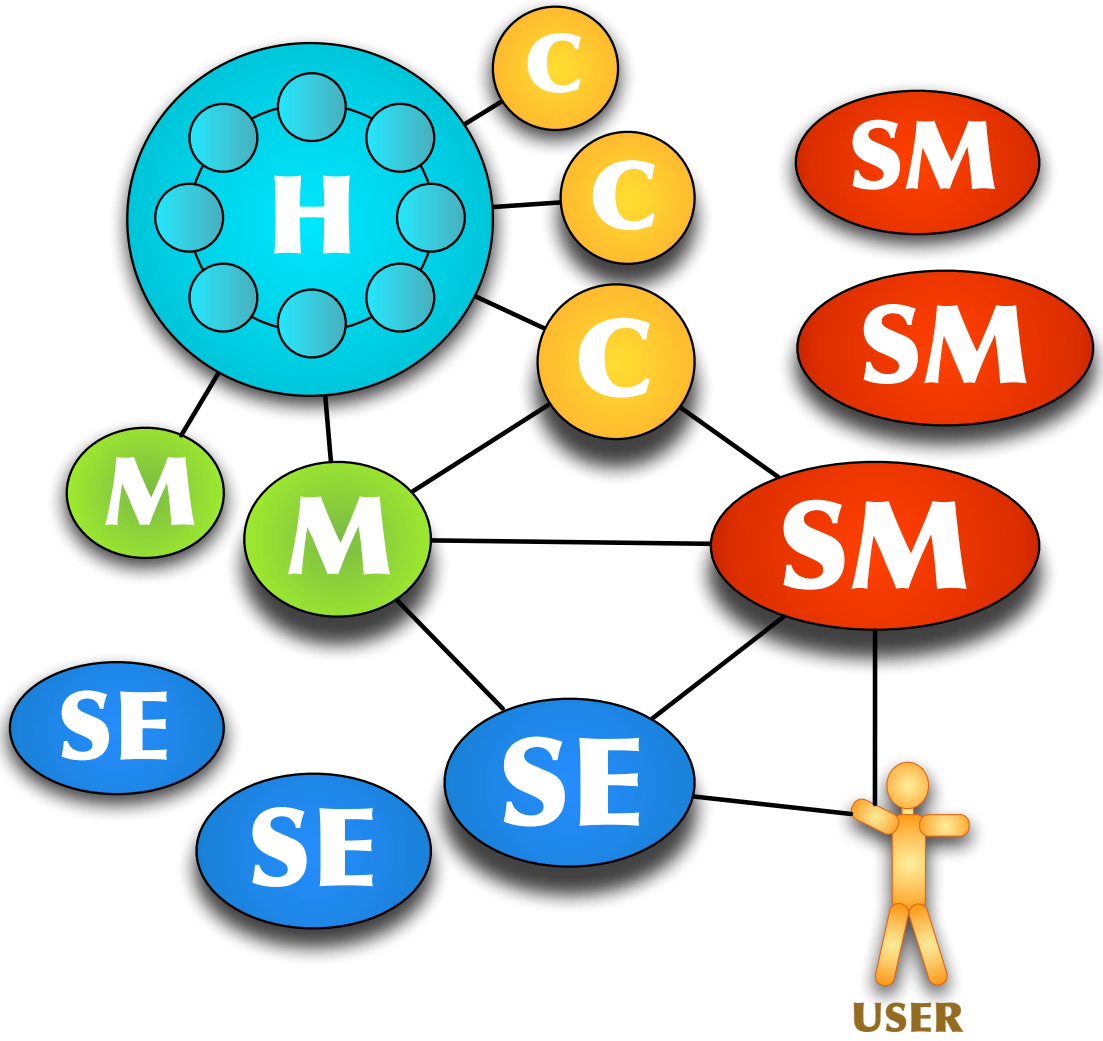
The ARC storage system

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 attribute-value pairs (a Mutable Distributed Hash Table presumably using the Etna[1] protocol)



The services of the ARC Storage: the Hash, the Catalog, the Maintainer, the Storage Manager and the Storage Element

[1] Etna: a Fault-tolerant Algorithm for Atomic Mutable DHT Data; http://www.lcs.mit.edu/publications/pubs/ps/MIT-LCS-TR-993.ps

- the Storage **Catalog**, which stores the metadata and hierarchy of collections and files using the Hash as database.
- the Storage **Maintainer**, which handles the replication of files and monitors the health of the Storage Elements (using the Hash as database)
- **Storage Manager**, which provides a high-level interface to the ARC Storage
- **Storage Element**, which provides a unified interface for storing and retrieving files using different back-ends including a **native implementation** of a simple file store and **wrappers** to third-party solutions (e.g. FTP, GridFTP, dCache, etc.)
- **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 which we could connect to.
- Each user should have a unique **ID** which we use e.g. when we specify the owner or the access control list of a file or collection. This could be e.g. the Distinguish Name in an X.509 certificate.
- The Storage Elements in the system organize their files into stores which has a **storeID**. The stores are useful if we want to use multiple backends in one Storage Manager instance. Within a store a file is identified with a **referenceID**. A storeID, referenceID pair unambiguously selects a file within a Storage Element.
- The **location** of a replica consists of three IDs: the ID of the Storage Element, the ID of the store within the Storage Element, and the ID of the file within the store: (serviceID, storeID, referenceID).
- Each file and collection has a globally unique ID called **GUID**.
- The files and collections are organized into a 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, but we put a slash on the end of it to indicate that this is a Logical Name: '1234/'.
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 some assertions (some kind of certificate) to the client, which allows the client to actually do the file 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 catalog 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 creates the capability to mount the namespace of third-party storage solutions 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 a distributed database.

The Hash

The Hash is a distributed service capable of consistently storing objects containing attribute-value pairs. It will be most likely based on a distributed hash table (DHT) algorithm called Chord with a consistency solution called Etna on top of it. All metadata about files and collections are stored in the Hash, and the Maintainer service uses the Hash for storing information about ongoing replications and the health of the Storage Elements 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 hierarchy 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.

Maintainer services

These services are responsible for ensuring each file has at least as many valid replicas as needed. The Storage Elements send heartbeats to the Maintainer services which keeps track of the list of files on each Storage Elements so if one of them goes offline the Maintainer knows which files are short of replicas and it can create more replicas.

Standards

The ARC Storage internally does not use any storage related standards, but the components which may be used separately could provide standard interfaces as well, e.g. the Storage Managers and maybe the Storage Elements can provide an SRM interface or GridFTP.

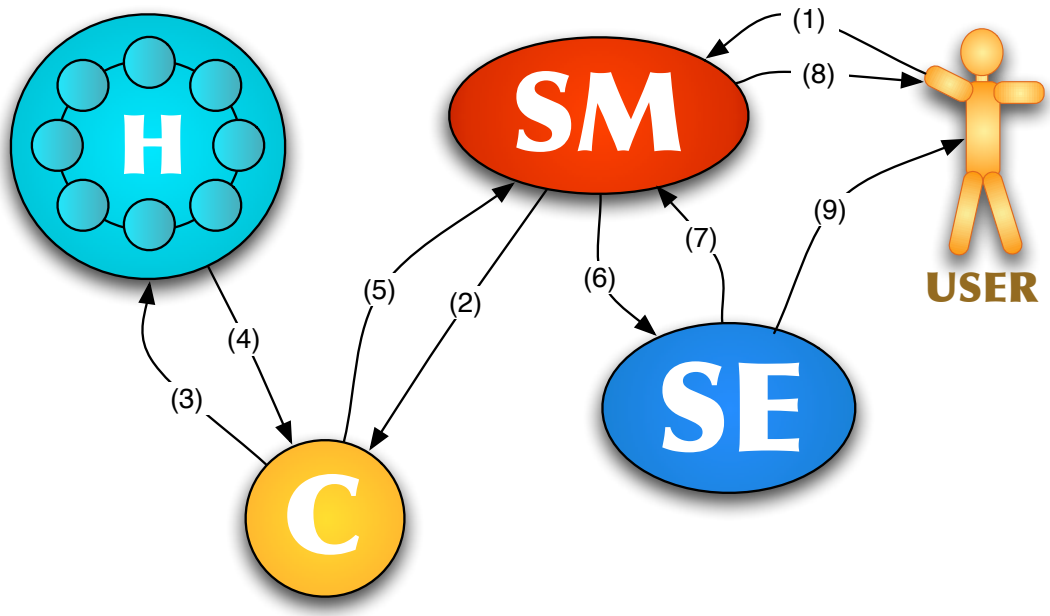
Scenarios

The following scenarios currently deals with no authorization. It is very urgent to identify the steps we need for a proper authorization.

Downloading a file

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

- We want to find a Storage Manager. Maybe we have a cached list of recently used Managers or we can get one from the information system.
- We have an endpoint reference of a Manager, we could call its *list* method (1) 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'. (2)
- 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



Downloading a file from the ARC Storage: (1) user initiates the downloading; (2) Storage Manager wants to get the locations of the file replicas; (3) Catalog gets data from the Hash; (4) Hash returns data; (5) Catalog returns replica locations; (6) Storage Manager initiates transfer; (7) Storage Element returns TURL; (8) Storage Manager returns TURL; (9) user downloads the file

it can find the GUID of ‘users’, and in the entries of ‘users’ there is the GUID of ‘we’ (3,4), which the Catalog returns to the Manager. (5)

- The Manager now has the GUID of the collection ‘/ourvo/users/we’, it connects the Catalog again to get all information about it (2), the Catalog replies with all the metadata of the collection and the list of its entries. (5)
- The Manager now has the names and GUIDs of all the entries in ‘/ourve/users/we’, but the list method should return timestamps and sizes etc. about these entries, so the Manager ask the Catalog again about all these GUIDs (2), the Catalog replies with the needed metadata about them. (5)
- Now the Manager has all the information we need, which is returned to us. (8)
- 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 (1), 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 Manager a list of our preferred Storage Elements and a list of transfer protocols we are able to use.
- The Manager contacts the Catalog to get the locations of replicas of this file (2), the Catalog connects the Hash (3) to get the metadata of the file (4) and returns it to the Manager (5). The Manager uses some decision algorithm and chooses one location, if the file has a replica on one of our preferred Storage Elements it will most likely choose that location. In the chosen location there is the ID of the Storage Element, and the storeID and referenceID of the file within. 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 with our list of transfer protocols. (6) Hopefully the Storage Element supports one of these, and can create a transfer URL (TURL) with a protocol we can download. The Storage Element returns the TURL to the Manager (7), and the Manager returns it to us along with the checksum of the file. (8)
- Now we have a TURL from which we can download it (9) 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, which Storage Elements we prefer, who is the owner of this file, who has what kind of rights, etc. And of course we give the Logical Name we want to call 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 asks the Maintainer to choose 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 Maintainer that there is a new replica with state 'creating'. If someone tries to download this file now, still gets a 'try again later' error message.
- The Manager adds the referenceID along with the storeID and the serviceID of the Storage Element to the Catalog into the file entry as a new location. This new location initially has a state 'creating'. The Manager then returns the TURL of the file to the client.
- 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 Maintainer which alters the state of that replica of the file in the Catalog. At this point the file has only one replica, and if it needs more, then the Maintainer start the replication process, chooses a Storage Element, initiates the transfer and then asks the Storage Element with the existing replica to copy that file to this newly chosen Storage Element.
- 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 initiate a 'reput' to get a new TURL without removing and recreating the file.

Storage Elements and Maintainers

Each Storage Element registers itself to a Maintainer and periodically send heartbeats and report of state changes of its files. Also it checks periodically all of it files if they do exist and has a proper checksum.

- If a Storage Element finds out that a file is missing or has a bad checksum, it reports this to any Maintainer using the GUID of the file.
- The Maintainer uses the GUID to find the file in the Catalog, and sets the state of its replica in question to 'invalid', then if the file has not enough replicas, it starts the creation of more replicas.
- The Maintainer keeps track of all the registered Storage Elements, and knows when they sent a heartbeat last time. If a Storage Element won't send a heartbeat within a given time it is considered offline. The Maintainer has a 'Storage Elements to GUIDs' mapping and using it all the files can be found which has replica on this Storage Element, and the state of these replicas could be set to 'offline', and all files which now has less replicas than needed will be inserted into the queue of the Maintainer to create more replicas.
- If a Storage Element comes online again it is asked to report all the files it has, and if it claims that the replicas are intact, all the states could be changed back to 'valid'.

Removing a file

- If we want to remove a file, we should connect to a Storage Manager, and tell it what is the LN of the file we want to remove.
- The Storage Manager removes the entry from the collection in which the file is. (TODO: what about hard links? maybe reference counting is needed.)
- The Storage Manager then marks the file for deletion, and asks the Maintainer to remove it.
- The Maintainer starts to remove the replicas with asking each Storage Element to remove, then it removes the Catalog entry itself.
- If a Storage Manager is not reachable, then the next time it goes online and reports the files it has the Maintainer could notify it to remove the unneeded replicas.

Using a third-party storage as flat store

We have an FTP server which we want to use as a Storage Element in ARC Storage, but we do not want the existing files on the FTP server to appear in the namespace of the ARC Storage.

- We should install a Storage Element with the flat-FTP backend. It is not needed for it to be on the same machine as the FTP server.
- We choose a directory on the FTP server where we want to store the files and then configure the Storage Element to expose it as a flat store, e.g. with ID ‘0’. The Storage Element advertises itself as having a flat store with storeID ‘0’.
- When the Maintainer chooses this store of this Storage Element to store a replica, a put request arrives, the Storage Element generates a referenceID and asks the FTP server to create a temporary directory with a temporary username and password and encapsulate these into a simple URL (TURL).
- The Storage Element returns the TURL and the referenceID and monitors the temporary directory. When a file arrives, the Storage Element asks the FTP server to move it to the permanent directory, and removes the temporary user and the temporary directory. Later the referenceID could be used to get this file.
- When a get request arrives the Storage Element asks the FTP to create a temporary link for this file, and a temporary user who has only read rights to this temporary link. This forms the TURL.
- If the Storage Element could get the information from the FTP about this temporary file that it is already downloaded once, then asks the server to remove this user and the link as well. If it could not get this information from the FTP, then after a given timeout it will remove the file by all means.
- Maybe we could create a more sophisticated way to ensure that the user won’t mess up the internal structure of the FTP server during the direct transfers.

Using a third-party storage as a hierarchical store

We have an FTP server with files and directories on it which we want to share and mount into the ARC Storage.

- We need to install a Storage Element with the hierarchical FTP-backend and configure it with the parameters of the FTP server and the directory on the FTP server we want to share (e.g. there is a directory on the FTP server called ‘/bill’, and we want to share its contents). This defines a hierarchical store on the Storage Element, e.g. with an ID, say ’42’.
- We create a mount point in the Catalog e.g. at ‘/ourvo/share/steve’, this will point to this Storage Element with storeID ’42’.
- We connect a Storage Manager wishing download a file with LN ‘/ourvo/share/steve/jobs’.
- The Storage Manager finds out that ‘/ourvo/share/steve’ is a mount point which points to a particular Storage Element, so it turns directly to the Storage Element specifying the storeID ’42’ and a referenceID which contains the remaining part of the Logical Name which is ‘/jobs’.
- The Storage Element receives a get request with storeID ’42’ and referenceID ‘/jobs’. The FTP backend concatenate the configured ‘/bill’ path with this referenceID and gets a path ‘/bill/jobs’. It creates the TURL which is a URL with FTP protocol pointing to the file ‘/bill/jobs’. It may asks the FTP server to create a temporary user with read rights only to this file, and later remove it, or use some other method which prohibits the misuse of this TURL.
- If we want to upload a file to ‘/ourvo/share/steve/gates’ we ask the Manager to do so. The Manager checks the Catalog and gets the information about the mount point, so it turns directly to the Storage Element with the storeID to initiate the put transfer. But in this case along with the storeID there is a referenceID as well with the remaining part of the LN: ‘/gates’.

- The Storage Element receives the put request with storeID ’42’ and referenceID ‘/gates’. This is a hierarchical store so the referenceID is not ignored as it would be by a flat store. The FTP backend concatenates the configured ‘/bill’ with the referenceID ‘/gates’, then it gets the path of the new file, and asks the FTP server to create a temporary directory and a temporary user who has write access only to this temporary directory. It creates a TURL containing the temporary user and the temporary directory, and returns it.
- The Storage Element monitors the file and if it arrived, the Storage Element asks the FTP server to move it to ‘/bill/gates’.

Hash

Functionality

The Hash is a distributed service capable of storing objects containing attribute-value pairs in a scalable manner. Each object has an ID from a fixed width binary space, and contains any number of attribute-value pairs, where attribute and value are arbitrary strings. If there are multiple occurrences of an attribute then it could be considered that a list of values belongs to that attribute.
There could be any number of Hash services in the system and it does not matter which one a client connect to, the answer will be the same. If you have an ID, you can get all attribute-value pairs of the corresponding object with the *get* method. You can add or remove an attribute-value pair in an object, or delete all occurrences of an attribute with the *change* method, and you can request conditional changes, when the change only would be applied if the given conditions are met with the *changeIf* method.

Data model

- *ID* is a fixed width binary number
- *object* is a a list of attribute-value pairs, where attributes and values are strings, and there could be multiple occurrences of attributes

Interface

- **get**(list of IDs): returns list of (ID, object) pairs.
  For each *ID* it returns the *object* (which is a list of attribute-value pairs) referenced by that ID.
- **change**(list of changes): returns a list of which changes was successful and which was not.
  a *change* is: (changeID, ID, attribute, AVChangeType, value), where changeID is an arbitrary ID which is used on return to identify which change was successful; ID points to the object we want to change; AVChangeType can be ‘add’ (add a new attribute-value pair), ‘remove’ (remove the attribute-value pair), ‘delete’ (remove all occurrences of an attribute, no value needed for this change)
  Try to apply changes to objects, creates object if a non-existent ID is given.
- **changeIf**(list of conditional changes): list of which changes was successful and which was not.
  a *conditional change* is a (changeID, ID, list of conditions, attribute, AVChangeType, value)
  a *condition* is an (attribute, conditionType, value), where conditionType can be ‘has’ (there is such an attribute-value pair), ‘not’ (there is no such attribute-value pair), ‘exists’ (there is at least one occurrence of this attribute, the value does not matter), ‘empty’ (there are not any occurrences of attribute, the value does not matter)
  For each conditional change if all conditions are met, try to apply the change.

Security-related questions

- Who has right to modify object in the Hash? Among the storage services the Catalog and the Maintainer use the Hash as their database, should we hardwire into all of the Hash services to accept only a special kind of certificates which we could be sure only Catalogs and Maintainers has?
- Should the objects have an owner and access control list?

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 *newCollection*, 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 with the *changeEntries* method, that makes the existing collection the parent of the new collection. A collection can be closed with *closeCollection*, this cannot be undone and prevents files to be added or removed from this collection. A new file can be created with the *newFile* 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 can be manipulated with *changeLocations*, the number of needed replicas can be altered with *changeNeededReplicas*. A mount point can be created with the method *newMountPoint*, and the target of an existing mount point can be changed with *changeTarget.* Each Catalog entry can have arbitrary metadata which can be altered with *changeMetadata*. The owner and the access control list (ACL) of an entry can be changed with the *changeOwner* and *changeACL* methods. The *remove* method deletes an entry from the Catalog. The *traverseLN* method try to traverse Logical Names 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 of the Catalog.

Data model

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

- 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 a Collection is a list of name-GUID pairs, plus there are also some well-defined metadata associated with the Collection in the form of attribute-value pairs such as:
  - created: timestamp of creation
  - modified: timestamp of last modification
  - owner: ID of owner
  - mutable: if the collection is closed, then it is not mutable anymore
  - access control list: a list of (ID, right), where ID could be an ID of a user, a VO or some special semantic, e.g. ‘everybody’; right could be ‘**list**’ (to list the contents of the collection), ‘**delete**’ (to remove an entry from the collection), ‘**add**’ (to add an entry to the collection), ‘**modify**’ (to change metadata, owner or close the collection)
  - ... any other arbitrary metadata
- A **File**: a File entry contains a couple of attributes such as:
  - size: the file size in bytes

  - checksum: checksum of the file
  - created: timestamp of creation
  - modified: timestamp of last modification (metadata, ACL, owner, etc.)
  - owner: ID of owner
  - access control list: list of (ID, right), where right could be '**read**' (to download the file or any part of it), '**write**' (to reset the file with a new replica), '**delete**' (to delete all replicas of the file)
  - number of needed replicas: how many valid replicas should this file have
  - state of the file, which could be '**normal**' and '**deleted**'
  - list of locations of the replicas: a location is a (serviceID, storeID, referenceID, state) tuple where *serviceID* is the ID of the Storage Element service storing this replica, *storeID* is the ID of the store within the Storage Element, *referenceID* is the ID of the file within that store, and *state* could be '**valid**' (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)
  - any other arbitrary metadata including list of preferred Storage Elements (the IDs of Storage Elements to use for storing replicas)
- A **Mount Point**: there is one more type of Catalog entries called Mount Point which is a reference to a Storage Element which handles a subtree of the namespace. Its attributes:
  - target: a pair of (serviceID, storeID) where serviceID is the ID of the Storage Element; storeID is an ID to identify the store within the Storage Element
  - created: timestamp of creation
  - owner
  - access control list: list of (ID, right), where right could be the same as of the Collection
  - any other arbitrary metadata

Interface

- **newCollection**(newCollectionRequestList): returns a list of (requestID, GUID)
  newCollectionRequestList is a list of (requestID, entrylist, mutable, owner, acl, metadata) where *requestID* is an arbitrary ID used to identify this request in the list of responses;
  entrylist is a list of (name, GUID) pairs, which are the actual content of this collection, empty by default;
  mutable is true by default, if it is false, then no more files can be added besides the initial entrylist;
  owner is the ID of the owner of this collection;
  acl is a list of (ID, right) pairs where ID is the ID of the user;
  metadata is a list of any arbitrary attribute-value pairs.
  This method generates a GUID for each request, and inserts the new collection entry into the Hash, then returns the GUIDs of the newly created collections.
- **changeEntries**(entryChanges): returns a list of (changeID, success)
  entryChanges is a list of (changeID, collectionGUID, entryChangeType, name, GUID) where entryChangeType is 'add' or 'remove'.
  For each change try to apply it to the entries of the collection referenced by collectionGUID.
- **closeCollection**(list of GUIDs): returns list(GUID, status)
  Try to close each given collections.
- **changeOwner**(ownerChanges): returns a list of (changeID, success)

ownerChanges is a list of (changeID, GUID, owner) where owner is the ID of the new owner

- **changeACL**(ACLChanges): returns (changeID, success)
  ACLChanges is a list of (changeID, GUID, ID, ACLChangeType, right) where *ACLChangeType* can be 'add' or 'remove' and *right* can be one of the valid rights of both the collection and the file, but of course only file-rights could be applied to a file and only collection-rights could be applied to a collection
- **changeMetadata**(metadataChanges): returns (changeID, success)
  metadataChanges is a list of (changeID, GUID, attribute, AVChangeType, value) where *AVChangeType* can be 'add' (add a new attribute-value pair), 'remove' (remove the attribute-value pair), 'delete' (remove all occurrences of attribute, no value needed for this change)
- **newFile**(newFileRequestList): returns a list of (requestID, GUID)
  newFileRequestList is a list of (requestID, size, checksum, locations, neededReplicas, owner, acl, metadata) where *requestID* is used for the response;
  size is the size of the file;
  checksum is some kind of checksum of the file;
  locations is a list of (serviceID, storeID, referenceID, state) and is empty by default;
  neededReplicas is an integer specifying the number of needed replicas of this file;
  owner is the ID of the owner;
  acl is a list of (ID, right) pairs where ID is the ID of the user;
  metadata is a list of any arbitrary attribute-value pairs (may include 'preferredSE' attributes)
  This method creates a new file entry in the Hash after generating a new GUID for it.
- **changeNeededReplicas**(replicaChanges): returns a list of (changeID, success)
  replicaChanges is a list of (changeID, GUID, neededReplicas) which gives for each GUID the new number of needed replicas.
- **changeLocations**(locationChanges): returns a list of (changeID, success)
  locationChanges is a list of (changeID, GUID, locationChangeType, serviceID, storeID, referenceID, state) where *locationChangeType* can be 'add' (to add a new location: serviceID, storeID, referenceID and state), 'remove' (to remove a location if all four attributes match)
- **get**(list of GUIDs): returns getResponseList
  getResponseList is a list of (GUID, type, created, modified, owner, acl, metadata, size, checksum, locations, neededReplicas, entrylist, mutable, target) where *type* is 'collection', 'file' or 'mountpoint' and the attributes are filled accordingly.
- **remove**(list of GUIDs): returns a list of (GUID, success) where *success* shows if the removing was successful or not
- **traverseLN**(traverseRequestList): returns traverseResponseList
  traverseRequestList is a list of (requestID, LN) with the Logical Name to be traversed
  traverseResponseList is a list of (requestID, traversedList, wasComplete, traversedLN, GUID, type, restLN) where:
  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;
  wasComplete indicates if the full LN could be traversed;
  traversedLN is the part of the LN which was traversed, if wasComplete is true, this should be the full LN;
  GUID is the GUID of the traversedLN;
  type is the type of traversedLN which can be 'collection', 'file' or 'mountpoint';
  restLN is the postfix of the LN which was not traversed for some reason, if wasComplete is true, this should be an empty string.

- **newMountPoint**(newMountRequestList): returns a list of (requestID, GUID)
  newMountRequestList is a list of (requestID, target, owner, acl, metadata) where *target* is a pair of (serviceID, storeID) where *serviceID* is the address of the Storage Element service handling this mount and *storeID* is an ID of the store within the Storage Element.
  After you create a mountpoint-entry with this method, you should add it to a collection to mount it to the global namespace.
- **changeTarget**(targetChanges): returns a list of (changeID, success)
  targetChanges is a list of (changeID, GUID, target) where *GUID* is the GUID of the Mount Point whose target is to be changed, *target* is a pair of (serviceID, storeID).

Security related questions

- How has right to modify the Catalog? The Catalog stores all of its data in the Hash, so the Catalog should have right to modify data in the Hash. The Catalog is used by the Storage Managers, so all Storage Managers have to have full control of the Catalog data.
- Each entry has an owner and ACL, should the Catalog only store these information or somehow act upon them? If it only stores the information then it is up to the Storage Manager not to allow someone the change e.g. an entry in a collection who has got no proper rights. If the Catalog acts upon these rights, then an entry can only be modified by its owner and the ones who has proper rights in the ACL of the entry, but in this case because a change is always made by a Storage Manager, this Storage Manager has to act on behalf of the user. How can this be accomplished?

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 register themselves by a number of Storage Maintainer services, and periodically send heartbeat to them. 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 Maintainer services about the error referring the file with its GUID.
The Storage Element organize its files into *stores* and within a store a file could be identified with a *referenceID* which is unique within the store. The Storage Elements advertise itself with giving the IDs of their stores. There are two kinds of stores: hierarchical and flat. A flat store uses referenceID from a flat namespace (e.g. UUIDs such as '31F831E2-CA38-4950-91DF-A89C6C40868D'), and usually you cannot specify the referenceID upon uploading, the Storage Element generates it itself. There will be a native flat store implementation within the Storage Element.
A hierarchical store uses a hierarchical filesystem-like namespace in the referenceIDs, and you should specify this upon uploading. There will be backends in the Storage Element for several third-party storage systems such as plain FTP, GridFTP, dCache, etc. If e.g. a plain FTP is wrapped by a Storage Element it could use a hierarchical store to expose it, using FTP paths as referenceIDs.
If we know the Location of a file, which is the ID of the Storage Element service, the ID of the store within the Storage Element, and the referenceID, we could from some information system get the endpoint reference of the Storage Element, then we should call its *get* method with the storeID and 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 which store we want use, (in case of hierarchical store the referenceID is needed), the size and checksum of the file and its GUID as well. We should provide owner and ACL information too, but see security related questions about this service. 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 in case of a flat store 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.
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 the Storage Manager initiate the putting of the new replica on a Storage Element and receives a TURL, then the Storage Manager asks a Storage Element holding one of the old replicas to upload that replica to the given TURL with the *copy* method. With *getState* we can get the state of a replica ('creating', 'alive' or 'invalid'). The *delete* method removes a file.

Data model

A file in a storage element is referenced by a *storeID* and a *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.
It is up to the Storage Element how to organize its stores internally, but there are some scenarios.

- The Storage Element uses only one store, e.g. with ID '0', and advertising itself only with this store. At the beginning it is empty, and if a put request arrives with storeID = 0, the Storage Element generates a new referenceID for the new file, which is later can be retrieved with this storeID and referenceID.
- The Storage Element has an internal namespace with files already in it, there is a file with path e.g. '/dir/file'. The Storage Element exposes this with some storeID, e.g. '1', and uses the referenceID as path. If a get request arrives with storeID = 1, and referenceID = '/dir/file' then the Storage Element provides this file for downloading. If there is a put request for this store, then a referenceID has to be given, e.g. '/dir/file2'. This kind of store should be mounted to the global namespace with a *newMountPoint* call of a Storage Manager.
- A Storage Element can provide both kind of stores, even multiple of each. There could be e.g. two separate namespace-based stores with ID '1000' and '2000', and one flat store with ID '3000'.

Interface

- **get**(getRequestList): returns list of (requestID, TURL, protocol, checksum)
  getRequestList is a list of (requestID, storeID, referenceID, protocols) where *requestID* is an arbitrary ID used in the reply; *storeID* and *referenceID* refers to the replica to get, *protocols* is a list of protocols the client able to use. The *TURL* in the response is a URL called Transfer URL which can be used by the client to download the file; the TURL usually contains the protocol, but just in case the chosen *protocol* is also returned along with the *checksum* of the replica.
- **put**(putRequestList): returns a list of (requestID, TURL, protocol, referenceID)
  putRequestList is a list of (requestID, GUID, checksum, size, storeID, referenceID, owner, acl, protocols) where *requestID* is a ID used for response, *GUID* and *checksum* is the GUID and checksum of the file, this is needed for a better self-healing, *size* is the size of the file in byte, *storeID* is the store we want to put the file into, *referenceID* is a proposed ID for the replica, the Storage Element is allowed the ignore it and generate a different ID, *owner* is the

ID of the owner of this file, *acl* is a list of (*ID*, *right*), where *ID* is the ID of the user, and *right* could be ‘read’, ‘write’ and ‘delete’.

- **delete**(deleteRequestList): returns a list of (requestID, status)
  deleteRequestList is a list of (requestID, storeID, referenceID) selecting the files to remove. The *status* could be ‘deleted’ or ‘nosuchfile’.
- **copy**(copyRequestList): returns a list of (requestID, status)
  copyRequestList is a list of (requestID, storeID, referenceID, TURL, protocol) where *storeID* and *referenceID* select the file which should be uploaded to *TURL* which is a URL with the given *protocol*.
- **getState**(stateRequestList): returns a list of (requestID, state)
  stateRequestList is a list of (requestID, storeID, referenceID) where *storeID* and *referenceID* points to the file whose state we want to get. The *state* in the response could be ‘creating’, ‘alive’ or ‘invalid’.

Security related questions

- Should the Storage Element store owner and ACL information for each replica? Is this exactly the same as in the Catalog? Or each replica of a file could have different owner and ACL on different Storage Elements? If the Storage Element stores no owner and ACL information how can it authorize a client who want to download or upload a file?
- When a user wants to download a file, connect a Storage Manager. The Storage Manager can decide based upon information from the catalog if this user has read rights to this file or not. If the user has proper rights, the Storage Manager chooses a Storage Element and initiate a get request. In this case the Storage Manager acts on its own behalf when communicating with the Storage Element or acts in behalf of the user? Who creates the security assertion? If the Storage Manager creates some kind of assertion which authorize the user to download the file from the given Storage Element, then the Storage Element should accept this assertion. Or should the Storage Element create this assertion using its on owner and ACL information or just getting the ID of the user from the Storage Manager?
- When we want to copy a replica from one Storage Element to an other one, we initiate the put request on the target Storage Element. Should we initiate it on behalf of the source Storage Element? Then we call the copy method of the source Storage Element. Is there a need for an assertion between Storage Elements? Who can the target Storage Element authorize the source Storage Element?
- If the Storage Element wraps a third-party storage how can we map the owners and ACLs of the third-party storage and the ARC Storage? How can we create assertions acceptable by the third-party storage? How can we create one-time TURLs with third-party storages which ensures that after the client downloads or uploads the requested file no one can use that TURL again to access that file?

Maintainer

Functionality

The Maintainer has two main functionality, the first is to receive heartbeat messages with list of invalid replicas from registered Storage Elements, the second is to manage the creation and deletion of replicas and making decisions about which Storage Element to choose for storing a file. The Maintainer services store all data in the Hash so it does not matter by which Maintainer service a Storage Element registered itself and to which Maintainer service the heartbeat messages are sent

to, all Maintainer services access the same database. If a Storage Element reports that one of its file is corrupted, then the Maintainer change the state of that particular replica to invalid and insert the file into the waiting queue to create a new replica of it. If a Storage Element stops sending heartbeats the Maintainer check the Hash for the Storage Element to GUID mappings and set the state of all replicas in the Catalog to offline, and put all the affected files to the waiting queue.
When a file has not enough replica or has a replica marked for deletion the Maintainer service can handle the replication and deletion if we call the *assist* method with the GUID of the file.
The *register* method should be used for registering a new Storage Element, and then periodically the *report* method should be called to indicate that the Storage Element is still alive and to report on the invalid replicas. The *choose* method is called by the Storage Manager to choose an appropriate Storage Element to store a replica.

Data model

The Maintainer keeps track of the registered Storage Elements. It stores (using the Hash as database) attributes such as:
- serviceID: the ID of the Storage Element
- lastHeartBeat: the timestamp of the last heartbeat sent by the Storage Element
- list of (GUID, storeID, referenceID) for each replica stored on this Storage Element: this list is maintained by the Catalog when a new file is created or locations change.

The Maintainer also maintains a waiting queue of files with something to do:
- GUID: the GUID of the file
- actionList: a list of (timestamp, action, location, maintainerID), where timestamp is the time of the action, action can be ‘replicating’ or ‘deleting’, location is a (serviceID, storeID, referenceID) refers to a replica of the file and maintainerID is the ID of the Maintainer creating this action-entry.

Interface

- **register**(serviceID, filelist): returns number of seconds, which is the time within the Maintainer expects the first heartbeat
  filelist is a list of (GUID, storeID, referenceID) including all the files the Storage Element already has
- **report**(serviceID, changedFiles): returns number of seconds
  changedFiles is a list of (GUID, storeID, 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).
- **assist**(list of GUIDs)
  add files to the Maintainer which it has to do something with (create or remove replicas)
- **choose**(chooseRequestList): returns a list of (requestID, serviceID, storeID)
  chooseRequestList is a list of (requestID, metadata) where *metadata* may contain information about the preferred Storage Elements and any other data which can be used to make the decision. The *serviceID* is the ID of the chosen Storage Element and the *storeID* is the ID of the chosen store within the Storage Element

Security related questions

- The Maintainer has to write the Hash, has to use the Catalog for modifying replica states and adding new replicas for any file, it has to call copy methods of any Storage Element to create new

replicas. How could all these services authorize a Maintainer service? When creating a new replica should the Maintainer acts behalf of the owner of a file? The replication is not initiated by any user, is this affects the creation of the assertion?

Manager

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, close, remove and list collections with *makeCollection*, *closeCollection*, *unmakeCollection* and *list.* The owner, the ACL and all the metadata of a file or collection can be changed with *modifyOwner*, *modifyACL* and *modifyMetadata.* The number of the needed replicas of a file can be changed with *modifyNeededReplicas*, 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.*

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.

Interface

- **putFile**(putFileRequestList): returns a list of (requestID, TURL, protocol)
  putFileRequestList is a list of (requestID, LN, size, checksum, protocols, neededReplicas, preferredSEs, owner, acl, metadata, reput), where *requestID* is an arbitrary ID used in the response; *LN* is the wanted Logical Name of the new file, *size* is the size of the file in bytes, *checksum* is some kind of checksum of the file (should be self-describing to know what kind of checksum it is), *protocols* is a list of protocols we want to use for uploading, *neededReplicas* is the number of how many replicas of this file we want, *preferredSEs* is a list of the IDs of the Storage Elements we want to use, *owner* is the ID of the owner of this file, *acl* is a list of (ID, right) where *ID* is the ID of the user and *right* can be 'read', 'write' or 'delete', *metadata* is a list of arbitrary attribute-value pairs we want to store, *reput* is a boolean indicating if it is a reput of an existing file. A reput removes all the existing replicas of the file and initiates a new upload, but preserves the GUID, so from the Catalog point of view it remains the same file. The returned *TURL* is a URL with a chosen *protocol* to upload the file itself.
- **getFile**(getFileRequestList): returns a list of (requestID, TURL, protocol)
  getFileRequestList is a list of (requestID, LN, protocols, preferredSEs) 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, *preferredSEs* is a list of IDs of the preferred Storage Elements, so if the file has a replica on one of our preferred Storage Element we get a TURL to that one. *TURL* is the transfer URL using which we can download the file.
- **delFile**(delFileRequestList): returns a list of (requestID, status)
  delRequestList 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**(statRequestList): returns a statResponseList
  the *statRequestList* is a list of (requestID, LN) with the Logical Name of the file or collection we want to get information about
  the *statResponseList* is a list of (requestID, type, owner, acl, metadata, size, checksum, entrylist, mutable) where type is 'collection' or 'file' and the other attributes are filled accordingly.
- **modifyOwner**(ownerRequestList): returns a list of (requestID, status)
  ownerRequestList is a list of (requestID, LN, owner) where *LN* is the Logical Name of the file or collection, and *owner* is the ID of the new owner. The *status* in the response could be 'modified' or 'denied' or 'nosuchLN'.
- **modifyACL**(ACLRequestList): returns a list of (requestID, status)
  ACLRequestList is a list of (requestID, LN, ID, ACLChangeType, right) where *LN* is the Logical Name of the file or collection, *ID* is the ID of the user whose rights we want to modify, *ACLChangeType* could be 'add' or 'remove', and *right* could be one of the rights of collection or file, but of course only the appropriate rights could be applied.
  The *status* in the response could be 'done', 'denied', 'invalid' (if that right is not applicable), 'nosuchLN'.
- **modifyMetadata**(metadataRequestList): returns a list of (requestID, status)
  metadataRequestList is a list of (requestID, LN, attribute, AVChangeType, value) where *AVChangeType* can be 'add' (add a new *attribute*-*value* pair), 'remove' (remove the *attribute*-*value*
  pair), 'delete' (remove all occurrences of *attribute*, no *value* needed for this change).
  The *status* in the response could be 'done', 'denied', 'nosuchentry' (if a remove cannot find a metadata with the given *attribute* and *value*, or if a delete cannot find an entry with the given *attribute*) or 'nosuchLN'.
- **modifyNeededReplicas**(replicaRequestList): returns a list of (requestID, status)
  replicaRequestList is a list of (requestID, LN, neededReplicas) which gives for each file referred by a Logical Name the new number of needed replicas. The *status* could be 'changed', 'denied', 'nosuchLN'.
- **makeCollection**(makeCollectionRequestList): returns a list of (requestID, status)
  makeCollectionRequestList is a list of (requestID, entrylist, mutable, owner, acl, metadata) where *entrylist* is the list of (name, LN) pairs, which is the initial content of this collection (the entries in the new collection will be hardlinks to the given Logical Names with the given *name*) ; if *mutable* is false then no more files can be added later: the collection is closed; *owner* is the ID of the owner of this collection, *acl* is a list of (ID, right) pairs, where *ID* is the ID of a user and *right* could be 'list', 'delete', 'add', 'modify' (see the data model of the Catalog).
  The *status* of response is 'made' or 'denied' or 'nosuchLN' (if the parent collection does not exist)
- **closeCollection**(closeRequestList): returns a list of (requestID, status)
  closeRequestList is a list of (requestID, LN) listing the Logical Names of the collections we want to close. The *status* in the response can be 'closed', 'denied', 'nosuchLN'.
- **unmakeCollection**(unmakeRequestList): returns a list of (requestID, status)
  unmakeRequestList is a list of (requestID, LN) with all the Logical Names of the collections we want to remove. The *status* could be 'unmade' or 'denied' or 'nosuchLN'.
- **list**(listRequestList): returns listResponseList

listRequestList is a list of (requestID, LN, neededMetadata) where *LN* is the Logical Name of the collection (or file) we want to list, *neededMetadata* is a list of attributes, only metadata with these attributes will be returned.
listResponseList is a list of (requestID, name, type, created, modified, owner, size, validReplicas, mutable, metadata) where *name* is the name of the entry within the collection, *type* is 'collection', 'file' or 'mountpoint', *created* and *modified* are timestamps, *owner* is the ID of the owner, *size* is the size of a file (a collection has no size), *validReplicas* is the number of valid replicas of a file, *mutable* indicates whether the collection is closed or not (a file has no mutable flag), *metadata* is a list of (attribute, value) pairs.

- **move**(moveRequestList): returns a list of (requestID, status, statusdetail)
  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* is to be 'moved', 'denied' or 'nosuchLN' and *statusdetail* could be 'source' or 'target' depending which part of the moving was denied or missing.
- **copy**(copyRequestList): returns a list of (requestID, status)
  copyRequestList is a list of (requestID, sourceLN, targetLN, preferredSEs, neededReplicas) where *sourceLN* is the path of the file we want to duplicate, the *targetLN* is the new path, *preferredSEs* is a list of the IDs of the Storage Elements we prefer to put the replicas on, *neededReplicas* is the number of needed replicas for the new file.
- **glob**(globRequestList): returns a list of (requestID, LNs)
  globRequestList 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.

Security related questions

- How can the Storage Manager create an assertion which authorizes the user to upload or download something from a Storage Element?
- How can we authorize the Storage Manager to get full access of all the data in the Catalog?