SRM Functionality And Interface Design
SRM Version 2.1

Document written by: Arie Shoshani

This document summarizes the discussions and conclusions of a 2-day meeting whose purpose was to further define the functionality and standardize the interface of Storage Resource Managers (SRMs) – a Grid middleware component. The meeting took place at CERN on December 4-5, 2002. This document is a follow up to the basic SRM design consideration document that describes the basic functionality of SRM Version 2.0 (see "SRM.v2.0.joint.func.design.rev2.doc" posted at http://sdm.lbl.gov/srm). In this meeting we focused on additional functionality, especially in the area of dynamic storage space reservation and directory functionality in client-acquired storage spaces.

Participants:

EDG-WP2: Peter Kunszt, Heinz Stockinger, Kurt Stockinger, Erwin Laure
EDG-WP5: Jean-Philippe Baud, Stefano Occhetti, Jens Jensen, Emil Knezo, owen synge
JLAB: Bryan Hess, Andy Kowalski
FermiLab: Don Petravick, Timur Perelmutov
LBNL: Arie Shoshani, Alex Sim
Other contributors not at the meeting: Chip Watson (Jlab), Rich Wellner (FermiLab), Junmin Gu (LBNL)

1. Details of issues and considerations

This section describes the options and design considerations for the decisions made jointly by the people who attended the meeting at CERN. A summary of the decisions is given at the end of this document. All the recommendations discussed here are reflected in the "SRM.v2.1.methods.rev0.doc" that has the specific interfaces defined.

• Issue: terminology

The terminology used seem a minor issue, but the following seem to be important enough to discuss and agree on.

a) It was suggested that the term "srm" will be dropped from all function calls (methods). For example, instead of "srmCopy" or "srmReleaseFiles", simply have "copy" or "releaseFiles". The reason for that is that if we use WSDL to define the interface, then we can use a common class name "srm". The counter argument was that WSDL may not be the only interface (for example, some SRMs may provide a C++ based interface), and in this case it is wise to keep the "srm" prefix to identify these methods as SRM methods.

Recommendation: keep the "srm" in the methods names

b) Another terminology issue was the use of "srmGet" and "srmPut". Actually, these SRM commands to not move files at all, but rather perform the space allocation and/or pinning in preparation to have the client either "get" a file from SRM's managed space or "put" a file into SRM's managed space. Specifically, "srmGet" is intended to pin a file if the SRM already has the file; otherwise the SRM will allocate space, copy a file from its archive or a remote location, and pin the file. In the case of "srmPut", the SRM only allocates space.

Recommendation: use the names of "srmPrepareToGet" and "srmPrepareToPut" instead. However, we will continue to use "srmCopy", since it is expected that a file will actually be copied by the SRM calling some FTP service to copy the file.

c) LFN vs. GFN. It was pointed out that "logical file names" (LFN) are not considered by all as being immutable (used once name). There is a concern that over time this will cause difficulties. It was also agreed that immutable file names should be used with SRMs. The term "global file name" (GFN) was suggested for this purpose.

Recommendation: use the term GFN in the methods parameters.

• Issue: what type of spaces should be possible to reserve?

Reserving permanent space is obviously needed for dynamic archival storage reservation. Similarly, reserving volatile space is necessary for shared disk resources that are not intended to be used for long-term archival storage. However, support for reservation of a durable space requires explanation as to its value.

Recall that "durable files" are files that have a lifetime associated with them, but the SRM cannot remove them without some explicit action. The action can be sending a notification to the file owner or to an administrator, or some automatic action such as archiving the file before notifying the owner. Durable files are especially useful when a client generates a large number of files that need to be stored temporarily in some shared disk cache before being archived. The purpose of durable space is similar: reserving a guaranteed space on a temporary basis that cannot be removed without an explicit action. Durable space can be "released" when the client is finished using it. Durable space is particularly useful for request planning. For example, durable space can be required when running a large simulation as the space to "dump" the generated files into. The files in that space can then be moved to an archive, and when this action is completed the durable space is released.

In contrast to durable space which is a space that cannot be removed without explicit action, volatile space can be reclaimed by the SRM if space is needed. For example, suppose that a client asks for 200 GB volatile space when there are very few clients using the SRM. The SRM can initially allocate this space, but when many new clients request space the SRM can reduce the space used by the first client to say, 50 GBs, provided that there are no "pinned" files in that space. Thus, volatile space reservation is considered a

"best-effort" space that can be reclaimed by the SRM if the need arises. (Note: the term "best-effort" is common in the context of making reservations, and is used by the NeST project as well [http://www.cs.wisc.edu/condor/nest/]).

We also discussed the issue that some SRMs may not want to pre-allocate durable and/or permanent space on a "guaranteed" basis. Rather, the space is assigned only when files are put into that space. Thus, the concept of a "best-effort" reservation can be used with durable space as well. However, once the file is put into a durable space, the file (and the space for it) cannot be removed without explicit action even if the lifetime expires. The following table summarizes the discussion above.

Spaces / Features	**volatile**	**durable**	**permanent**
Lifetime applies	Yes	Yes	No
Can be reclaimed by SRM?	Yes	No	No
Best-effort option	Always	Yes	Yes

Recommendation: permit space reservation of all three types: volatile, durable, and permanent. All SRMs are expected to support volatile space, which is the default if no space-type is specified. SRMs are free to have their own policies regarding support of space-types, space-size, and space-lifetime. "Best-effort" and "guaranteed" is a policy choice of each SRM, and can be applied to durable and permanent space types. Volatile space is by definition "best-effort".

Size of space and lifetime for all spaces are negotiable. No conformation is required – instead clients can "release space" if they do not like the lifetime assigned.

• Issue: how many spaces of each type should be supported?

Our first inclination was to allow as many spaces of a certain type as the client wishes. An example where this could be useful is as follows. Suppose that a client runs two simulations concurrently and wishes the output of each to go to a separate durable space. The alternative is to have a single durable space, and to store the files from each simulation into separate sub-directories.

Supporting multiple spaces per type requires that a name for each space is assigned. This brings up the following question: should the name be assigned by the user or by the SRM, or both. Whatever solution we choose, this adds to the complexity of the SRM interfaces and implementation. We concluded that the benefit of supporting multiple

spaces per type per user is not worth the additional complexity in the design and implementation of SRMs. Having a single space per type leaves it up to the client on how to utilize the space efficiently for all of their activities.

Recommendation: support a single space per type. This implies that a client can request to increase (or decrease) the space previously acquired as well as request a longer (or shorter) lifetime. We also decided to add the functionality of requesting that a space will be "compacted", i.e. reduced to the size of the files currently in the space.

• Issue: charging for storage space usage

The main concern for SRMs is that shared storage space is used efficiently. Thus, there needs to be some incentive for client to use space as sparingly as possible, claiming and releasing space on as-needed basis. The usual incentive is some currency. The most obvious model is to charges the client according to their space usage in real currency. However, this model is not always feasible. An alternative is to allocate a quota to each client (e.g. megabyte-hours) and let the client work within this limits.

Recommendation: we agreed, in principle, that the currency/quota policies and its enforcement should not be dealt with by SRMs. Rather, in the grid, this should be the concern of the "virtual organization" (VO). It is not clear at this time which component should manage currency/quota policies, but it seems that it could be an additional task of the "community authorization service" (CAS), or a separate "currency/quota service" (CQS). The function of SRMs will therefore be limited to keeping track of storage usage by clients, and reporting this usage to the service that grants and enforces currency/quota policies. The granting of storage space to a client will be based on some "authenticated ticket" provided to the client by the CQS.

However, until such time as the CQS becomes a reality, we need to have default policies. Thus, we decided to permit reservation requests for space, but since space cannot be charged at this time, each SRM can choose its own method of limiting usage by enforcing their own quota policies. Under this method, the main incentive for a client to release space as soon as possible, is that the released space can be re-used by the client given the quota limitation. For example, if the quota for volatile space is 20 GBs per user, then after the 20 GBs are filled no more files can be brought into this space until some other files are released.

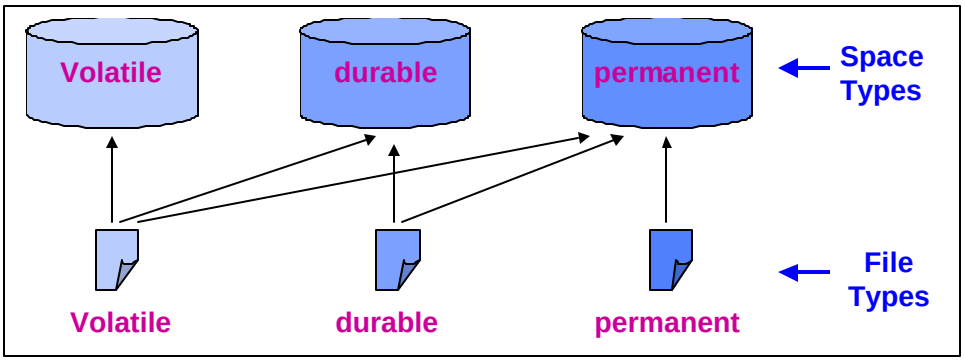
• Issue: assigning files to space types

Since SRMs can support 3 file types: volatile, durable, and permanent, as well as the corresponding storage space types, the natural question is whether a file of a certain type should be restricted to reside only in the corresponding space-types?

We concluded that there is no reason that, for example, a volatile file could not be assigned to a durable or permanent space. When a client makes such a request, the SRM will assign the volatile file to the client's private space, but will also provide the service of "cleaning up" and releasing the file when its lifetime expires. A similar argument can be made for allowing durable files to be assigned to a permanent space that a client owns.

Recommendation: permit assignment of:
a) Volatile files to volatile, durable, or permanent spaces.
b) Durable files to durable and permanent spaces only.
c) Permanent files to permanent spaces only.

This is shown schematically in the following figure.



• Issue: access control to files

We restricted our discussion to read-only file. The issue of files that can be written into or modified is a complex one in that there needs to be a "master-copy" concept, and the synchronization of replicas with the master-copy. We decided to avoid such issues at this time and concentrate only on access permissions to read-only files.

Suppose that a client A writes a file F into an archival space S(A) that belongs to him/her, and wants to allow another client B to read this file. This requires that the file F would be assigned a "read-group-permission" for client B. Thus, the SRM should be able to support dynamic "group permissions" requests. This is not usually supported by file systems. For example, a "unix" type file system requires that groups are assigned ahead of time. It was suggested that this capability will be supported by SRMs using "access control lists" (ACLs).

The second issue implied by the scenario above is which component checks the identity of client B when he/she wants to read the file. In many systems, security if enforced by the local file system using some security model (such as GSI, Kerboros, or SSL). Given that GSI is the common mechanism for the grid, this implies some translation to the local system security model.

The third issue implied by the scenario above is how to deal with community authorization. Suppose that client B is not known to the system that manages S(A), then how would client A assign permission only to client B who is a member of the community? We concluded that this could be done using the client B's distinguished name (DN).

The fourth and most thorny issue implied by the scenario above is what happens after client B gets the file. One option is to preserve the access control of the original file. Another option is to let the Client B do whatever he/she wishes to do with that file, such as assign "read-world" permission to it. Although the first option sounds attractive, it requires the coordination of community permissions between SRMs, and agreements on standard universal access control mechanisms. We chose to follow the second option, which is consistent with file system policies: once a user grants permission to another user to read his file, the second user can do what he wishes with that file, including passing it on to others.

Recommendation: file access in durable and permanent spaces is determined by the space owners. When a file is initially moved into a client's space, the SRM will assign it a "read-user" permission only. The user can then request to add "read-group" or "read-world" permissions.

To allow file sharing that is controlled by the SRM, we recommend that the owner of volatile spaces is the SRM. This is consistent with the choice that SRMs can reclaim volatile space without explicit client's release of the space. However, access control of shared files is problematic. Rather than SRMs inheriting "access lists", we concluded that the SRM should always check for access permission from the source site (using the siteURL, i.e. SURL). If permission is granted for the requesting client, that client is then granted "read-group" permission. The exception to the above is for files that are originally "world readable" – in this case there is no need to check with the source site.

• Issue: request to assign files to another user

Suppose that a client A puts a collection of files in his/her (durable or permanent) space, and is finished using the files. Before removing the files, the client A wishes to assign the files to another client B. Client A has no right to ask SRM to moves the files to client's B space – this has to be done by client B. A useful service that an SRM can provide is to let client A notify SRM that it is OK to move the file collection to client's B space, and after this action, the space can be freed. We chose to support this capability by a "reassign-to-user" call that has a lifetime associated with it.

Recommendation: we chose to support this capability by a "srmReassignToUser" function call that has a lifetime associated with it. It is expected that after client A reassign all files under a directory to client B, client B will make a request to move this

entire directory to his/her space. The files in client A's space will be removed either when the files are moved to client B or when the lifetime expires.

Issue: how to charge space when files are shared?

When a file is in some space, should the file-space charged to each client sharing the file? Should we treat charging of space differently depending on the space type? For example, suppose that an SRM enforces a quota policy, and suppose that a file was brought into volatile space for client A, and client B wishes to access this file as well. Should the space be counted against client B's quota? A fair policy could charge each for half the space. Alternatively, each could be charged for the entire space, as if the file was not shared.

Recommendation: This is a question of policy, but as a general guideline we recommend that volatile space is charged to each client who shares a file. On the other hand, if a file is in a client's (durable or permanent) space, the space is not charged to a client that has permission to share the file. If the shared file is in durable space, the access lifetime of the file cannot exceed the lifetime of the durable space.

• Issue: directory support and semantics

Should SRMs support all the usual unix-like functionality for all spaces? Supporting such functionality can usually be built upon the underlying file system. However, when file sharing is supported, the implementation is more difficult. For example, in order to share a file between two or more users, the SRM have to store the file in a single physical location, and allow each user to be virtually linked to this file. Accordingly, permissions to access this file need to be dynamically added as well. One possible implementation of volatile shared files, is to make SRM the owner of the file, link each user's directory to shared files in the SRM space, and add group permission to each user that is allow to read it dynamically.

As mentioned above, to determine a user's permission to share a file, the SRM should check with the original source site (using SURL) and if permission is granted, the SRM can shared the file rather than transferring it again and replicating it.

Recommendation: We decided to support all directory functions for all the space types. Specifically, the following functions will be supported: srmMkdir, srmRmdir (applies to *dir*), srmRm (applies to *file*), srmLs (applies to both *dir* and *file*), srmMv (applies to both *dir* and *file* within spaces that belong to the same user), srmCp: (applies to both *dir* and *file* between spaces that can belong to different users), srmCd, srmPwd.

Note that srmCp leaves the original directory/file intact, and can be used to copy directory/files from a space belonging to one user to a space belonging to another user.

In contrast srmMv allows moving directory/files from between spaces that belong to the same user only.

• Issue: how to allow direct file I/O to SRM controlled files?

In many applications, it is useful to have a client's program perform direct file I/O operations, such as open, read, close the file. Before the direct I/O can be performed the program communicates with the SRM to get the desired file, which may involve getting it from an archive or from a remote location. The issue is how to communicate to the SRM that a direct I/O is intended to be used next by the client's program.

There are several methods of direct file I/O supported by different systems. For example rfio (used by the HENP community), dcap (used by the DESY experiment), and globus-xio. We concluded that these can be supported by the protocol negotiation mechanism we already have (see "SRM.v2.0.joint.func.design.rev2.doc" posted at http://sdm.lbl.gov/srm). It works as follows. An ordered list of one or more protocols can be provided in the srmPrepareToGet command, and the SRM will try to match to highest possible protocol and return the selected protocol in the "transfer URL".

Recommendation: Use the existing protocol negotiation method to request a direct file I/O protocol.

• Issue: revisit "srmCopy" in push mode

In the specification of SRM version 2.0, we decided to support only "srmCopy" in pull mode, that is requesting an SRM at site A (call it SRM-A) to contact another site, site B and copy files from that site. First SRM-A needs to allocate the space for the file. Then it needs to contact site B. Site B may or may not have an SRM. If it has an SRM (call it SRM-B), then the "srmPrepareToGet" will be sent from SRM-A to SRM-B, so that the SRM-B can pin the file. This will be followed by calling on the FTP service to "get" the file. If an SRM does not exist at site B, then SRM-A will simply call the FTP service.

An "srmCopy" in push mode does the opposite. SRM-A is now being asked to copy a file from its space to site B. SRM-A will then start by pinning the file. Then it will contact SRM-B (the target SRM) to allocate space by performing the "srmPrepareToPut" call. Then SRM-A will invoke an FTP service to "put" the file. Then SRM-A will release the file in its space. In case that site B does not have an SRM, SRM-A simply call the FTP service to "put" the file.

The reason that we felt it was important to have the "srmCopy" in push mode is that target site B may not have an SRM, in which case there is no way to perform the copy operation without the push mode.

Recommendation: support "srmCopy" in both push and pull modes. The modes are not explicit in the "srmCopy" call. Instead, the parameters identify the "fromSURL" and "toSURL". If the "fromSURL" is local it is interpreted as a push mode. Conversely, if the "toSURL" is local it is interpreted as a pull mode.

• Issue: combined copy and release function

One of the most common scenarios of using SRMs is the process of data production. Suppose that a client is scheduled to use some compute resources to run a large simulation. The results of the simulation have to be eventually archived, perhaps on a Mass Storage System. It is important that the client has a temporary disk cache to dump the results into, so that the computation is not slowed down because of the archiving process. For example, suppose that 100 GBs were dumped into some SRM disk cache. The client then wants to be able to schedule a transfer of the files from the SRM managing the disk cache (the sourceSRM) to an SRM that manages an archive (the targetSRM).

An srmCopy command can be used to "push" the files from the sourceSRM to the targetSRM, but it could be useful to have the files in the sourceSRM automatically released (or deleted) as soon as they move to the targetSRM. For this reason we considered whether to add a "boolean flag" in the copy command, called "releaseSourceFiles". If this flag is set to "true" the SRM will release the source files after they are copied to the targetSRM. The same "releaseSourceFiles" flag can be used if the client chooses to use the srmCopy command to "pull" the files into the archive. In this case the client communicates with the targetSRM asking it to copy the files from the sourceSRM. If the "releaseSourceFiles" flag is set to true, the targetSRM notifies the sourceSRM after each file is copied (by sending a srmRelease command to the sourceSRM). If the flag is not set, no srmRelease is issued after the file is copied.

Recommendation: Add the "releaseSourceFiles" to the srmCopy function call.

• Issue: should we have user ID in the SRM methods

If gsi is used, then the user ID is obtained from the gsi libraries in the form of a "distinguished name" (DN). This prompted us to consider dropping the user ID from all calls. Later, it was argued that not all systems use DNs internally, and that additional information about the user may need to be supplied (for example to log the user into the mass storage system that the SRM is associated with). For this reason, a user ID was added to all calls as an optional parameter, except for call where the "request-token" is used, such as "srmReleaseFiles", "srmAbort", or "srmGetFilesStatus".

Recommendation: include userID as an optional parameter in all appropriate methods.

Issue: granularity of space and time measures

Recommendation: for specifying *space*, we agreed that a MB is sufficient for keeping track of space allocated to users. It was also suggested that we use the binary value for a MB, i.e. 1024x1024 bytes.

Recommendation: for specifying *lifetime*, it was agreed that it should be specified as *relative time.* That is, a lifetime of 30 minutes returned to the client means: "30 min from now". To avoid synchronization complexities we decided that the SRM will compensate for the expected time that the response travels to the client by adding a minute or so when it keeps tracks of lifetime. To simplify the software implementation we agreed to measure lifetime in *seconds*.

Recommendation: for specifying *file creation time*, we agreed to use GMT. It is best if the SRM can synchronize with NTP or atomic clock.

Recommendation: for specifying space quotas, we agreed to use the units of MB-seconds, to be consistent with the space and lifetime measures.

2. Summary

The following topics were discussed and decisions made on what should be supported by SRMs.

2.1 Main functionality to be added to the SRM interface

• Space reservations - the ability to dynamically make a request for space, and negotiate its size. Should we support reservation for all 3 types of spaces: volatile, durable, and permanent?
Decision: support all three types, but each SRM is only required to support volatile space; permanent and durable spaces are optional.

• Directory functions – the ability to create directories in the space acquired by a client.
Decision: support all commonly used unix directory functions in all 3 types of spaces.

• Access control – the ability to assign permissions to access files by the system owner.
Decision: support this functionality consistent with original owner's access control assignments.

• Lifetime negotiations for space – the ability to negotiate the size and lifetime of an acquired space.
Decision: support this functionality by returning lifetime assigned. No conformation is required – instead client can "release space" if they do not like the lifetime assigned.

2.2 Other issues

• Meaning of lifetime – should lifetime be relative or absolute?

Decision: use relative lifetime for files and space. Support absolute time in GMT for time file was created, modified, etc.

• Renew lifetime – the ability to renew a lifetime of a file or a storage space.
Decision: support this capability. It is up to each SRM implementation on to limit such renewal requests.

• Revisit srmCopy (push mode) – should a user be allowed to specify the "push" transfer of a file to other locations?
Decision: yes, both "pull" and "push" will be allowed. The push-pull direction will be determined from the URL specified by the from-to parameters.

• Revisit call-backs – should we introduce call-backs through the WSDL "handle" reference capability.
Decision: not mandatory. In the future, we should consider "notification services" where call-backs can be posted.

3. minimal common functionality

After we discussed these issues, we agreed on a minimal common functionality that all participating groups will implement. Specifically, the following systems are targeted to have the new SRM interface and capabilities:
LBNL – an SRM to disk, an SRM to HPSS,
Jlab – an SRM to JasMINE,
Fermi – and SRM to Enstore,
EDG:WP5 – and SRM to Castor, as well as an SRM to the system being developed at RAL in England.

In addition, the people in EDG:WP2 will interface to various SRMs to get and store file to these systems.

The minimal functionality we agreed on includes:
• Space types
  — Volatile at minimum
  — Permanent, durable is optional
  — Durable: support can be for best effort or guaranteed, or both.
• Space reservations
  — Space reservation will be supported, but SRMs can choose to respond with a policy default.
•Directory support
  — Yes
• srmMv applied to directories
  — Yes. This capability will permit moving directories and all their content from one space to another, including spaces owned by different client
• Security
  — All SRM will support gsi over http

- Transfer protocols
  - GridFTP is the common protocol; others are allowed.
- srmChangeFileType
  - Provided that space for the requested type was acquired.
- srmCopy
  - Both push and pull will be supported, by specifying “from” and “to”.