Issues with the SRM v1.1 interface specification

Experience from the CASTOR implementation

1 Introduction

The Storage Resource Manager, SRM, v1.0 interface specification [1] is now implemented for several storage systems running at various sites. The so called SRM v1.1 definition differs to v1.0 only with the additional 'copy' method, which allows for SRM controlled inter-site transfers initiated by a client. The SRM interface addresses fundamental functionality needed by grid tools for interactive with local site storage. The SRM interface definition itself has a number of interesting design concepts that allows for building fault-tolerant and robust systems, in particular

- Transfer protocol negotiation
- Asynchronous operations
- Multi-file requests

With the SRM v1.1 'copy' extension, the specification does probably already contain most of the functionality necessary for building an automated system for 'broadcasting' LHC raw data from CERN towards the Tier-1 institutes.

The SRM v2.1 specification [2], attempts to address some of the shortcomings in v1.0. It also extends the specification to include more user oriented functionality such as listing directories and space reservations.

This report summarises the interoperability issues found with the SRM v1.1 specification from the experience gained with the CERN CASTOR implementation. Some of those points are already addressed in SRM v2.1, see Table 1.

Issue	**Section (pg)**	**Fixed in v2.1**
Lack of enumeration	2.1 (2)	Yes
Request and file state lifecycle	2.2 (2)	No
Request history	2.3 (2)	No
Uniqueness of the request identifier	2.4 (3)	Yes
SURL (Site URL) semantics	2.5 (3)	No
Exception handling and error propagation	2.6 (3)	Partially
Pin lifetime	2.7 (4)	Yes
AdvisoryDelete != Delete	2.8 (4)	Yes

Table 1: summary of v1.1 issues discussed in this report

However, because of its very rich and complex method and data type sets the SRM v2.1 is likely to bring along much more substantial interoperability issues than v1.1. For this reason it was proposed at GGF10 in Berlin that before rushing towards finalising v2.1 the experience and interoperability issues with v1.1 must be at least documented. The present document is the CERN draft contribution to that process.

2 Detailed descriptions of issues

This section provides a detailed description of the SRM v1.1 specification issues listed in Table 1.

2.1 Lack of enumeration

The SRM v1.0 data structure specification contains three enumeration-like types implemented with string values. The following values are given in the specification:

- RequestStatus types: "Get", "Put", "Copy", …
- RequestStatus states: "Pending", "Active", "Done", "Failed"
- RequestFileStatus: "Pending", "Ready", "Running", "Done", "Failed"

The specification is not explicit as to whether the implementation should exactly match those values or if, for instance, the match is case-insensitive or if it is enough to match the first character. An additional confusion is the slight difference between the RequestStatus states and RequestFileStatus states. One may argue that "Active" and "Running" have identical meaning and why RequestStatus lacks the "Ready" state.

Although this is a minor issue that can easily be negotiated between developers of the SRM implementations, it is an unnecessary hole in a specification that based on request lifecycles.

The SRM v2.1 specification uses true enumeration types.

2.2 Request and file state lifecycles

The request lifecycle for "Put" and "Get" types of requests is well described in the v1.0 specification. However, the lifecycle for "Copy" request, which belongs to the v1.1 of the interface, is not specified. The "Copy" request is significantly different from "Put" and "Get" in that for the latter two the client user is responsible for closing the request once file processing has finished by updating the RequestFileStatus state to "Done". The "Copy" request on the other hand, is executed without user interaction. This lead one implementation (CERN) to use the RequestStatus and RequestFileStatus states serve as progress indicators, where the RequestFileStatus state became "Done" for the files successfully transferred. Another implementation (FNAL) decided to put the RequestFileStatus state to "Ready" when the file had been successfully transferred and it was still up to the client user to set the status to "Done" although it had no other significance than allowing the SRM to forget about the request.

2.3 Request history

It is unspecified in the SRM v1.0 definition what the SRM should do with requests that have reached the "Done" or "Failed" status. One could argue that when a request reaches "Done" status, it is eligible for removal since the client has declared to have finished with the request. It should be noted, however, that there is no method provided for the client to set the status of the whole request. The 'setFileStatus' method is provided for updating the state of the individual file sub-requests. The assumption is that when all file sub-requests have reached "Done" status, the whole request is automatically set to "Done". If the request is removed when it reaches "Done" status there is therefore a risk that the

client never detects the final status before the SRM forgets about the request. The implicit assumption is then that if a request suddenly disappears from a SRM it means that it has finished successfully.

A second option is that the SRM keeps a history of all requests it has executed. In that case the client can continue to query for requests that have reached "Done" status, at least for as long as the SRM is designed to retain the history records.

A third alternative is that the SRM keeps the request in "Done" status until the client has queried for the status at least once.

The lack of specification means that clients must be prepared to handle all cases.

Although it is not explicitly stated in the specification, it is logical that "Failed" request must be kept until the client has queried for the status at least once.

2.4 Uniqueness of the request identifier (token)

Related with the request history issue is the question whether or not a SRM can re-use a request identifier when it forgets about a finished request. The SRM v1.0 specification states that the request identifier must be unique but it is unclear if the uniqueness is only required in a given moment of time or if it must remain over time. If it re-uses the request identifier there is a potential risk that the client of the previous request may be confused if it queries for the request after it has finished.

In SRM v2.1 the requestToken is defined to be unique and immutable (non-reusable).

2.5 SURL (Site URL) semantics

The SURL is a useful invention from the SRM v1 definition. It provides a transfer and location neutral unique reference to a file. When accessing or transferring a file, a client presents the SRM with the file's SURL together with a list of protocols known to the client. When the file is ready for access the SRM returns the Transfer URL (TURL), which is a, possibly temporary, location and transfer specific reference to the file.

Whether or not the site specific part of the SURL should follow some specific semantics has been up for discussion for some time. Some of the SRM group members argue that the SURL is an URI and hence without any general semantics that, for instance, would reflect a directory hierarchy. Others argue that the SURL is an URL and hence supports the notion of relative and absolute paths and delimiters reserved for building a directory hierarchy. It is unclear what the conclusion from this discussion will be but it should be noted that the v2.x specification already contains methods for operating on directories (e.g. srmLs, srmRmdir), and thus at least implicitly assume there is a general convention for specifying a directory hierarchy in the site specific part of the SURL.

2.6 Exception handling and error propagation

In the SRM v1.0 it is unspecified if a multi-file request should fail in case of the file sub-requests fails. If the request should fail, it is also unspecified if the execution should stop

immediately or if the other file sub-requests are allowed to finish before the whole request is flagged as failed. The specification only contains one error message field in the main request status structure so there is no defined mechanism to report several individual file failures. The only way to provide a consistent exception handling is therefore to immediately fail the request as soon as one of the file sub-requests fails and report the failure reason for that file in the error message field of the main request status structure. However, since the interruption of a running multi-file SRM request may imply significant waste of resources most SRM implementation have chosen to continue the request until all file sub-requests have reached "Done" or "Failed" status. The implementation may choose that the full request is successful if at least one file sub-request is successful. Other implementations may choose to fail the request if at least one file sub-request fails. The error message field in the main request status structure may then contain the reason for the last failed file sub-request. In either case the client exception handling becomes impossible since it is undetermined whether or not it is useful to retry the request.

The error message field in the main request status is used for passing the implementation specific reason for a failure. There are no format specifications or other convention for how the message should be parsed implying that its sole purpose is to be printed for human interpretation.

2.7 Pin lifetime

In the SRM v1.0 specification the 'pin' expiration time (or lifetime) is entirely determined by site policy and cannot be specified with the request. While this is pragmatic and probably the only way the pin concept can be handled, there is no way for the client to know the pin lifetime for any particular file in the disk cache. If the policy is static (e.g. does not change with file type or client group membership) it can be published as an attribute in the StorageElement schema. However, even if the policy is known, there is no way to query the remaining pin lifetime of a particular file because there are no time related attributes in the RequestFileStatus and FileMetaData properties. The 'pin' method is therefore of none or very little use to clients who wish to assure that a file remains resident in cache for a given period of time.

Leaving the pin expiration time to be entirely decided by site policy makes the 'pin' method even less useful since the policy definition may be completely arbitrary. For instance, the CASTOR SRM pin policy is "file is garbage collected when space is needed".

While this is not an argument against pinning it highlights the lack of means for passing information about pinned files. In SRM v2.x this is addressed by the inclusion of a 'lifetime' field in T(Get/Put/Replicate)FileRequest structures and a corresponding remainingPinTimeIfAny field in the T(Get/Put/Replicate)RequestFilestatus structures.

2.8 AdvisoryDelete != Delete

The SRM v1.0 specification contains an optional method 'AdvisoryDelete'. Its purpose is said to be "This operation is a recommendation to the SRM that the client is done with

the file, does not expect to use it again, and that the file can be scheduled for removal before others. The SRM implementation may choose to ignore this method.". This definition does not state whether the delete only concerns a disk resident copy or the HSM file in case the SRM is interfaced with an archive. The fact that the deletion is advisory also implies that the SRM is free to simply ignore the request without returning an error. Thus, the client cannot know if the file is ever going to be deleted and if so, when that is going to happen.

3 Comments on SRM v1.1 specification process

This section comments on specification process itself.

3.1 Lack of reference implementation

Implementation details that are not covered by the specification lead to divergence in the response or behaviour between SRM implementations and the client must know which SRM implementation it is talking to. While this can possibly be dealt with at client level, the SRM specification also includes functionality that requires interoperability (specifically the 'copy' method) between different SRM implementation. It is therefore particularly important to be able to certify an SRM implementation against a reference implementation before the service is opened for public use. A reference implementation should include at least the functionality required for the interoperability between SRMs.

A reference implementation can be provided in two ways:

- A permanent (well-known) service instance managed by one of the SRM working group partners
- A packaged SRM for self-install and configuration for those who wish to certify their implementations

It is recognised that the SRM working group is a loose collaboration and specification work is mostly provided through spare efforts provided by its different partners. While a reference implementation could possibly be drawn from the set of the first ready SRM implementations, it will require a substantial effort to manage and support it.

4 References

[1] SRM v1.0 specification
[2] SRM v2.1 specification