized in sections. All metadata about files and collections are stored in the A-Hash, and some other information (e.g. about A-Hash replication, about Shepherd services, etc.) is stored in it as well. The A-Hash itself does not interpret the data, it basically just stores tuples of strings.

1.4 The Librarians

The Librarian is capable of managing the hierarchy and metadata of files and collections, and health information of the Shepherd services. It can traverse Logical Names and return the corresponding metadata. It can receive heartbeat messages from Shepherd services, and it automatically modify the states of files if needed. The Librarian itself is a stateless service, it uses the A-Hash to actually store and retrieve the metadata, thats why there could be any number of independent Librarian services (all using the same A-Hashes) which provides high-availability and load-balancing.

1.5 The Bartenders

The Bartender service provides a high-level interface for the storage system to the clients. Every interaction between a client and the ARC storage system begins with a request to a Bartender. You can create and remove collections, create, get and remove files, move files and collections within the namespace using Logical Names. The Bartender authorizes users and force access policies of files and collections. It communicates with the Librarian and Shepherd services to accomplish the clients requests. The actual file data does not go through the Bartender; file transfers are directly performed between the storage nodes and the clients. There could be any number of independent Bartender services in the system which provides high-availability and load-balancing. The Bartender also provides a way to access files on third-party storage solutions through its interface by mounting the namespace of the third-party storage into the namespace of the ARC storage (this is accomplished by so called 'gateway' modules).

1.6 The Shepherds

The files in the ARC storage system are usually replicated on different storage nodes. For each storage node there is a Shepherd service which manages the storage element service on the node, reports its health state to a Librarian and provides the interface for initiating file transfers. For each kind of storage element service (e.g. a HTTP server, an FTP server, a storage solution with a GridFTP interface, etc.) it is needed to have a Shepherd backend which is capable of managing the given storage element service. The Shepherd service periodically checks the health of the replicas based on their checksums, and if a replica is deleted or corrupted, the Shepherd tries to recover it by downloading a valid copy from an other storage node. The Shepherds also check if a file has fewer replicas in the system than needed, and they initiate replication if needed.

1.7 Security

The ARC storage system consists of several services. Most of the services (A-Hash, Librarian, Shepherd) are 'internal' services in a way that the end-user of the storage system never communicates with them directly. But these internal services are communicating with eachother, so they have to know who to trust. We call this aspect of the security architecture of the ARC storage 'inter-service authorization'.

The end users always connect to one of the Bartender services, which will decide if the user has permissions to do something or not. This is the 'high-level authorization' part of the security architecture.

For transfering the actual file data, the users have to connect to storage element services which are sitting on storage nodes. These services also have their own authentication and authorization methods. Managing these aspects is the 'transfer-level authorization' part of the security architucture of the ARC storage.

1.7.1 Inter-service authorization

In a deployment of the ARC storage system, we could have several A-Hash, Librarian, Shepherd and Bartender services. The Bartenders send requests to the Librarians and the Shepherds, the Shepherds communciate with the Librarians, the Librarians talk with the A-Hashes. If any of these services get compromised or a new rouge service gets inserted in the system, we loose security completely. That's why it is vital for

each service to authorize the services before sending or accepting requests. The services commincates via HTTPS protocol, which means that they should provide an X.509 certificate for each connection, and they can examine the other service's certificates. Because of these X.509 certificates each service has Distinguish Name (DN). We can use these DNs to exactly specify which services we trust. We can configure a list of trusted DNs into each service, or we can store this list on a remote location. The services will only accept connections if the DN of the other end is listed in this list of trusted DNs. 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 the ARC storage system stores 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 these metadata. The users are identified by their DNs, and an access control rule specify the rights of the given user. One 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: we can have a rule for a whole VO (Virtual Organization) or for ALL users, like this:

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

Additionaly, 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 kept 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 a URL which is colled the transfer URL (TURL). This TURL is unique for each request, even for request to the same replica, and this TURL is only valid for one download. Currently we configure the storage element services to not do any authorization, and we use these one-time URLs to ensure that only the authorized users can access the contents of the storage elements.

Chapter 2

Use cases

2.1 Listing the contents of a collection

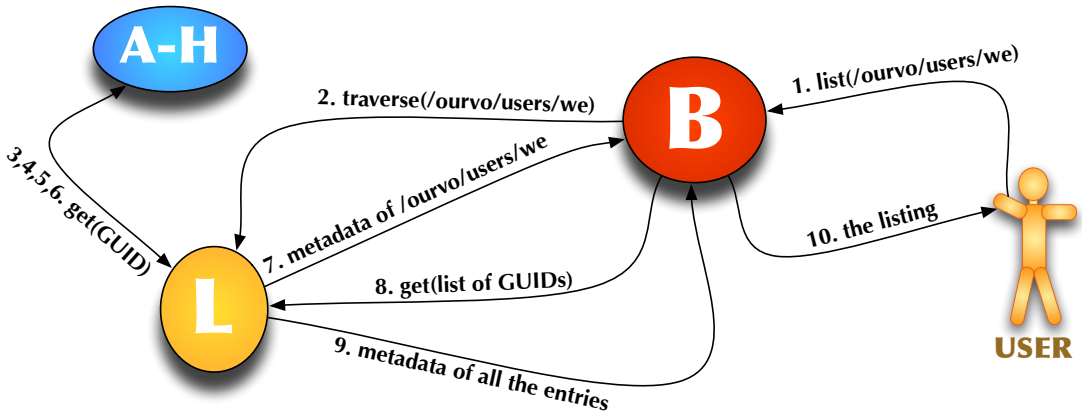

Figure 2.1: Listing the contents of a collection

We want to list the contents of a collection, which has a Logical Name of `/ourvo/users/we`. In order to do this, we have to contact a Bartender, and send a request to it containing the Logical Name we want to list, and the response from the Bartender will contain the list of entries. The steps are represented on Figure 2.1.

1. We need to know the URL of a Bartender. This could be preconfigured on the client side or in future releases it could be acquired from an information system. When we have the URL, we send a 'list' request which contains the Logical Name `/ourvo/users/we`.
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 GUIDs and not about LNs, but the GUID of the root collection is well-known, so the A-Hash can return the metadata of it which contains 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 new 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 Librarain 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 `we`, so the Librarian can ask for its metadata.

7. At last the A-Hash returns the metadata of `/ourvo/users/we` 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 if based on our DN (or our VO membership) and the access policies of this collection do we have rights to get the contents of this collection or not. If we are approved, then because the 'list' request should return additional metadata about each entry in the collection, the Bartender send a 'get' message to a Librarian requesting metadata of all the entries in this collection.

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

10. Now the Bartender has all the needed information, so it could return the proper response to us, the user. Our client tool formats and prints the results nicely.

2.2 Downloading a file

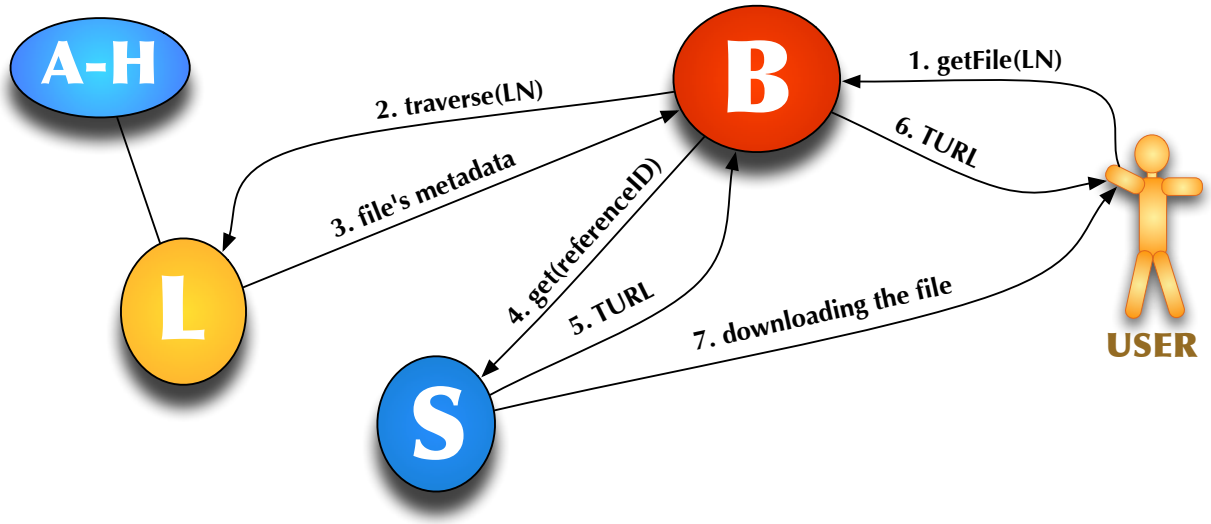

Figure 2.2: Downloading a file

In this use case we want to download a file which has a Logical Name of `/ourvo/users/we/thefilewewant` (see Figure 2.2).

1. We connect a Bartender and send a 'getFile' request with the LN of our file.

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

3. The Librarian do the step by step traversing and gets all the data 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 our identity if we are allowed to get the file, and if we are good to go, then it chooses a replica location. 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 URL for this replica. This URL called the transfer URL, and it will only be valid for one download. The Shepherd returns the TURL to the Bartender.

6. The Bartender returns the TURL to us.

7. Now we use this TURL to get the file directly from the storage element service on the storage node.

2.3 Creating a collection

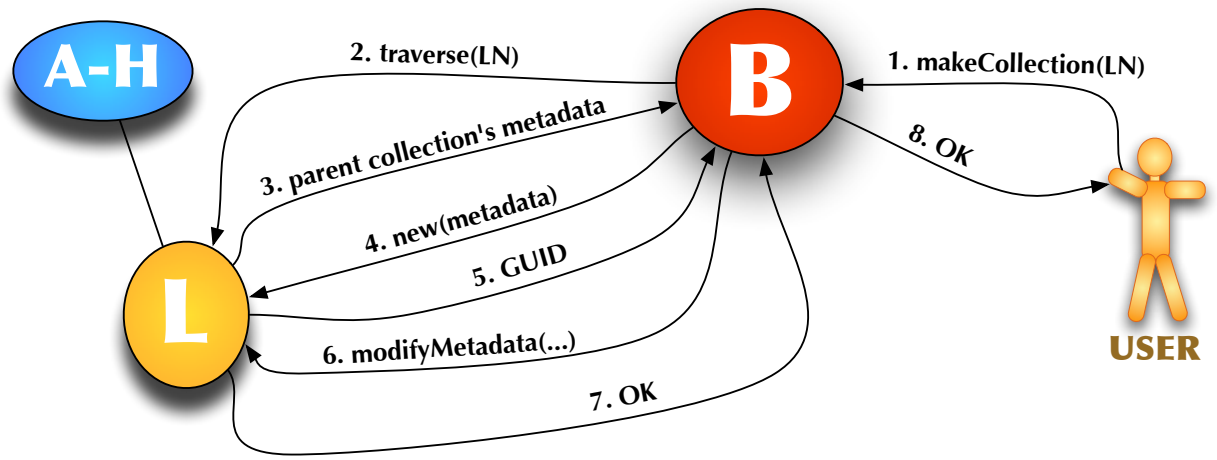

Figure 2.3: Creating a collection

We want to create a new (empty) collection as sub-collection of `/ourvo/common`, and we want to call it `docs` (see Figure 2.3)

1. We contact 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 try to traverse the Logical Name and it stops at the last possible point and returns the metadata of the last element. Because we want to put our collection to a new path but into an existing collection, we expect that only the `/ourvo/common` part of the LN can be traversed. If the Librarian could traverse the whole LN that would mean that there is already an existing file or collection by that name. If the parent collection does exist then the Librarian can traverse it which means that the parent collection's metadata is returned to the Bartender.
4. The Bartender checks the access policies to decide if we have 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 yet, 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 us if everything went OK or not.

2.4 Uploading a file

We have a file on our local disk we want to upload to a collection called `/ourvo/common/docs`. (See Figure 2.4.)

1. We contact a Bartender to put the file, we give the size and checksum and other metadata. And of course we give the Logical Name where we want to put the file, which in this case will be `/ourvo/common/docs/propos`
2. The Bartender ask a Librarian to traverse this LN.

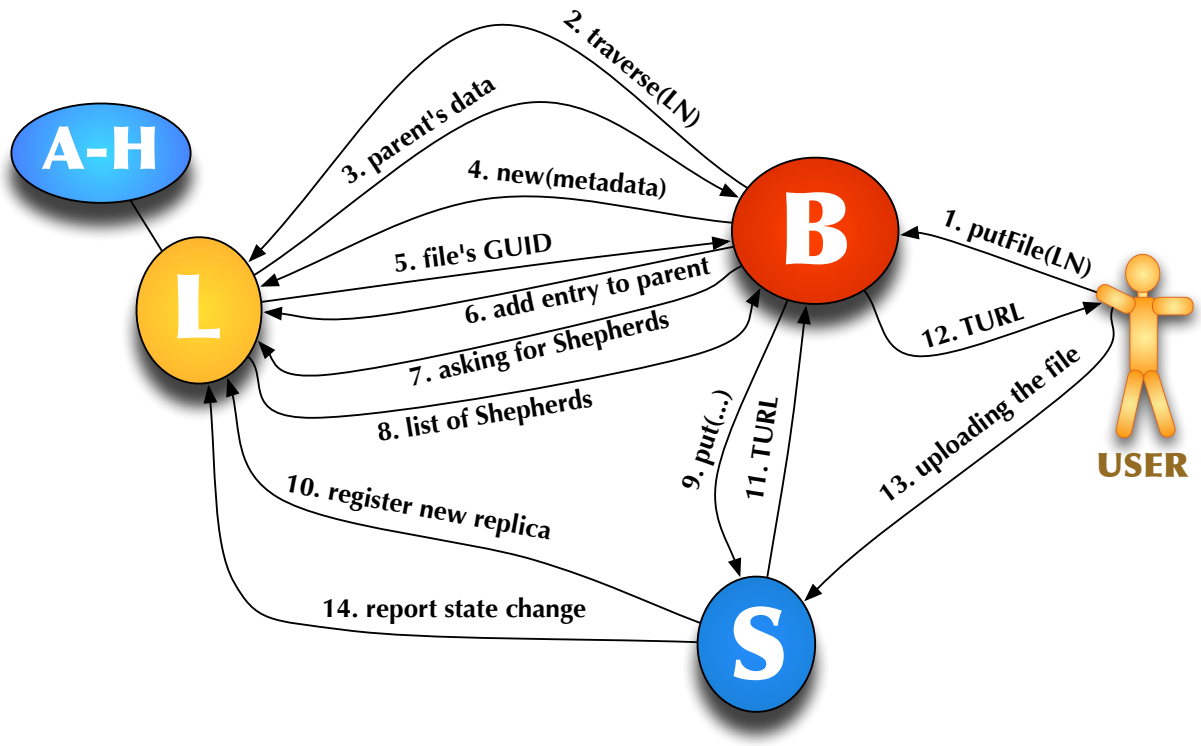

Figure 2.4: Uploading a file

3. The Librarian traverses the Logical Name, and we expect it to stop at the `/ourvo/common/docs` part of the LN, because that means that the name is available and the parent collection exists. If everything is fine, the metadata of the parent collection is returned to the Bartender.

4. The Bartender checks the access policies to decide if we have permissions to put something 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. Then the Bartender add 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, 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 chooses a Shepherd service and sends it a 'put' request to initiate the file upload.

10. The Shepherd communicates with the storage element service on the same node to create a new transfer URL (TURL). Then it creates a 'referenceID' for this file and then reports to the Librarian that there is a new replica in `creating` state. The Librarian gets the message from the Shepherd and adds the new replica to our new file. Now the file has one replica, which is not uploaded yet into the system. If someone tries to download this file now, still gets an error message.

11. The Shepherd returns the the TURL to the Bartender.

12. The Bartender returns the TURL to us.

13. Then we can upload the file to this TURL.

14. The Shepherd detects that the file is arrived. It 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 finally the file has one valid replica.

2.5 Removing a file

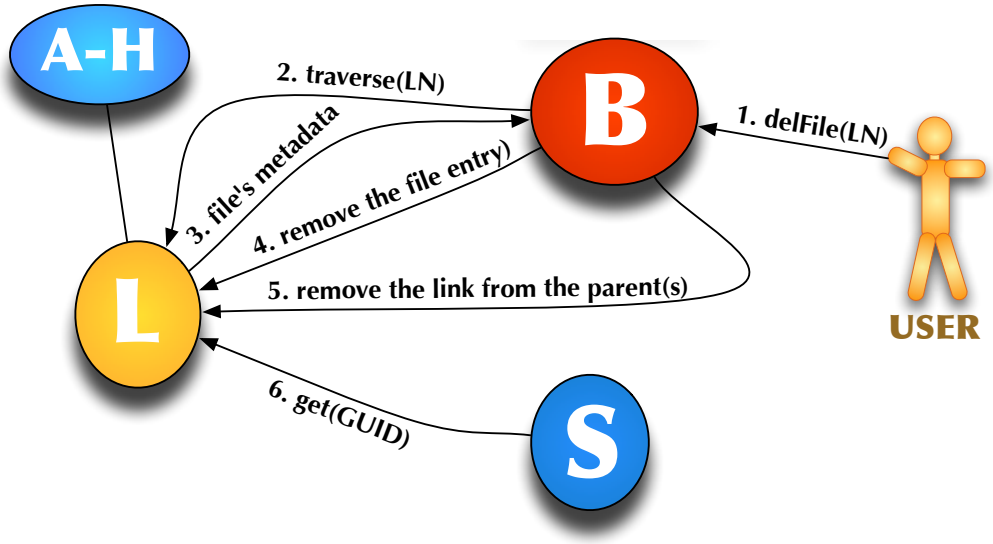

Figure 2.5: Removing a file

1. If we want to remove a file, we should connect to a Bartender with the LN of the file we want 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. 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 services are written in Python and running in the HED[1] hosting environment. The HED itself is written in C++, but there are language bindings which allow us to write services in other languages, e.g. in Python or Java. The source code of the storage services are in the NorduGrid Subversion repository[2].

Because the next-generation information system of ARC is currently under development, we cannot use it to discover services, that's why currently the URLs of almost all the services are hardcoded in the configuration files of the service. There is one exception: the Shepherd services are reporting their URLs to the Librarians, so a Bartender always could get a very current list of alive Shepherd services.

The HED also has a security framework which the Bartenders use the make access policy decision . The details of this is described in section X.

3.2 Data model

The ARC storage system stores different kinds of metadata about files, collections, mount point, Shepherd services, etc. Each of these has a unique ID which we call '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 too are simple character strings. The A-Hash services provide a simple interface to interact with these objects, e.g. to do conditinal and atomic changes.

The Librarian services use the A-Hash services to store the metadata of all kinds, using the GUIDs as unique IDs. For this, we have to represent the metadata 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'; and each entry could have 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 how ownes 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 identites (users, VOs, etc.).

parents section

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

metadata section

- any other arbitrary property-value pairs

3.2.1 Files

A file in the ARC storage has a logical entry in the namespace, and a couple of physical replicas on the storage nodes. We store the locations of the replicas in the *location* 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 could be '**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). We also store (in the *states* section) the size of the file, the number of needed replicas, and a checksum of the file, which is created by some kind of checksuming 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-hierarchy. Each entry has a unique name within a collection and a GUID which points to corresponding file or collection, so a collection is basically a list of name-GUID pairs, which list is stored in the *entries* section. The collections could 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 garanteed that the contents of a closed collection remains the same (e.g. if we include in our collection a file which we don't own, then the owner of the file could remove it nevertheless), but if a closed collection is changed, its state will be broken, and this state could 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 values could be '**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.

mountpoint section

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

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, storing 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 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 could 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 extract the identity information of the connecting client (DN of client, DN of the issuer CA, extended attributes, etc.),

and all these 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 either the request and the policy is converted to the native ARC policy language, and then the policy evaluator of the security framework is called to make the decision. Within the native ARC policy language XML representation of the requests and policies the storage actions are put into an 'Attribute' element with the ID '`http://www.nordugrid.org/schemas/policy-arc/types/storage/action`'.

For inter-service authorization we can configure the services to only accept connections from trusted services. We trust a service if we know its DN. For each service we can write trusted DNs in the configuration, or we can periodically get the list 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: we will trust all certificates issued by this CA

DNsFromAHash is a list of A-Hash URLs (in **AHashURL**) from where we periodically get a list of trusted DNs (there can be multiple A-Hash URLs listed here in case of one is offline). This has some attributes: **CheckingInterval** specifies how frequently we should get the list from the A-Hash (default: 600); **ID** specifies the ID of the A-Hash objects, where the list is stored (default: '3').

The TrustManager section itself could have some attributes: **FromFile** could have a filename, if we want to get the whole TrustManager configuration from a separate file; and **Force** indicates (with 'yes' or 'no') that we want to ensure that our 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>
```

Because of the heavy use of X509 certificates, of course, all the services should have their own certificates, which they can use when connecting to other services. Currently it is needed to put the paths of the certificate and private key files (and the trusted CAs as well) into the configuration of each services, 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 seperate 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 could 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. There could only be a single *value* for a *property* in a *section*.

If you have an ID, you can get all property-value pairs of the corresponding object with the *get* method, or you could specify only which sections or properties do you need. You can add or remove property-value pairs of an object or delete all occurrences of a property or create a new object with the *change* method, and you can specify conditions, which means 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 *value*s for all the *property* 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*), where *changeID* is an arbitrary ID which is used in the response to refer to this part of the request; *ID* points to the object we want to 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* could 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), '**unset**' (which will be true if the property of the section is not set at all). If all conditions are met, tries to apply changes to the objects, 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 could be **set**, **unset**, **deleted**, **failed**, **condition not met** (in this 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 you can decide if you want to deploy it 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 data on disk, e.g. serialized into 'pickle' format. And there is a module which stores the data replicated on multiple nodes.

TODO: here comes a fairly detailed description of how the replicated A-Hash works.

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 implementation, the `CentralAHash` is a centralized version, and the `ReplicatedAHash` is a replicated version.

StoreClass specifies the name of the store class. This class is used to store the data on disk. Currently there are several versions (`PickelStore`, `CachedPickleStore`, `StringStore`, `ZODBStore`, `TransDBStore`), however only the `TransDBStore` is suitable for using in a replicated deployemant, all the others are only for a centralized deployement.

StoreCfg could contain some 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 should save its files.

TODO: describe here the configuration needed for the replicated A-Hash.

An example 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>
```

3.5 Librarians

3.5.1 Functionality

The Librarian knows about the type of possible entries in the namespace of the ARC storage system: files, collections and mount points. It knows that these are organized in a tree-hiearchy with grouping them 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 the ARC storage, e.g. it does not put new files into collections, or 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 long 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 GUIDs, 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 knowa exactly that it stores metadata about files, collections, etc. With this interface you can create new files, collections and mount points with the *new* method, you can get or modify the metadata of any entry with the *get* and the *modifyMetadata* methods, and you can remove an entry with the *remove* method.

Besides being an interface to the metadata store, the Librarian has an other importat functionality: it provides a way for Shepherds to register themselves, and to periodically send reports about the state of their stored files. A Shepherd send 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 the 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 the ARC storage system 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 the requested GUIDs.

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

The response contains a list of *GUID, metadatList* 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 of the new entry.

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 GUIDs 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 we 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**' (set the property in the section to value), '**unset**' (remove the property-value pair from the section), '**add**' (set the property in the section to value only if it does not exist

already) or '**setifvalue**=value' (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*) 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* could be '**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*) with the Logical Names to be traversed

The response is a list of (*requestID*, *traversedList*, *wasComplete*, *traversedLN*, *GUID*, *metadataList*, *restLN*) 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 postfix 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, contains the ID of the Shepherd (currently the URL), 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 where GUID refers to the logical file of this replica. The *state* could be '**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) or '**alive**' (if the file is uploaded and the checksum is OK).

The response is *nextReportTime*, a number of seconds, which is the timeframe within the Librarian expects 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 GUIDs as IDs in the database. It has the URL of one or more A-Hash services from its configuration, and it can acquire more A-Hash URLs from these first A-Hashes in case of a replicated A-Hash deployement.

TODO: here comes some sentences about how the Librarian reads from any A-Hash and finds the master A-Hash for writing.

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 GUIDs 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. We can deploy multiple Librarians, if we configure them to use the same replicated group of A-Hashes (or the same central A-Hash) then the Bartenders could use any Librarian to achieve the same results, and the Shepherds also could 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 gets a report.

3.5.4 Configuration

The Librarian has these configuration variables:

AHashURL is the URL of an A-Hash which the Librarian should use. There could be multiple URLs specified here.

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 Shepherds

3.6.1 Functionality

A Shepherd service is capable of managing a storage node. It keeps track all the files it stores with their GUIDs and checksums. It periodically checks each file to detect corruption, and send reports to a Librarian indicating that the storage node is up and running, and whether some file's state has been changed. If a file goes missing or has a bad checksum then the Librarian is notified about the error (here the Shepherd refers to the file with its GUID, that's why it needs to store the GUIDs of its files). 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 we need a backend module which makes the Shepherd capable of communicating with it to initiate file transfers, to detect whether a transfer was successful or not, to generate local IDs and checksums, etc.

A file in a storage node could be identified with a *referenceID* which is unique within that node. If we know the *location* of a file, which is the ID of the Shepherd service (*serviceID*, currently the URL of the Shepherd) and the *referenceID*, we could call the Shepherds *get* method with the *referenceID* and a list of transfer protocols we can use, the Shepherd chooses a protocol from this list which it can provide, and create a transfer URL (*TURL*) and returns it along with the *checksum* of the file. We could download the file from this *TURL*, and verify it with the *checksum*. An end user of the storage system does not need to call this *get* method, because the Bartender service will do it, the user just asks the Bartender and gets the TURL.

Storing a file starts with initiating the transfer with the *put* method of the Shepherd, we should give the *size* and *checksum* of the file and its *GUID* as well. We also specify a list of transfer protocols we are able to use, and the Shepherd chooses a *protocol*, creates a *TURL* for uploading and generates a *referenceID*, then we can upload the file to the TURL. Again, the end user just asks the Bartender, and gets the TURL, the user does not need to call the *put* method of the Shepherd directly.

These *TURL*s are one-time URLs which means that after the client uploads or downloads the file these *TURL*s cannot be used again to do the same. If we want to download the same file twice, we have to initiate the transfer twice, and will get two different *TURL*s.

With the *stat* method we can get some information about a replica, e.g. checksum, GUID, state, etc. The *delete* method removes the replica.

In normal operation the *put* and *get* calls is made by a Bartender but the actual uploading and downloading is done by the user's client. 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, then the Bartender returns the TURL to the initiator Shepherd, which uploads its replica to the given TURL.

The replicas have a state, which could be '**creating**' when the transfer is initiated but the file is not uploaded yet, '**alive**' if the file is uploaded and has a proper checksum, or '**invalid**' if it does not exists anymore or has a bad checksum.

3.6.2 Interface

get(getRequestList) initiates a download and returns the TURL

> The *getRequestList* is a list of (*requestID*, *getRequestData*) 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 (there could be multiple protocols in getRequestData).
>
> The response is a list of (*requestID*, *getResponseData*), where *getResponseData* is a list of (*property*, *value*) pairs, such as: '**TURL**' is a transfer URL which can be used by the client to download the file; '**protocol**' is the protocol of the TURL; '**checksum**' is the checksum of the replica; '**checksumType**' is the name of the checksum method and '**error**' could contain an error message if there is one.

put(putRequestList) initiates an upload and returns the TURL

The *putRequestList* is a list of (*requestID*, *putRequestData*) where *requestID* is an ID used for the response, *putRequestData* is a list of (*property*, *value*) pairs such as '**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*), where *putResponseData* is a list of (*property*, *value*) pairs such as: '**TURL**' is the transfer URL where the client can upload the file, '**protocol**' is the chosen protocol of the TURL and '**referenceID**' is the generated ID for this new replica, '**error**' could contain an error message.

delete(deleteRequestList) removes a replica.

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

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

stat(statRequestList) returns information about replicas.

The *statRequestList* is a list of (*requestID*, *referenceID*) where *referenceID* points to the file whose data we want to get.

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

3.6.3 Implementation

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

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.

Let's see the scenario for the Hopi service which should be in a special 'slave' mode for this kind of operation: if a client asks for a file called `file1`, and this file is in the store directory (`./store/file`), then the Shepherd service creates a hardlink into the transfer directory (e.g. `./transfer/abc`) and sets this file read-only. If the Hopi service is configured that way that it handles the HTTP path `/prb` and it is serving files from the directory `./transfer` then after the hardlink is created, we have this URL for this file: `http://localhost:60000/prb/abc`. Now we can give this URL 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 this `http://localhost:60000/prb/abc` URL invalid (so this is a one-time URL), but because of the hardlink the file is still there 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` and now we have an URL `http://localhost:60000/prb/qwe`.

If a client wants to upload a new file, then the Shepherd creates an empty file in the store directory, e.g. `./store/file2` and creates a hardlink into the transfer directory, e.g. `./transfer/oiu` and makes it writable, and now we have a URL `http://localhost:60000/prb/oiu`, and the client is able to do a `PUT` to this URL. When the client `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 module for the Hopi service periodically checks whether a new file has two or just one hard links. If it has only one that means that a file is uploaded, so it could notify the Shepherd that the file is arrived. In order to do that, all the backend modules get a callback method 'file_arrived' from the Shepherd.

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

prepareToGet(referenceID, localID, protocol) returns the *TURL*.

Initiate transfer with *protocol* for the file which has these IDs: *localID* and *referenceID*. The reason for including here the referenceID as well is that this information could be used by the backend module later, e.g. when the transfer finished and the state of the file needs to be changed.

prepareToPut(referenceID, localID, protocol) returns the *TURL*.

Initiate transfer with *protocol* for the file which has these IDs: *localID* and *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 that 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 these 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 we can discover services by IDs.

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

MinCheckInterval specifies in seconds how much time should the 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 cheksum 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'.

StoreClass specifies which type of store the Shepherd should use for storing its metadata on disk.

StoreCfg can have parameters for the store class, e.g. the local directory to be used.

BackendClass specifes which backend the Shepherd should use for communicating with the storage element service.

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

LibrarianURL is the URL of a Librarian where the Shepherd should send the reports.

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

```
<Service name="pythonservice" id="shepherd">
    <ClassName>storage.shepherd.shepherd.ShepherdService</ClassName>
    <ServiceID>https://localhost:60000/Shepherd</ServiceID>
    <CheckPeriod>20</CheckPeriod>
    <MinCheckInterval>0.1</MinCheckInterval>
    <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 Bartenders

3.7.1 Functionality

The Bartender provides an easy to use interface of the ARC storage system to the users. You 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*. You can remove the link to a file or sub-collection 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 you can move (or create hardlinks of) collections and files within the namespace with *move*. You can upload an entirely new replica to a file (e.g. if the file lost all its replicas, or when a Shepherd service offers its replica for replications) with *addReplica*. You can create and remove mount points with the *makeMountpoint* and *unmakeMountpoint* methods.

The Bartender communicates with the 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 different kinds of thrid-party storage solutions. With these gateway modules it is possible to create mount points within the namespace of the ARC storage, and to access the namespace of the third-party storage through these mount points, which means the user can use the Bartender to get listing of directories on the third-party storage or to get files from the third-party storage. Thus the user can use a single client tool to access different kind of storages easily.

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 genereted 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 initiates the file transfer, and returns a TURL for each new file.

The *putFileRequestList* is a list of (*requestID*, *LN*, *metadata*, *protocols*), 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 protocols we can use for uploading, *metadata* is a list of (*section*, *property*, *value*) tuples which should contain some mandatory properties in the '**states**' section: '**size**', '**checksum**', '**checksumType**', and '**neededReplicas**'.

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

getFile(getFileRequestList) finds the files referenced by the given Logical Name, chooses a replica and initiates the download, and returns a TURL for each file.

The *getFileRequestList* is a list of (*requestID*, *LN*, *protocols*) where *requestID* is used in the response, *LN* is the Logical Name referring to the file we want to get, *protocols* is a list of transfer protocols we support.

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

delFile(delFileRequestList) removes the files references by the given Logical Names.

The *delFileRequestList* is a list of (*requestID*, *LN*) with the Logical Name of the files we want to delete.

The response is a list of (*requestID*, *status*), where the *status* could be '**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*) with the Logical Name of the files and sub-collections we want to unlink.

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

TODO: makeMountpoint, unmakeMountpoint

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

The *statRequestList* is a list of (*requestID*, *LN*) with the Logical Names we want to get information about.

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

makeCollection(makeCollectionRequestList) creates new collections.

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

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

unmakeCollection(unmakeCollectionRequestList) deletes empty collections.

The *unmakeCollectionRequestList* is a list of (*requestID*, *LN*) with the Logical Names of the collections we want to remove.

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

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

The *listRequestList* is a list of (*requestID*, *LN*) where *LN* is the Logical Name of the collection we want to list, *neededMetadata* is a list of (*section*, *property*) pairs which filters the returned metadata.

The response is a a list of (*requestID*, *entries*, *status*) 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* could be '**found**', '**not found**', '**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, does not move real file data.

The *moveRequestList* is a list of (*requestID*, *sourceLN*, *targetLN*, *preserveOriginal*) where *sourceLN* is the Logical Name referring to the file or collection we want to move (rename) and *targetLN* is the new path, and if *preserveOriginal* is true the *sourceLN* would not be removed, so with *preserveOriginal* we are actually creating a hard link.

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

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

The *modifyRequestList* is a list of (*changeID*, *LN*, *changeType*, *section*, *property*, *value*) where *changeType* can be '**set**' (set the *property* in the *section* to *value*), '**unset**' (remove the *property-value* pair from the *section*), '**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* could be '**set**', '**unset**', '**entry exists**' (for an 'add' request), '**denied**', '**no such LN**', '**failed**: reason'.

3.7.3 Implementation

For managing collections the Bartender needs only to communicate with a Librarian service, for managing 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. Fortunately the Librarian has a list of registered Shepherd services, so the Bartender can use that.

TODO: The Bartender can utilize gateway modules for accessing file listings and transfer URL on third-party storages. Curretly we have one implementation of a gateway module [...] A gateway module should provide these methods: [...]

TODO: a few sentences about the proxy delegation.

3.7.4 Configuration

The Bartender has these 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.

GatewayClass is the name of the gateway module we want to use in this Bartender.

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

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/
    </GatewayCfg>
    <ClientSSLConfig FromFile='clientsslconfig.xml'/>
</Service>
```

3.8 Client tools

3.8.1 Prototype CLI tool

In the first prototype release there is one 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 uploading and downloading TURLs via HTTP. However in order to use proxy certificates or to transfer files with GridFTP protocol, you need to have the ARC client libraries installed.

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.

Example for the `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>
```

Example for the 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
```

You can specify the number of needed replicas for new file with the `ARC_NEEDED_REPLICAS` environment variable.

The `arc_storage_cli` has its built-in help. The methods can be listed with:

```
$ 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, removeCredentials
```

Without arguments, each method prints its own help:

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

Here is an example of uploading, downloading and stat files:

```
$ 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

TODO: what is the FUSE module, how does it work, how can we use it

3.9 Grid integration

TODO: what is the ARC DMC, how does it work