Project no. 032691

KnowARC

Grid-enabled Know-how Sharing Technology Based on ARC Services and Open Standards

Specific Targeted Research Project

Information Society Technologies

DESIGN OF ARC STORAGE SYSTEM

Author: Zsombor Nagy (NIIF)

Last updated: 2008.05.28.

The new ARC storage system

The new ARC storage system is a distributed system for storing replicated *files* on several Storage Elements and manage them in a global namespace. The files can be grouped into *collections*, 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 Names* (LN). Besides the Logical Names each file and collection has a globally unique ID called GUID which comes from a flat namespace and can also be used for referencing files or collections but for the end-user the human-readable path-like Logical Names are much more suitable.

Components of the storage system

The ARC storage system will contain these components:

- the **Hash**, which is a distributed database capable of storing objects with a unique ID, where these objects are property-value pairs grouped in sections.
- the Storage **Catalog**, which stores the metadata and hierarchy of collections and files using the Hash as database, it stores the location of replicas of a given file and maintaining an index to be able to quickly get all the files having a replica on a given Storage Element.
- **Storage Manager**, which provides a high-level interface to the ARC storage
- **Storage Element**, which provides a unified interface for flat file stores using different backends to be able to use the free space of third-party storages, and includes a native implementation of a simple file store
- **Storage Gateway**, which provides a unified interface to third-party storages which have their own namespace to mount them into the global namespace of the ARC storage
- **client** API, CLI and GUI clients

IDs used in the system

There are a number of IDs used in the ARC storage system, such as:

- Each service has a unique **serviceID** which can be used to get an endpoint reference from the information system. We need an endpoint reference which is an address (URL) which we could connect to.
- The Storage Elements in the system identify their files using a **referenceID**.
- The **location** of a replica consists of two IDs: the ID of the Storage Element and the ID of the file within the Storage Element: (serviceID, referenceID).
- Each file and collection has a globally unique ID called **GUID**.
- The files and collections are organized into a global hierarchical namespace and can be referred to using paths of this namespace called **Logical Names (LN)**.

The Logical Name (LN)

The syntax of Logical Names (LN): [<GUID>][/<path>]
Each file and collection has a GUID which is globally unique, so they can be unambiguously referred using this GUID, that's why a single GUID is a Logical Name itself.
In a collection each entry has a name, and this entry can be a sub-collection, in which there are files and sub-sub-collections, etc. Example: if we have a collection with GUID '1234', and there is a collection called 'abc' in it, and in 'abc' there is another collection called 'def', and in 'def' there is a file called 'ghi', then we can refer to this file as '/abc/def/ghi', if we know the GUID of the starting collection, so let's prefix the path with it: '1234/abc/def/ghi'. This is the Logical Name of that file. If

there is a well-known system-wide root collection (its GUID could be e.g. '0'), then if a LN starts with no GUID prefix, it is implicitly prefixed with the GUID of this well-known root collection, e.g. '/what/ever' means '0/what/ever'.
If a client wants to find the file called '/what/ever', the client knows where to start the search, it knows the GUID of the root collection. The root collection knows the GUID of 'what', and the (sub-)collection 'what' knows the GUID of 'ever'. If the GUID of this file is '5678', and somebody makes another entry in collection '/what' (= '0/what') with name 'else' and GUID '5678', then the '/what/else' LN points to the same file as '/what/ever', so it's a hard link.
Each VO should create a VO-wide root collection, and put it in the generic root collection, e.g. if a VO called 'vo1' creates a collection called 'vo1' as a sub-collection of the root collection (which has the GUID '0'), then it can be referred as '0/vo1' or just '/vo1'. Then this VO can create some files, and put them in this '/vo1' collection, e.g. '/vo1/file1', etc. Or sub-collections, e.g. '/vo1/col1', '/vo1/col2/file3', etc. For this the VO does not need the install any service. These files and collections can be created using a Storage Managers.

Storage Managers

Clients can access the storage system through a Storage Manager. If a client wants to create a collection, upload or download a file, the first step is to connect a Storage Manager. The Storage Manager then resolves Logical Names and gets metadata using the Catalog, initiates file transfers on some storage elements, and gives the transfer URL to the client, which allows the client to directly transfer from/to a storage element. So the data transfer itself is not going through the Storage Manager, it is performed over a direct link between a storage element and the client.

The Catalog

The Catalog is a distributed service capable of managing the hierarchy of files and collections, storing all of their metadata. Each file and collection in the Catalog has a globally unique ID (GUID). A collection contains files and other collections, and each of these entries has a name unique within the collection very much like entries in a usual directory on a local filesystem. Besides files and collections the Catalog stores a third type of entries called Mount Points which are references to Storage Gateway services creating the capability to mount the namespace of third-party storages to our global namespace and make the files on a third-party storage available through the interface of the ARC storage system. The Catalog uses the Hash as database.
The Catalog also stores information about registered Storage Elements and receives heartbeat messages from them and change replica states automatically if needed.

The Hash

The Hash is a distributed service capable of consistently storing objects containing property-value pairs organized in sections. All metadata about files and collections are stored in the Hash, and some information about Storage Elements is stored in it as well.

Storage Elements

When a new file is put into the system the number of needed replicas is given for the file. The file replicas are stored on different Storage Elements.
The naming of files on an ARC Storage Element has nothing to do with the the hierarchy of collections, or Logical Names. When a replica is stored on a Storage Element it gets an ID which refers to it within that Storage Element. Each Storage Element has a unique ID itself, so with these IDs the replica can be unambiguously referenced, this is called a Location. The namespace of these

Locations has nothing to do with the namespace of GUIDs or the namespace of Logical Names. It consists of two IDs: the ID of the Storage Element and the ID of the file within the Storage Element: (serviceID, referenceID)

Heartbeats and replication

Each Storage Element is to periodically send heartbeats to the Catalog with state information about replicas whose state changed since the last heartbeat, The Catalog stores these file lists (which contains the GUIDs of the files as well), and if it doesn't receive a heartbeat for a Storage Element in a given time, it invalidates all the replicas the Storage Element stores. This invalidating means that the state of that location will be 'offline'.
If a Storage Element finds out that a file is missing or has a bad checksum, it reports this to the Catalog immediately, and the Catalog alters the state of the given replica of the file to 'invalid'.
The Storage Element tries to recover its replica by downloading it from an other Storage Element. In order to do this the Storage Element contacts a Manager and gets the file. The Manager chooses a valid replica, initiate file transfer by a Storage Element having a valid replica, and returns the TURL to the Storage Element with the invalid replica. The Storage Element downloads the file from the other Storage Element, and if everything is OK, signals to the Catalog that the replica is 'alive' again.
The Storage Elements periodically ask the Catalog whether the files they store have enough replicas. If a Storage Element finds that one of the files has not enough replica it turns to a Manager offering replication. The Manager chooses a Storage Element, initiates a put request then returns the TURL to the offering Storage Element which could upload the replica. The Storage Element who has now the new replica notifies the Catalog. The Catalog sets the state of this new replica to 'alive'

Scenarios

Downloading a file

We want to download a file about which we know that it is somewhere in our home collection on the storage. The LN of our home collection is e.g. '/ourvo/users/we'. We can get a list of entries in this collection from any Storage Manager.

- We need to find a Storage Manager. Maybe we have a cached list of recently used Managers or we can get one from the information system.
- When we have an endpoint reference of a Manager, we could call its list method with the LN '/ourvo/users/we'.
- The Manager has to find a Catalog service, again using its cache of recently used Catalog services or get a new one from the information system.
- The Manager has an endpoint reference of a Catalog service, it could ask the Catalog to traverse the LN '/ourvo/users/we'.
- The Catalog needs a Hash service to access the catalog data, when it has the endpoint reference of one Hash service, it could get the information about the root collection, which contains the GUID of the 'ourvo' sub-collection. Then the Catalog 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 Catalog returns to the Manager with all the metadata.
- The Manager now has the GUID and the metadata of the collection '/ourvo/users/we', including the list of its entries. This is returned to us.
- 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 Storage Manager 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 Manager need not to look up the whole LN again in the Catalog, a well-written client API should use this. With the get request we give the Storage Manager the list of transfer protocols we are able to use. (Here we could specify a list of Storage Elements which we prefer, but this is optional).
- The Manager contacts the Catalog to get the locations of the replicas of this file and the Catalog returns them. The Manager chooses a Storage Element. In the chosen location there is the ID of the Storage Element, and the referenceID of the file. Using the information system or its local cache it could get the endpoint reference of the Storage Element.
- The Manager initiate a transfer by the Storage Element providing our identity. The Storage Element decides if we are eligible to download this file. Hopefully the Storage Element supports one of the transfer protocols we give, and can create a transfer URL (TURL) with a protocol we can download. The Storage Element returns the TURL to the Manager, and the Manager returns it to us along with the checksum of the file.
- Now we have a TURL from which we can download it and checks if it is OK using the checksum.

Uploading a file

We have a file on our local disk we want to upload to a collection called '/ourvo/common/docs'.

- We contact a Storage Manager to put the file, we give the size and checksum of it, the transfer protocols we want to use, how many replicas we want, etc. We could specify which Storage Elements we prefer if we want. 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'
- The Manager uses the Catalog to get the GUID of the LN '/ourvo/common/docs' and check if the name 'proposal.pdf' is available in this collection.
- Then creates a new file entry within the Catalog with all the information we gave. The Catalog returns the GUID of this new entry.
- Then the Manager add the name 'proposal.pdf' and this 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 with 'try again later'.
- The Manager chooses a Storage Element with considering our preference, and from the information system it get some information about the chosen Storage Element including its endpoint reference. Then the Manager initiates the putting of the file on the Storage Element, the request includes the size and checksum of the file, the GUID, and the protocols we are able to use.
- The Storage Element 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 Catalog that there is a new replica with state 'creating'.
- The Catalog gets the message from the Storage Element and creates a new entry in the list of locations of the given while with the serviceID and the referenceID the Storage Element have just reported. If someone tries to download this file now, still gets a 'try again later' error message.
- The Storage Element returns the the TURL to the Manager, which is then returned to us.
- Then we can upload the file to this TURL.
- The Storage Element detects that the file is arrived and reports the change of state to 'alive' to the Catalog who alters the state in the given file-entry. At this point the file has only one replica. The

Storage Element periodically checks the Catalog if this is less than the needed replica number, and if it is then it initiates creating a new replica by a Storage Manager.
- The Manager chooses another Storage Element, initiates the transfer then returns the TURL to the first Storage Element which uploads the file to the new Storage Element. Both Storage Elements check periodically that their files have enough replica, and if they both find that there is more replica needed, they both initiate creating a new. This of course could cause that there will be more replicas than needed. If a Storage Element finds out that a file has more replicas than needed it notifies a Manager about it.
- The Manager ask the Catalog about all Storage Elements this file has replicas on, flags this file as 'removing a replica' which prevents other Managers to remove an other replica accidentally, then make a decision of which one is to be removed, then contacts the chosen Storage Element and asks it to remove the replica. The Storage Element then notify the Catalog, and the Catalog removes the replica, and removes the flag 'removing a replica' as well.
- If we cannot upload the file to the given TURL for some reason, we should remove the file entry from the collection, or we should call 'addReplica' to get a new TURL without removing and recreating the file.

Removing a file

- If we want to remove a file, we should connect to a Storage Manager with the LN of the file we want to remove.
- The Storage Manager asks the Catalog about the locations of the replicas of this file than ask the Catalog to remove the entry completely. Then asks all the Storage Elements which have replicas to remove it. Then the Storage Manager removes the entry from the collection in which the file is.
- After a Storage Element removes a replica, it notifies the Catalog to remove the location of that replica, the Catalog now knows nothing about this file, but this cause no problem.
- If something happens to the Manager after removing the file entry from the Catalog and before asking all the Storage Elements to remove, there will be Storage Elements which do not know they should remove the replica. But next time the Storage Element does its periodic check, it finds out that it has a replica whose file does not exist and then it could remove the replica.
- TODO: what about files which are in a closed collection?

Prototype status and plans

The current version of the prototype has no information system and security, these are soon to be integrated to the system. The information system is needed to discover services, and to translate serviceIDs to URLs. Currently the URLs are written in the config files. The sercurity 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 the properties extracted from the communication channel and these policies to make authorization decisions. If the properties and the policies are present, the decision is actually made by the security framework of HED.
Further prototype statuses and plans can be found below within each section about the services.

Hash

Functionality

The Hash is a distributed service capable of storing objects containing property-value pairs grouped in sections in a scalable manner. Each object has an arbitrary string ID, and contains any number of property-value pairs grouped in sections, where property, value and section are arbitrary strings. There could only be one 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 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.

Prototype status and plans

The Hash service currently implemented as a single central service, which stores the data on disk in separate files per object. In the fall of 2008 it will be reimplemented on a distributed hash table (DHT) algorithm, for example the Chord algorithm with a consistency solution called Etna on top of it. This reimplementation hopefully won't change the interface of the service.

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

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 values for each property in each section filtered by *neededMetadata* within *object* which is a list of (section, property, value) tuples.
- **change**(changeRequest): returns changeResponse which is a list of (changeID, success, failedConditionID) tuples.
  changeRequest is (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 proper from thesection 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.

Catalog

Functionality

The Catalog 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 Catalog 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 Catalog. 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.

Prototype status and plans

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

Data model

Each catalog entry has a unique ID called **GUID.**
The Catalog uses the Hash to store all the data about the files and collections. The 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. ('catalog','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:
    - **catalog**
        - *type*: "collection"
    - **entries**
        - *(name, GUID) pairs*: a Collection is basically a list of name-GUID pairs.
    - **timestamps**
        - *created*: timestamp of creation
        - *modified*: timestamp of last modification
    - **states**
        - *closed*: if the collection is closed, then nothing can be added to its contents
    - **policies**
        - XML representations of access policies
    - **metadata**
        - any other arbitrary metadata
- A **File**: a File entry contains the following sections:
    - **catalog**
        - *type*: "file"
    - **locations**

- *(location, state) pairs*, where a location is a (serviceID, referenceID) pair serialized as a string, where *serviceID* is the ID of the Storage Element service storing this replica, *referenceID* is the ID of the file within that store, and *state* could be '**alive**' (if the replica passed the checksum test, and the storage element storing it is healthy), '**invalid**' (if the replica has wrong checksum, or the storage element claims it has no such file), '**offline**' (if the storage element is not reachable, but may has a valid replica), '**creating**' (if the replica is in the state of uploading), '**sentenced**' (if the replica is marked for deletion)

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

states
- *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
- XML representations of access policies

metadata
- any other arbitrary metadata

- A **Mount Point**: there is one more type of Catalog entries called Mount Point which is a reference to a Storage Gateway which is capable of handling a subtree of the namespace. The properties of a Mount Point in sections:

catalog
- *type*: "mountpoint"

mount
- *target*: the ID of the Storage Gateway
- *ID*: optional ID within the Gateway

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

policies
- XML representations of access policies

metadata
- any other arbitrary metadata

The Catalog stores information about the Storage Elements, so each Storage Element 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 Storage Element:

nextHeartBeat
- (ID, timestamp) pairs

serviceGUID
- (ID, GUID) pairs

For each Storage Element there is a separate entry with the list of files:

catalog
- *type*: "storageelement"

file
- *(referenceID, GUID)* pairs for each replica stored on the Storage Element

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 into the Hash, then returns the GUIDs of the newly created entries.
- **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 the value), '**unset**' (remove the property-value pair from a section), '**add**' (set the property in the section to the 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
- **remove**(removeRequestList): returns a list of (requestID, success) pairs
  removeRequest is a list of (requestID, GUID) pairs.
- **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 if the full LN could be traversed;
  traversedList is a list of (LNpart, GUID) pairs, where *LNpart* is a part of the LN, *GUID* is the GUID of the Catalog-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 number of seconds, which is the timeframe within the Catalog expects the next heartbeat from the Storage Element
  filelist is a list of (GUID, referenceID, state) indicating files with changed state or which are new, where *state* could be '**invalid**' (e.g. the periodic self-check of the Storage Element found a non-matching checksum or missing file), '**creating**' (if this is a new file just being uploaded) or '**alive**' (if the new file was uploaded and now become alive).

Storage Elements

Functionality

A Storage Element is capable of storing files, it keeps track all the files it stores with their GUIDs and checksums. The Storage Elements periodically send reports to a Catalog indicating that the Storage Element is up and running, and that some file's state has been changed. The Storage

Element periodically checks each file to detect corruption. If a file goes missing or has a bad checksum the Storage Element notify the Catalog about the error referring the file with its GUID.
A file in a Storage Element could be identified with a *referenceID* which is unique within the Storage Element. If we know the Location of a file, which is the ID of the Storage Element service plus the referenceID, we could get the endpoint reference of the Storage Element from the information system, then we should call its *get* method with the referenceID and a list of transfer protocols we are able to handle (e.g. 'HTTP', 'FTP'), the Storage Element 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.
Storing a file starts with initiating the transfer with the *put* method of the Storage Element, 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 Storage Element chooses a protocol, creates a TURL for uploading and generates a referenceID, than we can upload the file to the TURL.
These TURLs are one-time URLs which means that after the client uploads or downloads the file these TURLs cannot be used again to acces the file. If we want to download the same file twice, we have to initiate the transfer twice, and will get two different TURLs.
With the *stat* method we can get the state of a replica ('creating', 'alive' or 'invalid'). The *delete* method removes the file.
In normal operation the put and get calls is made by a Storage Manager but the actual uploading and downloading is done by the user's client. In case of replication a Storage Element initiates the replication which has a valid replica of a file, this Storage Element asks the Storage Manager to choose a new Storage Element, the Storage Manager initiates putting the new replica on chosen Storage Element and receives a TURL, then the Storage Manager returns the TURL to the initiator Storage Element, which uploads its replica to the given TURL.
The Storage Element service cannot work without at least one backend. A storage element backend is able to communicate the local native or third-party storage service (e.g. an FTP, HTTP or ByteIO server). This backend has to implement some methods like prepareToGet, prepareToPut, copyTo, copyFrom, list, and methods to generate local IDs, checksums, etc.

Prototype status and plans

The current implementation of the Storage Element service has working get, put, stat methods. There is a separate service which provides a ByteIO-like interface for accessing files, and there is a backend for the Storage Element which can communicate with this separate service.
Further plans include other backends for third-party services, and a full implementation of the ByteIO interface.

Data model

A file in a storage element is referenced by its *referenceID.* Each file has a state which could be '**creating'** when it is just being uploaded, **'alive'** if it is alive or **'invalid'** if it does not exists anymore or has a bad checksum. Each file has a *localID* which is used in the backends.

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; '**protocol**' indicates a protocol the client is able to use, there could be multiple protocols in getRequestData. The *getResponseData* is a

list of property-value pairs, such as: '**TURL**' is a URL called Transfer URL which can be used by the client to download the file; '**protocol**', the TURL usually contains the protocol, but just in case the chosen protocol is also returned; '**checksum**' is the checksum of the replica, '**checksumType**' is the name of the checksum method, '**error**' could contain an error message.

- **put**(putRequestList): returns a list of (requestID, putResponseData)
  putRequestList is a list of (requestID, putRequestData) where *requestID* is an ID used for response, putRequestData is a list of property-value pairs such as '**GUID**' and '**checksum**', '**checksumType**' for the GUID and checksum of the file, this is needed for a better self-healing, '**size**' is the size of the file in byte and '**protocol**' is a protocol the client is able to use (can be multiple), '**acl**' contains the access policy for this file. The 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 of this new replica, '**error**' could contain 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 status we want to get.,

Interface of the backends

- **prepareToGet**(referenceID, localID, protocol): returns the *TURL.*
  Initiate transfer with *protocol* for the file '*localID*' which also known as *referenceID*, which information could be used by the service later, e.g. when the transfer finished and the state of the file needs to be changed, the referenceID is needed for this.
- **prepareToPut**(referenceID, localID, protocol): returns the *TURL.*
  Initiate transfer with *protocol* for the file '*localID*' which also known as *referenceID*.
- **copyTo**(localID, turl, protocol):
  upload the file referenced by *localID* to the given *TURL* with the given *protocol*.
- **copyFrom**(localID, turl, protocol):
  download the file from the given *TURL* with the given *protocol*, and register it as *localID*.
- **list**():
  returns a list of *localID*s.
- **getAvailableSpace**():
  returns the available disk space in bytes.
- **generateLocalID**():
  returns a new unoccupied *localID*.
- **matchProtocols**(protocols):
  returns a list of protocols which are supported by the transfer service and are also members of the given *protocols* list.
- **checksum**(localID, checksumType):
  returns the checksum of the file referenced by *localID*.

Storage Managers

Functionality

The Storage Manager provide an easy to use interface of the ARC storage to the users. You can put, get and delete files using their Logical Names with *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* gives all the information about a file or collection, and you can move collections and files with *move*, copy files with *copy*, and search for matching path names with *glob.* You can upload an entirely new replica to a file (e.g. if the file lost all its replicas) with *addReplica*.

Data model

The Storage Manager interface uses mostly Logical Names (LNs), which have the syntax of: ‘<GUID>/<path>’ where both sides can be omitted, e.g. ‘afg342/foo’ is an entry called ‘foo’ in the collection with GUID ‘afg342’; ‘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 the Catalog section of this document.

Prototype status and plans

The *getFile*, *putFile*, *addReplica*, *makeCollection*, *list*, *move* and *stat* methods are implemented, but need more error-checking and metadata-checking. The *modify*, *delFile*, *unmakeCollection*, *copy* and *glob* methods will soon be implemented.

Interface

- **putFile**(putFileRequest): returns a list of (requestID, success, TURL, protocol)
  putFileRequest is a list of (requestID, LN, metadata, protocols), where *requestID* is an arbitrary ID used in the response; *LN* is the chosen Logical Name of the new file, *protocols* is a list of protocols we want to 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’ sectio, 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 (...)’, etc.
- **getFile**(getFileRequest): returns a list of (requestID, success, TURL, protocol)
  getFileRequest 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 (...)’
- **delFile**(delFileRequest): returns a list of (requestID, status)
  delFileRequest is a list of (requestID, LN) with the Logical Name of the file we want to delete. The *status* in response could be ‘deleted’, ‘nosuchLN’, ‘denied’.
- **stat**(statRequest): returns a list of (requestID, metadata)

the *statRequest* 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 Catalog.

- **makeCollection**(makeCollectionRequest): returns a list of (requestID, success)
  makeCollectionRequest 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 (the entries in the new collection will be hardlinks 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 later), in the 'policies' section there could be access policies, and in the 'metadata' section there could be any other metadata.
  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.
- **list**(listRequest, neededMetadata): returns listResponse
  listRequest 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 Catalog, *status* could be 'found', 'not found', 'is a file'.
- **move**(moveRequest): returns a list of (requestID, status)
  moveRequest 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'.
- **unmakeCollection**(unmakeRequest): returns a list of (requestID, status)
  unmakeRequest is a list of (requestID, LN) with all the Logical Names of the collections we want to remove.
- **modify**(modifyRequest): returns a list of (requestID, status)
  this method is not designed yet, maybe similar to the *modifyMetadata* method of the Catalog
- **copy**(copyRequest): returns a list of (requestID, status)
  copyRequest is a list of (requestID, sourceLN, targetLN) where *sourceLN* is the path of the file we want to duplicate, the *targetLN* is the new path.
- **glob**(globRequest): returns a list of (requestID, list of LNs)
  globRequest is a list of (requestID, pattern) where *pattern* is a usual pattern used for paths. For each request a *list of LNs* is returned with all the LNs matched the pattern.

Storage Gateways

Functionality

A Storage Gateway is a wrapper for a third-party storage solution, which makes it possible to mount an external storage with existing data and its own namespace to the global storage namespace of the ARC Storage.

A particular Storage Gateway has a backend which knows the interface of the connected third-party storage and try to translate the method calls, the *get*s and *put* and the ACL modifications, and try to create property-value pairs grouped in sections according to the data model of the ARC Catalog as well.

Prototype status and plans

This service is not included in the prototype.

Data model

A file in a storage element is referenced by a path which is local in the namespace of the third-party storage.

Interface

- **get**(getRequest): returns list of (requestID, getResponseData)
  very similar to the *get* method of the Storage Element
- **put**(putRequest): returns a list of (requestID, putResponseData)
  very similar to the *put* method of the Storage Element
- **delete**(deleteRequest): returns a list of (requestID, status)
  very similar to the *delete* method of the Storage Element
- **stat**(statRequest): returns a list of (requestID, statResponse)
  very similar to the *stat* method of the Storage Element
- **list**(listRequest): returns listResponse
  very similar to the *list* method of the Catalog
- **move**(moveRequest): returns a list of (requestID, status)
  very similar to the *move* method of the Manager