Common Storage Resource Manager Operations

Ian Bird, Bryan Hess, Andy Kowalski
{Ian.Bird, bhess, kowalski}@jlab.org
Thomas Jefferson National Accelerator Facility

Don Petravick, Rich Wellner
{Petravick, wellner}@fnal.gov
Fermi National Accelerator Lab

Alex Sim, Arie Shoshani
{asim, shoshani }@lbl.gov
Lawrence Berkeley National Laboratory

October 22, 2001
Version 1.0

Scope and Sources

This document is a synthesis of discussions about a common set of Storage Resource Manager (SRM) operations. The SRM interface presents a combined view of a mass storage system's secondary and tertiary storage to wide-area and grid clients. We have adopted the nomenclature of Arie Shoshani (HRM, TRM, DRM) to describe the abstract structure of different types of storage systems, and SRM to refer generically to any of these types. His papers[1] (co-authored by Alex Sim and Junmin Gu) on SRM design and file replication formed an essential source. An initial set of SRM operations was circulated via email by Don Petravick and is another basis for this document. Discussions between Chip Watson, Ian Bird, Andy Kowalski, Ying Chen, and Bryan Hess also shaped the final form.

In creating this document we have drawn on the common ground established in previous writings, meetings, and conversations. We outline our assumptions and principles in this section and then go on to describe our goals and finally propose a set of operations for review by the collaboration.

In "Data Management in Data Grids"[2] Andrew Hanushevsky makes clear the need that data movement between grid agents and a local site's storage system must be done in such a way that internal priorities are not sacrificed. This principle informs our design. The operations detailed below can be implemented such that a storage system can maintain control of its own resources, but the client is always given accurate information as to the disposition of its requests.

There is some consensus that the tape resources and disk resources must be presented as a single element, never presented externally as having distinct tape and disk components. This abstraction is useful because it delegates control over file migration to the local site, and also encapsulates the details of the storage system. The protocol that we present below enforces this consolidation of the disk and tape components.

Another important point is that this interface deals with files as the smallest indivisible

unit of data movement. There is no provision to decompose files into objects or address the file contents in any way. Files in the SRM that we describe are immutable and always moved in an all-or-nothing basis.

We do not address the interaction with replica catalogs in great detail, although we have discussed allowing a replica catalog or other grid-agent register with the SRM so that it could be notified of file additions, deletions, and metadata changes. This notification could have some advisory-level feedback mechanism that would influence file deletion policy of the SRM. We stress that automatic updating of the replica catalogs for file replication and deletion in the grid sites is necessary for clients to make informed choices of where to get files from.

Finally, we are of the opinion that all interaction with the SRM must be client initiated. This addresses the practical problem of firewalls to some extent and also implies a model for resource reservation. The client must setup any advance resource allocation and these allocations have fixed times to live because the server cannot contact the client. The specific implication of client-initiated interaction is that, files to be gotten from SRM can only be pulled by the clients, and files to be written into SRM can only be pushed into SRM.

Protocols

There are three classes of network protocols to consider for this application: Data Movement protocols, request protocols, and security-related protocols. Data Movement protocols like GridFTP, BBFTP and the like are used for efficient bulk data transfer over wide area networks. The decision about which protocol to use can be made by the server. The client can provide a list protocols to the SRM for each request. This list tells the SRM the protocols that the client knows how to speak. This list is provided because an SRM may not be able to serve all protocols from all caches and it might therefore be required to cache differently based on the ways the client will be able to retrieve a file.

The request protocol needs to be expressive, simple, and not bound to any particular implementation. For this reason some groups (including Jefferson Lab) favor SOAP plus HTTPS.

There are two security implications for these protocols. First, we assume that there is some secure channel, perhaps TLS/SSL, which can protect the transmission of authentication requests. Secondly, there is a need to convey as a part of the request to the SRM the identity, authorization and capabilities of the requestor. Authority information should be kept on the SRM, else the SRM is beholden to some external representation of what permissions a user have. This violates the idea that the SRM be the master of its domain. This authority information is propagated from the secure transport layer to the SRM.

Some guiding principles

We have chosen to permit a request to consist of multiple files, rather than a single file per request. This choice was made for several reasons. It is a compact way to express a large number of file requests. More importantly, providing SRM with the entire request permits SRM to return the files cached in an order most useful to enhance the efficiency of the system. For example, an SRM that can access tape storage can order the files accessed in order to maximize the number of files read from the same tape, thus minimizing tape mounts. Similarly, SRM can order file access according to their popularity so as to serve as many clients as possible simultaneously.

We assume that the local site storage system can use any service policies it chooses. This includes the quotas of how many active files to have in the cache per user at the same time, as well as expiration-time policies. An expiration-time policy is necessary for SRM, since we assume that clients can be unreliable, and may not release files they are using. When a request is made, every file provided to the client for reading is automatically pinned in the disk cache. The client is expected to "release" or "unpin" the file when it is done with it. In case that the client does not issue a "release", the SRM uses the expiration-time to release the files.

When files are written into the disk cache, we also pin the file and associate an expiration-time with it. A file that is pinned as part of a put operation may be removed when the pin time expires, but if it has been marked as permanent it will first be migrated to tertiary storage.

It is useful to permit files to be attached "permanently" to a disk cache. This is useful if it is known that a file will be accessed repeatedly. However, permitting such an action by any user can clog the disk cache. For example, a user can make files permanent up to his/her quota limit, and leave the files there indefinitely.

We believe it is useful to provide status of files even if they were not requested for planning purposes. For example, a status request for 50 files can be made to an SRM, and the SRM returns the fact that 20 of these files are currently in its disk cache. The client may choose to request these 20 files, and the rest from other sites. Note that if the entire 50 file request was made to SRM, it is likely that the same 20 files that are in the cache will be provided to the client first subject to his/her quota and the other 30 files will be queued for fetching them from tape later.

File Naming and URLs

We assume that usually a grid agent first contacts a replica catalog to determine where the needed files are stored. It does this using a GFN (Grid File Name), also referred to as

the “logical file name” in the EU Datagrid documents. The replica catalog returns a list of SURLs (site URL, a site specific URL) to the client. These are used to find and contact different storage mechanisms (e.g. SRM, FTP, HTTP).

A grid agent can request services from the SRM using SURL filenames. Each SURL is presented to the SRM and it is mapped onto a protocol-specific TURL (transfer URL) that is used for the data transfer. Thus, the site is free to choose a “physical” location on any storage system it manages, as well as the transfer protocol that matches one of the protocols supplied by the client, and return that as the TURL.

We do not envision the SRM needing any knowledge of the transformation from GFN to SURLs or knowledge of the GFNs at all. If the replica catalog performs the translation between these, then it is left to the SRM to communicate only in terms of site-specific SURLs.

When writing files, there is within the SRM a third transformation from the SURL to a physical path name. This path is also site-specific and would not generally be exposed to grid-level clients. One exception to this is the writing of files where specifying of the local destination is important to the user, such as the path name that the mass storage system uses.

There are thus potentially four distinct filenames, two of which are embedded in URLs:
1.The Grid File Name (GFN)
2.The site-specific URL (SURL)
3.The site-specific path transfer URL (TURL)
4.The storage-system specific physical path (Not usually exposed)

The SRM concerns itself only with 2, 3 and 4.

Asynchronous Operations

Many SRM operations are long running because of resource contention. For example, a request to get a file might require that the file be read from tape, and there may be a queue for the tape drive. In this case, the return message contains a RequestFileStatus from the SRM will indicate that the operation is pending and it will provide an estimated time until the SRM client will see service begin.

It is the choice of the client to query the status within this estimated time period. For example, it may choose to check status after $t/2$ seconds to get either an updated time estimate or else the go-ahead to retrieve the file. In the case of an updated time, the client simply repeats the waiting/polling process.

However, the SRM should also provide a RetryDeltaTime time that should not exceed the length of a default policy expiration time. This has 2 advantages. One is that the client does not have to figure out when to check status again. This is because the file may be brought in earlier because of other clients, and it may be pinned also for the current

requesting client. By checking within the expiration time period, we can guarantee that this client will find out that the file was cached before its time expires.

This asynchronous protocol allows us to avoid the need for callbacks to the client. It does require that the SRM make a best effort to keep the files around for the time interval until the client polls for status again.

When a client contacts the SRM and makes a request, it will immediately receive a Request Status object in reply.

Request Status

Most SRM calls return a Request Status object. This object is a set of key/value pairs that describe the status of the request and every file in it. The file-related data is presented by a series of File Status objects nested inside the Request Status. We discuss the File Status below. The Request status itself consists of the values given in this table:

RequestStatus	
Key	**Value**
RequestId	An integer, the unique request id.
Type	The method that formed the request: Get, Put, MkPermanent, pin, unpin, requestStatus, fileStatus, ListProtocols, EstimateGetTime, EstimatePutTime
State	Status of the request as a whole: Pending, Active, Done, Failed.
SubmitTime	Date and time when the request was submitted
StartTime	Date and time when the SRM first started work on a file for this request.
FinishTime	Date and time when the last action for this request was completed.
EstTimeToStart	Estimted delta time (in seconds) until the request will become active.
FileStatuses	RequestFileStatus Objects, repeated for every file in the request
ErrorMessage	An error message to return to the user.
RetryDeltaTime	The client must re-check status with the SRM in this many seconds or the SRM may assume that the client has died and cancel the request.

File Status

There are two types of file status information. The FileMetaData object includes static attributes like its size or owner that is not part of a request to do some work to it. The second is RequestFileStatus, which contains request-related file information such as the state of the file or the expected start time.

FileMetaData Properties

Key	Value	Null OK?
SURL	The site specific file identifier	No
Size	File size in bytes	No
Owner	The creator of the file.	Yes
Group	The creating group for the file	Yes
PermMode	Unix permissions of the for 0644	Yes
ChecksumType	May be null. Example: CRC32	Yes
ChecksumValue	May be null.	Yes
IsPinned	Boolean or null	Yes
IsPermanent	Boolean or null	Yes
IsCached	Boolean or null. Is file in its most ready state?	Yes

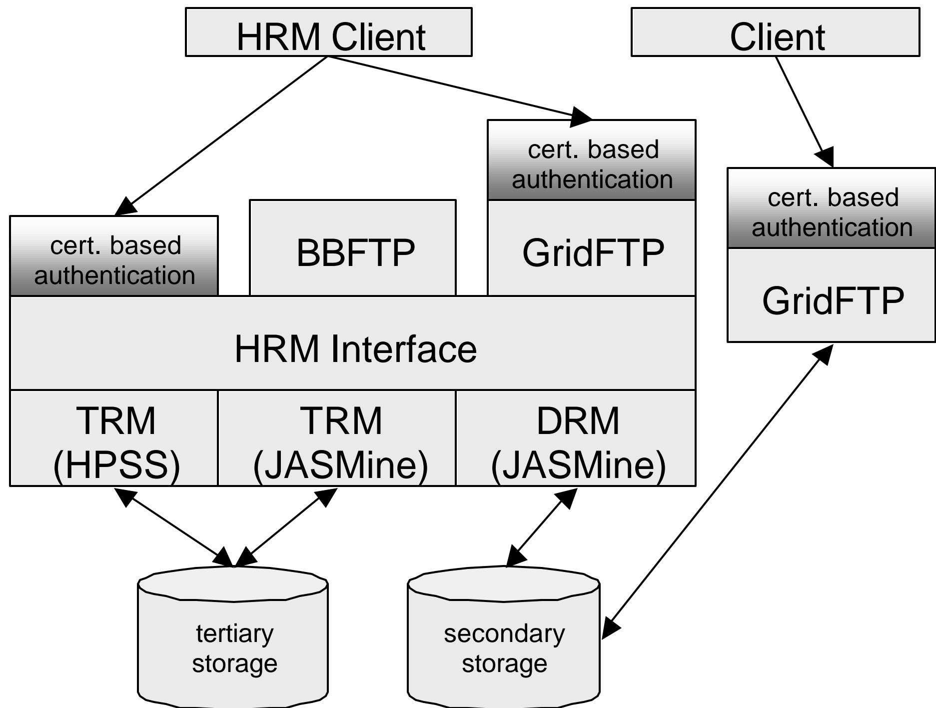
RequestFileStatus Properties		
Key	**Value**	**Null OK?**
SURL	The site specific file identifier	No
Size	File size in bytes	No
State	Pending, Ready, Running, Done, Failed	No
FileId	File ID within a request	No
TURL	Transfer URL	Yes, until state is Ready
Owner	The creator of the file.	Yes
Group	The creating group for the file	Yes
PermMode	Unix permissions of the for 0644	Yes
ChecksumType	May be null. Example: CRC32	Yes
ChecksumValue	May be null.	Yes
IsPinned	Boolean or null	Yes
IsPermanent	Boolean or null	Yes
EstSecondsToStart	Estimated time until the client may act on this file.	Yes

SourceFilename	Original filename. May not be meaningful to the SRM	Yes
DestFilename	Perhaps only useful to underlying mss for puts	Yes
QueueOrder	Expected order of service in the request	Yes
IsCached	Is the file in its most ready state?	Yes

SRM Methods

An SRM may reject a request outright if it does not have sufficient resources. If it accepts the request it must ensure that those resources remain available. For this reason, certain operations implicitly include other operations. In the example of getting a file, the SRM must make sure that the file is disk resident, which might involve staging it from tape. Further, the disk-resident file must not be removed before the client has retrieved the file, so a get must also imply a pin. Similarly with the put operation, once the request is accepted it is the responsibility of the SRM to make sure that the file will be accepted when the transfer begins, so the SRM reserves the filename, reserves the disk space, and makes sure that the request will succeed once it is started.

The SRM is not directly involved in the data transfer. In the following diagram we show an SRM client first contacting the SRM to initiate the operation and then the transfer service to perform the data transfer.



The basic SRM operations are given in the following table and then explained in more detail. Each operation is a call that returns an immediate response. This response does not complete the methods in most cases. This set of operations is meant to be a basic set upon which different transfer mechanisms can be fit.

SRM Methods			
Method	**Arguments**	**Return**	**Comments**
Get	SURLS, Protocols	RequestStatus	Implies reserving disk space, getting file to its most ready state, and pinning it.
Put	Src_file_names, dest_file_names, sizes, wantPermanent, Protocols	RequestStatus	Implies reserving disk space and dest file name, pinning the file, perhaps making it permanent.
MkPermanent	SURLS	RequestStatus	Check state in RequestStatus to see if it succeeded
Pin	TURLS	RequestStatus	
UnPin	TURLS, RequestId	RequestStatus	

SRM Methods			
Method	**Arguments**	**Return**	**Comments**
getRequestStatus	Request_ID	RequestStatus	Returns updated Request status including file statuses for every file in the request
getFileMetaData	SURLS	FileMetaData	Return file metadata. Not associated with a request.
getProtocols	None	Protocols	Returns list of SRM-supported protocols.
getEstGetTime	SURLS, Protocols	RequestStatus	Like Get, but does nothing except return status.
getEstPutTime	Src_file_namesm dest_file_names, sizes, wantPermanent, Protocols	RequestStatus	Like a Put, but does nothing except return status.
setFileStatus	RequestId, fileId, State	RequestStatus	Update a file from Ready to Running, or from anything to Failed by the client.
AdvisoryDelete	SURLS	None	Recommends that a file be removed before any others.

Get

The get operation takes a set of SURLs and a protocol list and returns a TURL (as part of the file status) for every SURL once the file is ready for transfet. The state in the RequestFileStatus indicates the progress of the file. Once the SRM is ready for the client to begin the transfer, the file state will be set to Ready. At that point the client calls UpdateRequest() to change the state to Running, and then initiates the transfer with the provided TURL. The TURL may be null until the file state goes from Pending to Ready. Once the client has completed the transfer it again updates the file's request status from Running to Done by calling the SRM.

Put

The put operation's file status transitions have the same names. Pending implies that the file is not yet ready to be received, perhaps because the SRM allows only a limited number of simultaneous transfers at once. Eventually the file status will become Ready, and the client will update the state again to Running and initiate the transfer. When the transfer is complete the client will again update the status to Done. See the use cases section for more detail on this interaction.

MkPermanent

Make an existing file permanent. This will fail if the file has been deleted or if the operation is not supported by the SRM.

Pin

Mark a file as in use. This pin has an expiration time that is set by the SRM. A second pin operation will renew the timeout, subject to local policy.

UnPin

An Unpin operation cancels a pin operation.

getRequestStatus

This updates the request status originally returned by a get, put, or other asynchronous SRM operation. A request ID is supplied and a complete Request Status object is returned. This object contains information about every file in the request, both its metadata and its request-specific data such as file states and expected queue ordering. A client should parse this structure and ignore those attributes that do not concern it.

getFileMetaData

Unlike RequestStatus, this call will only return information about files, not about requests. This is static file information like size, owner, and checksum. Most fields are optional and depend on SRM/MSS implementation.

getProtocols

This is a simple list of the transfer protocols supported by the SRM, ordered by descending preference.

getEstGetTime

This returns the estimated time in seconds from now required to start getting the file. This may depend on the SRM queue length and local policy decisions. If this is not known, it may return null.

getEstPutTime

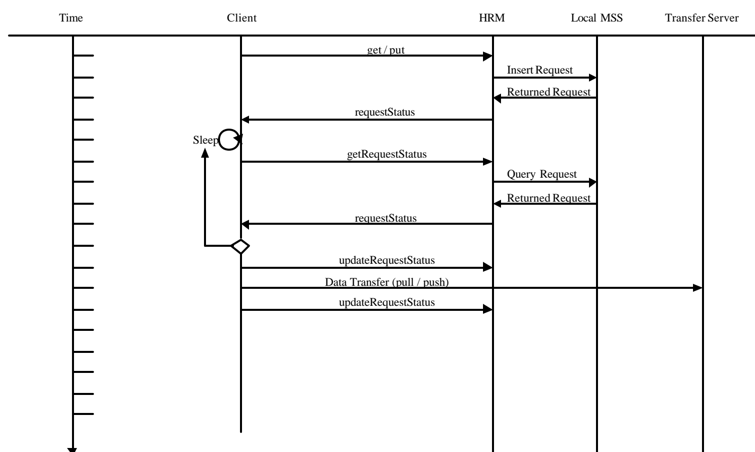
This returns the estimated the time required to start putting a file. This may depend on the SRM queue length and local policy decisions. If unknown, the SRM can return null.

AdvisoryDelete

This operation is a recommendation the 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 me thod.

SRM Use Cases

This protocol diagram demonstrates the client interaction with an SRM when transferring a file. There are at least two threads of execution within the client, a status thread and a transfer thread. It is the responsibility of the status thread to initiate the request with the SRM. As the response to this request it receives a RequestStatus containing a number of FileRequestStatus objects. It will recheck the RequestStatus with the SRM according to the interval mandated by the SRM so long as there is active work to be done with the request. As soon as the SRM changes the state of a file from Pending to Ready the client must first contact the SRM to update the state of that file from Ready to Running and then wake the transfer thread to perform the data transfer. Once the transfer is complete the client will again notify the SRM of the file's transition from running to Done.



This handshaking where the SRM changes the state to Ready and the client

acknowledges it by changing the state again to Running helps the SRM to provide a clearer picture of the state despite the fact that the SRM does not necessarily have close interaction with the data transfer.

Although the SRM cannot directly contact the client, it can detect a dead or unreachable client. If a client does not make any contact with the SRM within the RetryDeltaTime, the client can be assumed to have failed and all the unfinished files in the request must be failed.

Future Work

In the interest of moving toward a working prototype, certain functionality has been tabled for now.

SRM operation constraints based on user and group quotas are not part of this description. This is not to imply that they may not exist in some systems, there is just no means within this design to express them other than by request rejection.

Delete and Rename operations have been left out as has any operation that might imply a tight interaction with the replica catalog.

The SRM as a Web Service

By representing the SRM as a web service using SOAP/HTTPS we can produce a language-independent remote procedure call mechanism described using WSDL. Jefferson Lab's prototype implementation using this model works with both Java and Perl clients.

References

[1] "SRM Design Considerations", Arie Shoshani, Alex Sim, Junmin Gu, http://sdm.lbl.gov/srm/documents/srm-design-considerations-092601.pdf, and
"Description of SRM Interface and C++ bindings of SRM IDLs, Arie Shoshani, Alex Sim, Junmin Gu, http://sdm.lbl.gov/srm/documents/srm-idl-interface-description-092401.pdf

[2] "Data Management in Data Grids," Andrew Hanushevsky, http://www.slac.stanford.edu/~abh/PPDG/repl.html