NORDUGRID-TECH-17
24/2/2009

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/`

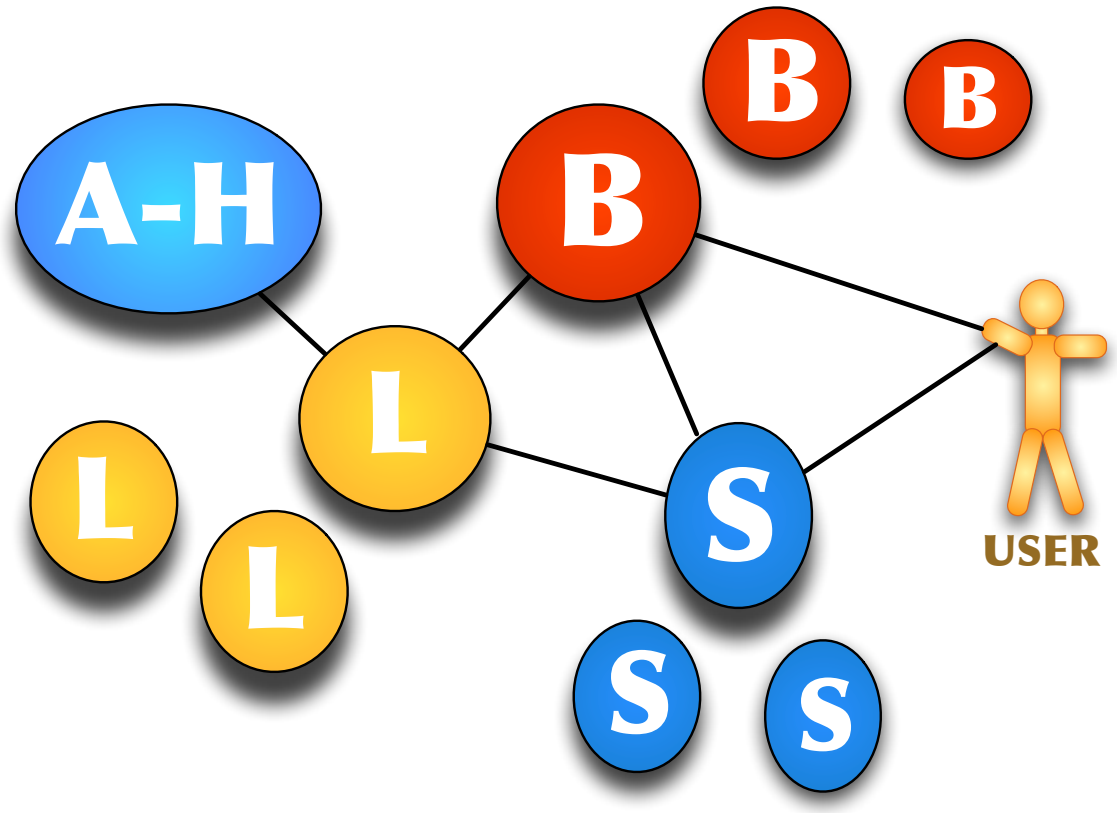

Figure 1.1: The components of the ARC storage: the **A-H**ash service, the **L**ibrarian service, the **B**artender service and the **S**hepherd service.

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 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

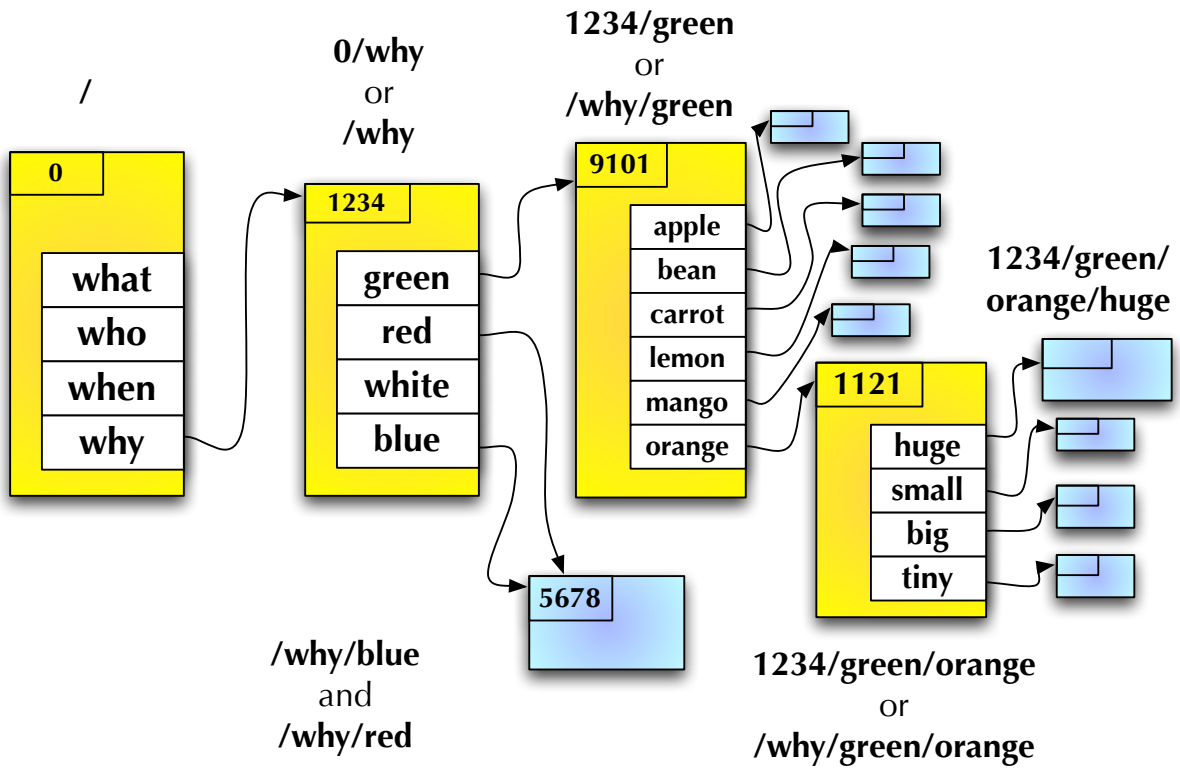

Figure 1.2: Example of the hierarchy of the global namespace

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 key-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 key-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 Downloading a file

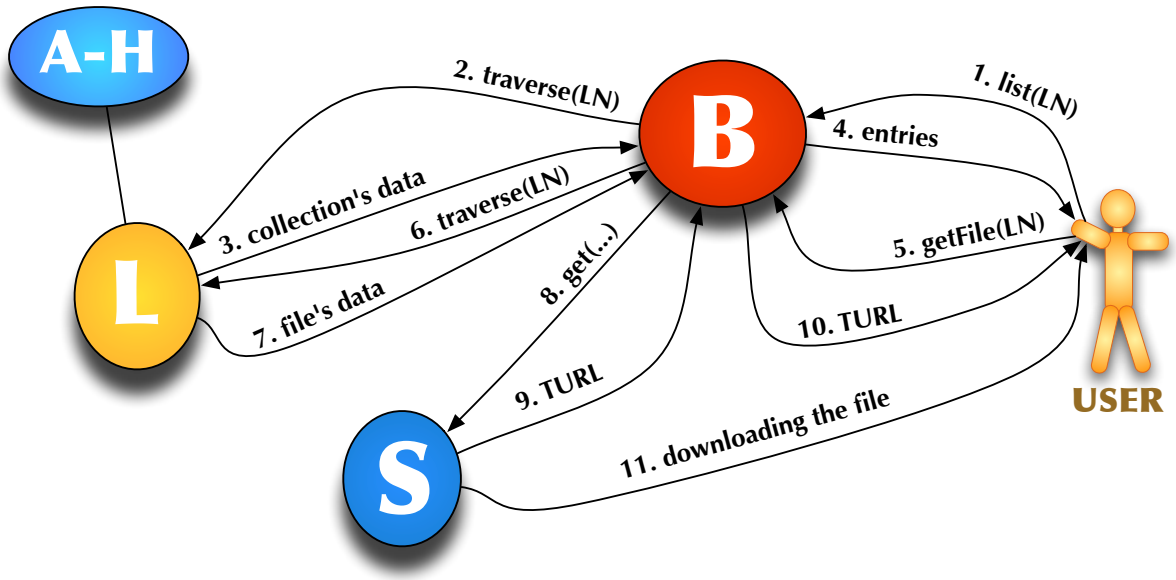

Figure 2.1: Downloading a file

We want to download a file about which we know that it is somewhere in our home collection on the storage (see Figure 2.1). The LN of our home collection is e.g. `/ourvo/users/we` (here a 'home collection' could be just a collection which is given to us by our VO). We can get a list of entries in this collection from any Bartender.

1. We need to find a Bartender. Maybe we have a cached list of recently used Bartenders or we can get one from the information system. When we have an endpoint reference of a Bartender, we could call its list method with the LN `/ourvo/users/we`.
2. The Bartender has to find a Librarian service, again using its cache of recently used Librarian services or get a new one from the information system. When the Bartender has an endpoint reference of a Librarian service, it could ask the Librarian to traverse the LN `/ourvo/users/we`.
3. The Librarian needs the A-Hash service to access the stored data, when it has the endpoint reference of the A-Hash service, it could get the information about the root collection, which contains the GUID of the `ourvo` sub-collection. Then the Librarian gets the entries of this `ourvo` collection, and in it it can find the GUID of `users`, and in the entries of `users` there is the GUID of `we`, which the Librarian returns to the Bartender with all the metadata.

4. The Bartender now has the GUID and the metadata of the collection `/ourvo/users/we`, including the list of its entries. This is returned to us.

5. So we get the list of our `/ourvo/users/we` collection, and now we realize that the file we want has the LN `/ourvo/users/we/thefilewewant` and we know the GUID of it as well: e.g. `a4b2e`. (Of course we know the GUID of the `/ourvo/users/we` collection too, which is e.g. `13245` and using this we could refer to our file as `13245/thefilewewant` which means the entry called `thefilewewant` in the collection with a GUID `13245`.) We connect a Bartender again (the same one or maybe another one) to get the file with any of these LNs, the `a4b2e` is the fastest solution because the Bartender need not to look up the whole LN again in the Librarian, a well-written client API should use this. With the get request we give the Bartender the list of transfer protocols we are able to use.

6. The Bartender contacts the Librarian to get the locations of the replicas of this file.

7. The Librarian returns all the metadata of the requested LN.

8. The Bartender chooses a 'location' which consists of the ID of a Shepherd, and the referenceID of the file within the Shepherd. Using the information system or its local cache it could get the endpoint reference of the Shepherd. The Bartender initiates a transfer by the Shepherd, if the Shepherd supports one of the transfer protocols we give, it can create a transfer URL (TURL) with a protocol we can download.

9. The Shepherd returns the TURL to the Bartender.

10. The Bartender returns the TURL to us.

11. Now we have a TURL from which we can download it.

2.2 Uploading a file

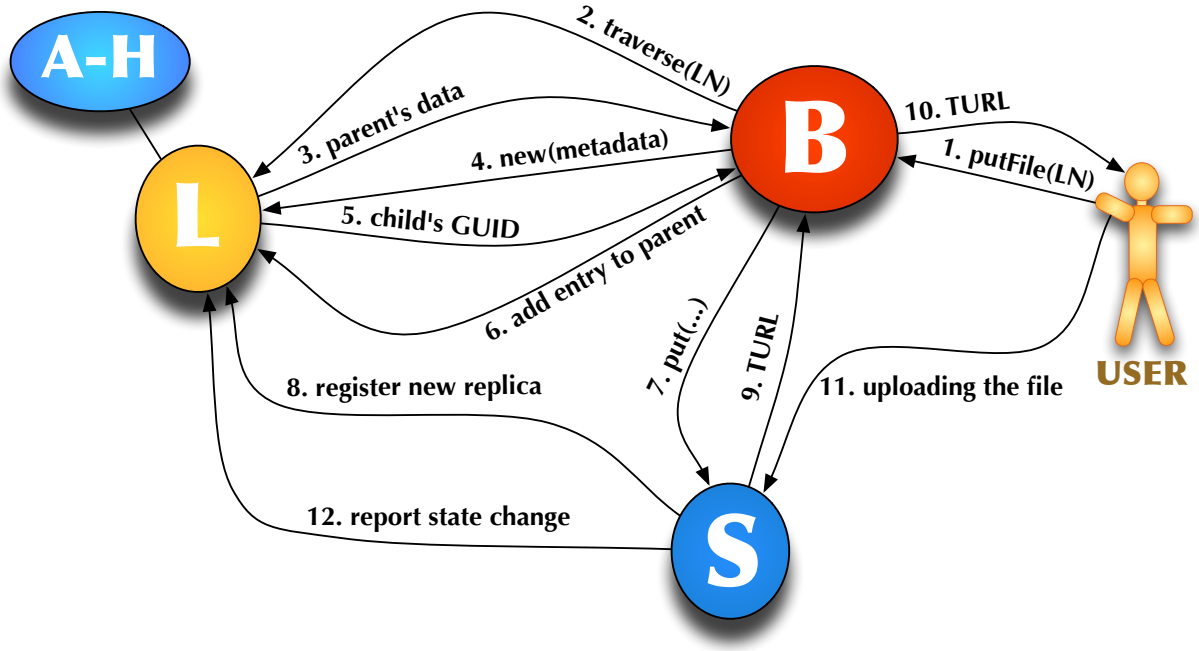

Figure 2.2: Uploading a file

We have a file on our local disk we want to upload to a collection called `/ourvo/common/docs`. (See Figure 2.2.)

1. We contact a Bartender to put the file, we give the size and checksum and other metadata, and the transfer protocols we want to use. And of course we give the Logical Name we want to be the name of the file, which in this case will be `/ourvo/common/docs/proposal.pdf`

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

3. If the Librarian can traverse the whole LN then this LN is already exists, but if the LN is still available and the parent exists, the Librarians response contains the GUID and metadata of the parent collection.

4. Then the Bartender creates a new file entry within the Librarian with all the information we gave.

5. The Librarian returns the GUID of this new entry.

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` which points to a file which has no replica at all. If someone tried to download the file called `/ourvo/common/docs/proposal.pdf` now, would get an error message 'try again later'.

7. The Bartender (using the information system) chooses a Shepherd and gets its endpoint reference. Then the Bartender initiates uploading of the file to the Shepherd: the request includes the size and checksum of the file, the GUID, and the protocols we are able to use.

8. The Shepherd creates a transfer URL and a referenceID for this file and registers the GUID of the file in its own database and reports to the Librarian that there is a new replica with state `creating`. The Librarian gets the message from the Shepherd and creates a new entry in the locations list of the given file with the serviceID and the referenceID the Shepherd have just reported. If someone tries to download this file now, still gets a 'try again later' error message, because this new replica is still not `alive`.

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

10. The Bartender returns the TURL to us.

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

12. The Shepherd detects that the file is arrived and reports the change of state to `alive` to the Librarian who alters the state in the given file-entry. At this point the file has only one replica.

- The Shepherd periodically checks the Librarian if this is less than the needed replica number, and if it is then it initiates creating a new replica by a Bartender.

- The Bartender chooses another Shepherd, initiates the transfer then returns the TURL to the first Shepherd which uploads the file to the new Shepherd.

- All the Shepherds check periodically whether their files have enough replica, and if any of them find that there is more replica needed, it initiates creating a new. If more than one Shepherd of course could cause that there will be more replicas than needed. If a Shepherd finds out that a file has more replicas than needed it notifies a Bartender about it.

- The Bartender ask the Librarian about all Shepherds this file has replicas on, flags this file as 'removing a replica' which prevents other Bartenders to remove an other replica accidentally, then make a decision of which one is to be removed, then contacts the chosen Shepherd and asks it to remove the replica. The Shepherd then notify the Librarian, and the Librarian removes the replica, and removes the flag 'removing a replica' as well.

- If the client cannot upload the file to the given TURL for some reason, it is possible to call `addReplica` to get a new TURL without removing and recreating the file.

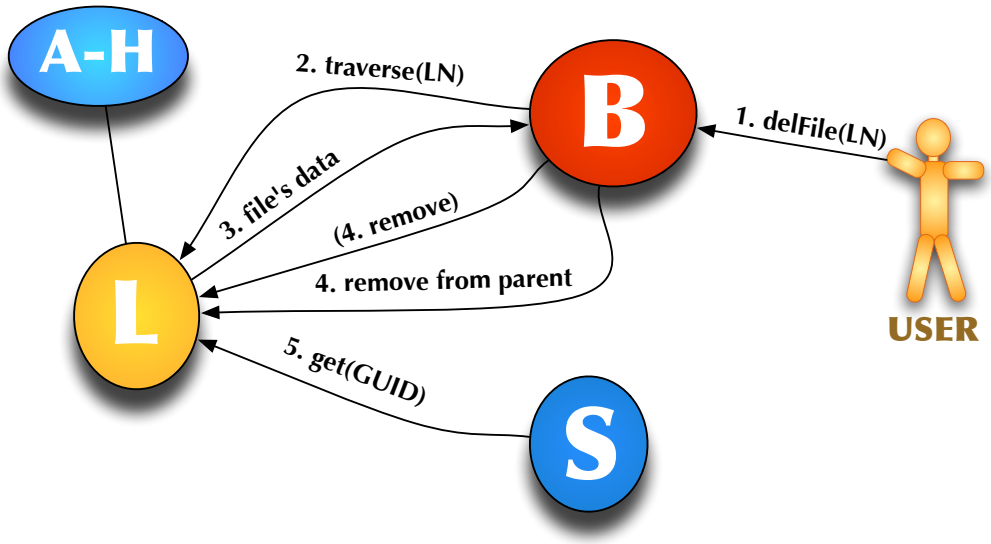

Figure 2.3: Removing a file

2.3 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 and return the list of parent collections of this file.
3. The Librarian returns the data.
4. If the file has only parent, it asks the Librarian to remove the file entry itself, and the entry from the parent collection. If the file has more parent collections (hard links) the Bartender only removes the entry from the parent collection and removes the parent from the list of parent collections of this file.
5. Next time the Shepherd does its periodic check, it asks the Librarian about each of its stored replicas, and finds out that one of them no longer exists, so it removes the replica from the storage node.

Chapter 3

Technical description and implementation status

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].

The current version of the prototype has no information system and no security, these are soon to be integrated to the system.

The information system is needed to discover services, and to translate *serviceID*s to endpoint references (URLs). Currently the URLs are written in the configuration files, and the Shepherd services are reporting their URLs to the Librarian, so the Bartender could ask for all alive Shepherds.

Some self-healing mechanisms are not implemented yet, e.g. the files and collections contains their parent-collections as their metadata, and of course a collection contains a list of its files and sub-collections, and if somehow this information became inconsistent, the system should detect and repair it. This is not implemented yet.

3.1 Plans for security

Security is needed to do proper authorization of the users, and to manage access policies of files and collections. ARC has its own policy language, for each file and collection there will be a policy XML document stored as a metadata. The storage services will use these policies and the properties extracted from the communication channel to make authorization decisions. If the properties and the policies are present, the decision will be actually made by the security framework of HED.

These are the planned 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 (if you want to remove a file/collection, then the Bartender needs to remove the entry from the parent collection, and then delete the file/collection itself, so you need to have both permissions)
- *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)

[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`

- *modifyMetadata*: user can modify the arbitrary metadata section of the file/collection (these are key-value pairs)

When a user has the permission in the Librarian to download a file then the user should have permission to access at least one of the file's replica, so there should be a Shepherd which allows the user the get the file. If the Bartender has permission to access the Shepherd, then the Bartender should create an assertion which allows the user to access the file. This could be a signed token which contain a policy defining access to particular file. But all these are currently just plans.

Further prototype statuses and plans can be found below within each section about the services.

3.2 A-Hash

3.2.1 Functionality

The A-Hash will be a distributed service capable of storing tuples of strings in a scalable manner. Currently it only has a centralized implementation. It 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.

3.2.2 Prototype status and plans

The A-Hash service currently implemented as a single central service, which stores the data on disk in separate files per *object*. Fall of 2008 it will be reimplemented using a distributed hash table (DHT) algorithm, one possible candidate is the Chord[3] algorithm with a consistency solution called Etna[4] on top of it. This reimplementation hopefully wont change the interface of the service.

3.2.3 Data model

- *ID* is an arbitrary string
- *object* contains property-value pairs in sections, technically it is a list of key-*value* pairs where the key is a (*section*, *property*) tuple

3.2.4 Interface

get(ids, neededMetadata) returns *getResponse* which is a list of (*ID*, *object*) pairs.

The *ids* is a list of string *ID*s, *neededMetadata* is a list of (*section*, *property*) pairs. For each *ID* it returns all the *value*s for each *property* in each *section* (filtered by *neededMetadata*), so *object* is a list of (*section*, *property*, *value*) tuples.

change(changeRequest) returns *changeResponse* which is a list of (*changeID*, *success*, *failedConditionID*) tuples.

changeRequest is a tuple of (*changeID*, *ID*, *changeType*, *section*, *property*, *value*, *conditions*), where *changeID* is an arbitrary ID to identify in the response which change was successful; *ID* points to the object we want to change; *changeType* can be '**set**' (set the property within the section to value), '**unset**' (remove the property from the section regardless of the value), '**delete**' (removes the whole object), conditions is a list of (*conditionID*, *type*, *section*, *property*, *value*) tuples, where *type* could be '**is**' (the property in the section is set to the value), '**isnot**' (the property in the section is not set to the value), '**isset**' (the property of the section is set to any value), '**unset**' (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. If one of the conditions is not met, returns the ID of the failed condition.

[3] The Chord project - `http://pdos.csail.mit.edu/chord/`

[4] Etna: a Fault-tolerant Algorithm for Atomic Mutable DHT Data - `http://pdos.csail.mit.edu/~athicha/papers/etna.ps`

3.3 Librarians

3.3.1 Functionality

The Librarian manages a tree-hierarchy of files, grouping them into collections. There is a root collection with a well-known GUID which can be used as starting point when resolving Logical Names. If you create a new collection with the method *new*, the Librarian generates a new GUID, but does not insert it into the tree-hierarchy which can be done by adding this GUID as a new entry to one of the existing collection using the *modifyMetadata* method of the existing collection which makes it the parent of the new collection. A collection can be closed via metadata modification which cannot be undone and prevents files to be added or removed from this collection. A new file also can be created with the *new* method which returns the newly generated GUID of the new file entry which should be added to a parent collection to insert it into the global namespace. A file has a list of locations where its replicas are stored, this list too can be manipulated with *modifyMetadata*. The access policies of the files and collections are also stored as metadata. The *remove* method deletes an entry from the Librarian. The *traverseLN* method try to traverse Logical Names by walking the hierarchy of the namespace and to return the GUID of the entry pointed by the LN. After you have a GUID of file, collection or mount point, you can get all the information using the *get* method.

3.3.2 Prototype status and plans

The Librarian service currently implements all the methods below, but doesn't do very much error checking. This should be changed, the Librarian should check the validity of metadata, and forbid some cases, e.g. reopen a closed collection.

3.3.3 Data model

Each librarian entry has a unique ID called *GUID*.

The Librarian uses the A-Hash to store all the data about files and collections. The A-Hash is capable of storing property-value pairs organized in sections, which actually means that it stores (*section*, *property*, *value*) tuples where each member is simply a string, e.g. ('entry', 'type', 'collection') or ('ACL', 'johnsmith', 'owner') or ('timestamps', 'created', '1196265901') or ('locations', '64CDF45F-DDFA-4C1D-8D08-BCF7810CB2AB:9A293F27DC86', 'sentenced'). There could be only one *value* for a (*section*, *property*) pair.

- A **collection** is a list of files and other collections, which are in parent-children relationships forming a tree-hierarchy. Each entry has a name which is only valid within this collection, and it is unique within the collection. Each entry is referenced by its GUID. So the metadata sections of a collection are as follows:

  entry section
  - *type*: 'collection'

  entries section
  - *(name, GUID) pairs*: a Collection is basically a list of name-GUID pairs.

  timestamps section
  - *created*: timestamp of creation
  - *modified*: timestamp of last modification

  states section
  - *closed*: if the collection is closed, then nothing can be added to its contents

  policies section
  - XML representations of access policies

  metadata section
  - any other arbitrary metadata

- A **file** entry contains the following sections:

  entry section
  - *type*: 'file'

  locations section
  - *(location, state) pairs*, where a location is a (*serviceID*, *referenceID*) pair serialized as a string, where *serviceID* is the ID of the Shepherd service storing this replica, *referenceID* is the ID of the file within that Shepherd service, and 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), '**sentenced**' (if the replica is marked for deletion)

  timestamps section
  - *created*: timestamp of creation
  - *modified*: timestamp of last modification (e.g. modification of metadata)

  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

  policies section
  - XML representations of access policies

  metadata section
  - any other arbitrary metadata

- There is one more type of Librarian entries called **mount point** which is a reference to a service which is capable of handling a subtree of the namespace. The properties of a mount point in sections:

  entry section
  - *type*: 'mountpoint'

  mount section
  - *target*: the ID of the service
  - *ID*: an ID within the service (optional)

  timestamps section
  - *created*: timestamp of creation
  - *modified*: timestamp of last modification (e.g. modification of metadata)

  policies section
  - XML representations of access policies

  metadata section
  - any other arbitrary metadata

- The Librarian stores information about the Shepherds, so each Shepherd has a GUID as well. There is an entry (with GUID '1' by default) which contains the GUID and the timestamp of the last heartbeat for each registered Shepherd:

  nextHeartBeat section
  - (ID, timestamp) pairs

  serviceGUID section
  - (ID, GUID) pairs

- For each Shepherd there is a separate entry with the list of files:

  entry section
  - *type*: 'shepherd'

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

3.3.4 Interface

new(newRequestList) returns a list of (*requestID*, *GUID*, *success*)

newRequestList is a list of (*requestID*, *metadata*) where *requestID* is an arbitrary ID used to identify this request in the list of responses; *metadata* 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. The ('entry', 'type') property of the metadata contains whether it is a file or a collection.

modifyMetadata(modifyMetadataRequestList) returns a list of (*changeID*, *success*)

modifyMetadataRequestList is a list of (*changeID*, *GUID*, *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).

get(GUIDs, neededMetadata) returns *getResponse*

GUIDs is a list of GUIDs, *neededMetadata* is a list of (*section*, *property*) pairs indicating only which properties we need, *getResponse* is a list of (*GUID*, *metadata*) where metadata is a list of (*section*, *property*, *value*) tuples. This method returns the metadata of all the *GUID*s filtered with *neededMetadata*.

remove(removeRequestList) returns a list of (*requestID*, *success*) pairs

removeRequestList is a list of (*requestID*, *GUID*) pairs. *success* could be '**removed**' or '**failed**: reason'.

traverseLN(traverseRequestList) returns *traverseResponseList*

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

traverseResponseList is a list of (*requestID*, *metadata*, *GUID*, *traversedLN*, *restLN*, *wasComplete*, *traversedList*) where:

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

GUID is the *GUID* of the *traversedLN*

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

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

wasComplete indicates whether the full *LN* was traversed

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 *LN* without its last part

report(serviceID, filelist) returns in *nextReportTime* a number of seconds, which is the timeframe within the Librarian expects the next heartbeat from the Shepherd

filelist is a list of (*GUID*, *referenceID*, *state*) tuples containing the state of changed or new files, where *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).

3.4 Shepherds

3.4.1 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 file transfer service (e.g. 'HTTP(S)', 'FTP(S)', 'ByteIO', 'GridFTP', etc.). For each supported file transfer service we need a backend module which makes the Shepherd capable of communicating with the file transfer service 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*) and the *referenceID*, we could get the endpoint reference (URL) of the Shepherd from the information system, then we could call its *get* method with the *referenceID* and a list of transfer protocols we can use (e.g. 'HTTP', 'FTP'), 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 ('creating', 'alive' or 'invalid'), etc. The *delete* method removes the file.

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.

3.4.2 Prototype status and plans

The current implementation of the Shepherd service has a working *get*, *put*, *stat*, *delete* methods, and a method called *toggleReport* which can be used the simulate storage node failure with the Shepherd not reporting to a Librarian. There is a separate service which provide a subset of the ByteIO interface, and there is an other separate service which is a basic HTTP server, these are both could be used as file transfer services, both have its backend module for the Shepherd. Currently both file transfer services have the problem of using too much memory while transferring files. Further plans include better file transfer services and backend modules for third-party file transfer services.

3.4.3 Data model

A file of a Shepherd service is referenced by its *referenceID*. Each file has 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. Each file has a *localID* which is used in the backend modules.

3.4.4 Interface

get(getRequestList) returns list of (*requestID*, *getResponseData*)

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).

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) returns a list of (*requestID*, *putResponseData*)

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**' (for additional access policy).

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) returns a list of (*requestID*, *status*)

deleteRequestList is a list of (*requestID*, *referenceID*) pairs selecting the files to remove. The status could be '**deleted**' or '**nosuchfile**' .

stat(statRequestList) returns a list of (*requestID*, *referenceID*, *state*, *checksumType*, *checksum*, *acl*, *size*, *GUID*, *localID*)

statRequestList is a list of (*requestID*, *referenceID*) where *referenceID* points to the file whose data we want to get. The method returns all the data the Shepherd know about the replica.

3.4.5 Backend modules

The Shepherd could communicate with the file transfer services via backend modules. Currently there are two kinds of backend modules, one for the *byteio* service (which is a simple implementation of a subset of the ByteIO interface) and one for the *Hopi* service (which is a simple HED-based HTTP server).

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.

In case of the byteio service, there is some small differences. The byteio service does not removes the files from the transfer directory, but it calls the backend module via SOAP to notify it that something is happened. The byteio backend has one SOAP method called '**notify**':

notify(subject, state) returns '**OK**' in *notifyResponse*.

When this method is called, the byteio backend module notifies the Shepherd that the file is arrived.

All the backend modules should have this common interface which the Shepherd can use to communicate with the file transfer service:

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) returns the checksum of the file referenced by *localID*, which checksum is generated by the method *checksumType*.

3.5 Bartenders

3.5.1 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*. 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 hardlink) 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*.

3.5.2 Prototype status and plans

The methods mentioned in the above section are all implemented, but need more error-checking and metadata-checking. There are plans of adding new methods, e.g. a *copy* method or a *glob* method for file pattern matching. The current version does not support closed (unmodifiable) collections yet.

3.5.3 Data model

The Bartender interface uses mostly Logical Names (*LN*s), which have the syntax of: `<GUID>/<path>` where both sides can be omitted (and in the case of a sole GUID we don't need the slash either), e.g. `afg342/foo` is an entry called `foo` in the collection with GUID `afg342`; the LN `f36a7481` refers to the a file or collection with GUID `f36a7481`; `/vo/dir/stg` points to the entry which is reachable from the root collection using the given path; and `/` simply refers to the root collection. The term '*metadata*' here refers to a list of property-value pairs organized in sections, see the data model description in Section 3.3.3.

3.5.4 Interface

putFile(putFileRequestList) returns a list of (*requestID*, *success*, *TURL*, *protocol*)

> *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 chosen 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 where properties could be in the '**states**' section: '**size**', '**checksum**', '**checksumType**', and '**neededReplicas**', policy documents in the '**policies**' section and any other property-value pairs in the '**metadata**' section. The returned *TURL* is a URL with a chosen *protocol* to upload the file itself, the *success* string could be '**done**', '**missing metadata**', '**parent does not exists**', '**internal error:** reason', etc.

getFile(getFileRequestList) returns a list of (*requestID*, *success*, *TURL*, *protocol*)

> *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 the client supports. In the response *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) returns a list of (*requestID*, *status*)

> *delFileRequestList* is a list of (*requestID*, *LN*) with the Logical Name of the file we want to delete. The status in response could be '**deleted**' or '**nosuchLN**'.

stat(statRequestList) returns a list of (*requestID*, *metadata*)

> *statRequestList* is a list of (*requestID*, *LN*) with the Logical Name of the file or collection we want to get information about, and it returns *metadata* which is a list of (*section*, *property*, *value*) tuples according to the data model of the Librarian (see Section 3.3.3)

makeCollection(makeCollectionRequestList) returns a list of (*requestID*, *success*)

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), in the '**states**' section there is the '**closed**' property (if it is true then no more files can be added or removed later), in the '**policies**' section there could be some access policies, and in the '**metadata**' section there could be any other metadata in key-value pairs. The *success* in the response could 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) returns a list of (*requestID*, *success*).

unmakeCollectionRequestList is a list of (*requestID*, *LN*) with the Logical Names of the collections we want to remove. *success* could be '**removed**', '**no such LN**', '**collection is not empty**', '**failed**: reason'.

list(listRequestList, neededMetadata) returns *listResponse*.

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

listResponse is 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 according to the data model of the Librarian (Section 3.3.3), the *status* could be '**found**', '**not found**', '**is a file**' (because only collections can be listed).

move(moveRequestList) returns a list of (*requestID*, *status*).

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 (or just rename) and *targetLN* is the new path, and if *preserveOriginal* is true the *sourceLN* would not be removed, so with *preserveOriginal* we actually creating a hard link. The status could be '**moved**', '**nosuchLN**', '**targetexists**', '**invalidtarget**', '**failed adding child to parent**', '**failed removing child from parent**'

modify(modifyRequestList) returns a list of (*changeID*, *success*)

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). *success* could be '**no such LN**', '**set**', '**unset**', '**entry exists**', '**failed**: reason'.

3.6 Client tools

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.

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]
```

Without arguments, each method prints its own help:

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

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
- ARC_BARTENDER_URL environment variable not found, using http://localhost:60000/Bartender
- 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
- ARC_BARTENDER_URL environment variable not found, using http://localhost:60000/Bartender
- 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
- ARC_BARTENDER_URL environment variable not found, using http://localhost:60000/Bartender
- 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.
```

You can find more examples in the SVN[5].

There are plans to create more sophisticated CLI and GUI tools and to create a FUSE[6] module and Windows Shell Extensions[7] to be able to mount the ARC storage namespace into the local filesystem namespace, and use it with the commands of the operating system.

3.7 Integrating third-party storage solutions

To integrate existing files on a third-party storage system to our namespace thus make them accessible through the interface of the ARC storage, we need a service which provides a common interface to the Bartender services, and hides the details of accessing the different third-party storages. It should translate the method calls, the *get*s, *put*s and *remove*s and the *ACL*[8] modifications, and try to create transform metadata according to the data model of the ARC storage (Section 3.3.3). The files here would be referenced by a path which is local in the namespace of the third-party storage. The interface could be something like this:

get(getRequest) returns list of (*requestID*, *getResponseData*)

very similar to the *get* method of the Shepherd

put(putRequest) returns a list of (*requestID*, *putResponseData*)

very similar to the *put* method of the Shepherd

delete(deleteRequest) returns a list of (*requestID*, *success*)

very similar to the *delete* method of the Shepherd

stat(statRequest) returns a list of (*requestID*, *statResponse*)

very similar to the *stat* method of the Shepherd

list(listRequest) returns *listResponse*

very similar to the *list* method of the Bartender

move(moveRequest) returns a list of (*requestID*, *status*)

very similar to the *move* method of the Bartender

This interface has some methods similar to the Shepherd and some other methods similar to the Bartender.

[5] `http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/services/storage/README`
[6] Filesystem in Userspace, `http://fuse.sourceforge.net/`
[7] `http://msdn.microsoft.com/en-us/magazine/cc188741.aspx`
[8] Access Control Lists, policies