SRM Joint Functional Design

Summary of Recommendations

January 2002

Contributors:
JLAB: Ian Bird, Bryan Hess, Andy Kowalski
Fermi: Don Petravick, Rich Wellner
LBNL: Junmin Gu, Ekow Otoo, Alex Romosan, Alex Sim, Arie Shoshani
WP2-EDG: Wolfgang Hoschek, Peter Kunszt, Heinz Stockinger,
Kurt Stockinger, Brian Tierney
WP5-EDG: Jean-Philippe Baud

1. Introduction

This document summarizes the conclusions reached for the functional specification of Storage Resource Managers (SRMs) by the participants of two recent meetings representing the US Particle Physics Data Grid (PPDG) project, and the European-Union DataGrid (EDG) project. The first meeting took place at CERN on October 11-12, 2001, between Arie Shoshani (LBNL) and people from the EDG listed above. The second meeting took place at LBNL on December 2-3, 2001, with participants from JLAB, Fermi, and LBNL listed above. The participants are people involved in the PPDG and EDG projects who are interested in SRM technology and who either developed or are in the process of developing SRMs. This document reflects the common wisdom and experience of people from both PPDG and EDG. It is intended as a guide for a joint PPDG-EDG document on the SRM functional specification. This document was written by Arie Shoshani.

In our discussions, we had the benefit from people's knowledge of four different archival systems: Fermi has experience with their own Enstore system, JLAB has its own home grown mass storage system, JASMine, LBNL has developed SRMs for HPSS, and CERN has developed their own system, CASTOR. Our goal is to achieve the generality of providing the same SRM interfaces to all these system, any disk cache systems, or any future storage systems.

The document is organized by topics, for each bringing up the issues and choices involved, and the recommendations made for the "joint SRM version". When appropriate we make a distinction between the "basic" level capabilities, and the "advanced" level capabilities. The "basic" capabilities should be supported by all SRM implementations, while "advanced" may be provided at the discretion of the SRM developers.

2. Issues and Recommendations

Issue1: What do we mean by an SRM?

We view an SRM as managing the use of a storage resource on a grid. The definition of a storage resource is flexible; it could be managing a single disk cache (we refer to this as DRM), or managing the access to a tape archiving system (we call this TRM), or both (we call this combination HRM for Hierarchical Storage system). Further, an SRM at a site can manage multiple resources. (We will address site management later.) The SRMs do not perform file transfer, but can invoke middleware components that perform file transfer, such as GridFTP.

Recommendation 1: Design the interfaces to all types of SRMs to be uniform.

By making SRM interfaces general, we allow any current or future hardware configurations to be addressed in the same manner. For example, an archive does not have to be a robotic tape system. Thus, the concept of "archiving" in SRM should not be tied up to a tape system; one should be able to archive into a disk system as well.

Issue 2: Chunk granularity

The granularity of chunks of data that SRMs can refer to depends on the type of applications we expect. The choices for the granularity that we considered are: object-level granularity, file-level granularity, or partial-file granularity (specified as offset+length).

Recommendation 2: File-level at the "basic" level, and partial-file at an "advanced" level.

We chose to avoid the granularity of an object-level, because this capability depends on the file organization, and requires specialized software to identify objects in that file. File-level was chosen as being the most common access need. However, some systems are capable of supporting "offset+length" specification. For example, CASTOR supports this feature for its DRM part. This is especially useful for access from a tape system where only the header of a file is requested, or a small part of a file. It also cuts down on the data to be transferred over the net. For this reason, we included this feature at the "advanced" level.

We note that another advanced feature might be to extend the SRM interfaces where a "filtering-spec" is allowed. This could be useful for SRMs that can invoke a filtering program, which extracts the desired part(s) of a file before providing it to the client. This is a possible way to access objects or desired parts of files that cannot be specified with a simple "offset+length". We decided not to pursue this further at this time.

Issue 3: What entities can communicate with SRMs?

One design choice is to make SRMs accessible by other middleware software only, such as request planning or request execution modules. Another is to permit client programs to invoke SRMs directly. An additional design choice is whether SRMs could communicate with each other, and request files to be transferred from one SRM to another.

Recommendation 3: Support all of the above choices.

We see the need for clients to communicate directly with SRMs. Some client programs may be sophisticated enough to request space to dump computation results into or to request files they need for analysis.

A typical use case is a client requesting a file from an SRM. If the SRM has that file, it will pin it for the client's use. But, if it does not have the file, there are two choices: 1) tell the client it does not have it, and 2) request the file from its source location and invoke a transfer to its disk cache. We made the second choice. In this case, the SRM will communicate with another SRM requesting the file on behalf of the client. This choice was made in order to simplify the communication needed by a client program. It can simply make the request for the desired file from an SRM, and let the SRM manage the coordination with other sites if files have to be brought over. This also demonstrates the need for SRMs to communicate with each other.

Issue 4: Should we support a request for "multiple files"?

The issue is whether to support a request for a set of files made all at once, rather than one file at a time. This will add to the complexity of the basic version. The advantage from a client's point of view is that it does not have to keep track of which files it requested. Instead, the SRM can be interrogated about the status of the request. Furthermore, the advantage to the storage system is that it can choose the order of files provided to the client, thus optimizing its resource usage. For example, a request to get a set of files from a tape system can be re-ordered to minimize tape mounts, by accessing all file from the same tape at the same time. Similarly, disk caches can maximize their use by sharing files needed by multiple clients.

Recommendation 4: Providing the capability to request reads and writes of multiple files.

The implication of this decision is that a "request" concept is introduced, where the status of all the files in a request can be inquired, the time estimation for a request can be made, and aborting a request implies aborting all the files belonging to that request.

We also considered the possibility of having a request for an *ordered* set of files. While we could imagine some applications needing files in a certain order, this is not a common requirement in HENP applications. Also, we noted that requests to files in a particular

order can be achieved by issuing multiple requests, each for a single file. Therefore, we excluded the capability to request an ordered set of files.

Issue 5: who assigns the requestID?

RequestIDs are needed for follow up on the status of a request or for terminating a request. The two options are: the client assigns the requestID, or the SRM assigns the requestID. The advantage of SRM-assigned IDs is that there is no possibility of assigning the same ID multiple times, and there is no need to check for identical Ids by different clients. Given that requests may be kept for a long time (for tracking of file movements), user-assigned IDs are too difficult to manage. On the other hand, SRM-assigned IDs are meaningless to clients, and can be long integers or strings. Since the client needs to keep the IDs to refer to them later, user assigned IDs are easier to remember by client.

Recommendation 5: Support SRM-assigned requestIDs, with an optional requestID-description.

The choice is to have SRM-assigned requestIDs, but allow a request-description to be provided by the client. SRMs would be required to return all requestIDs associated with a request-description string. All subsequent calls about a request (such as "status" or "abort") have to use the requestID. In case that a requestID is lost, the client can provide the SRM with a requestID-description, and get back the requestID. Note that a client is allowed to assign the same request-description to multiple requests. In this case, the SRM will return back all the request-descriptions associated with the requestID along with the time the requests were submitted.

Issue 6: Should SRMs support asynchronous requests?

Typically, services provided by a conventional disk storage system are synchronous. However, if it is possible for a response to take a long time (more that a few minutes), then asynchronous (non-blocking) response if very useful. This is because the calling process can be notified on the time till the service will be provided, and can issue status requests. Otherwise, the client cannot find out what is happening to the request it made. Long delays can occur when accessing tertiary storage or when transferring large files over the network.

Recommendation 6: Asynchronous calls should be supported.

Issue 7: Should a callback capability be supported?

Callbacks are very useful for asynchronous requests. For example, if a request for a file is made, and the file needs to be retrieved from tape, it may take a while before the file is made available depending on the system load. A callback capability will notify the client when the file is available. The alternative is to let the client perform repeated "status" calls to the system till the file arrives. The difficulty with requiring callbacks to be

supported is that the client needs to have some kind of daemon or server listening. In general this cannot be assumed, and there may be firewalls that do not permit callbacks to a client that is behind that firewall.

Recommendation 7: Support for "request-status" only in the basic version, and a callback capability in the advanced version.

Since callbacks may not be available, there needs to be a way to restrict very frequent status requests, which can increase the system load. We decided that it is up to each implementation to find a way to deal with this, such as restricting the number of status calls per request, controlling the status update frequency, etc. However, we also concluded that the system can choose to provide a optional "suggested-time-for-next-status" so that well-behaved clients can use that as a guide.

While a callback capability is not required at the basic level, SRMs can choose to provide that as an advanced feature. This can be very helpful for situations where only SRMs are involved in file movement coordination, such as automatic file replication. Since an SRM can behave both as a client and a server, SRMs can choose to coordinate their communication with callbacks.

Issue 8: Status of files stored in SRM-controlled resources

This issue involves the designation of the file type as "permanent", "durable", and "volatile".

Obviously, some storage systems should be able to designate files as "permanent". This is typical of archival storage where results of experiments, simulations, and other valuable data are stored. We avoid the term "master-copy", as one should be free to have several permanent copies of the same file. We do not limit archival storage to be tape archives, RAID systems, or for that matter any specific type of device. An archival storage is simply a storage system that permits "permanent" files to be stored on it. A permanent file can be removed only by the owner, or by the system administrator.

In contrast to "permanent" files, shared storage resources may choose to support only "volatile" files. This is typical of large shared disk caches. Volatile files can be removed by the SRM when space is needed. However, since there is a need for some minimal time that a file resides in the cache (to give the client a chance to access the file), the concept of file pinning is necessary. A volatile file is pinned in the cache for a certain "lifetime" period. The length of the "lifetime" is the choice of the SRM administrator or the SRM's policy. Usually, a file is expected to be "released" or "unpinned" by the client before its lifetime expires. Provisions can be made for extending the pinning of a file, but we felt that honoring pinning extension requests should be an implementation choice as well.

The concept of a "durable" file is a file that is intended to be removed as soon as possible, but should not be deleted by the SRM. Like a volatile file, it has a "lifetime"

associated with it (perhaps longer than that of a volatile file), but when its lifetime expires a system administrator is alerted. Similar to a permanent file it can be only removed by the owner or the administrator. Thus, the concept of a "durable" file has the features of both volatile and permanent files. The need for a "durable" file status was inspired by the scenario of files generated by some compute resource, and there is a need to temporarily store them in a shared space before they are archived. Normally, the files are stored in the shared space as "durable", and then scheduled to be archived on some other archival storage system. After the files are archived, they are released either automatically by the archiving SRM or by the client. In case that the client neglects to release them, an administrator is alerted when the lifetime expires.

Considering these options, the issue was whether to support them all in the basic version.

Recommendation 8: Support all three types in the basic version: permanent, durable, and volatile.

The choice whether to keep "permanent" files is an SRM choice. Clearly, SRMs that manage access to archival storage systems such as HPSS, CASTOR, JASMine or Enstore should support permanent files. However, SRMs managing shared disk caches can refuse to support permanent files. Yet, even SRMs designed only for shared volatile files, should support "durable" files as a way to provide temporary storage for files on the way to be archived elsewhere.

Issue 9: Space allocation model

Consider an analysis request made by a client for a large number of files (say, 500 1GB files) to an SRM that manages a shared disk (i.e. a DRM). One model is to allocate the space for the 500 files, or negotiate the amount of space that the client can have. We refer to this model as the "on-demand" model. The other possibility is to have the SRM allocate a certain amount of space to the client according to a quota determined by its policy. Then, the SRM brings in files up to the quota level. Only when a client releases one of the files, SRM will schedule another file to be brought in provided that the total space used is below the quota. We call this the "streaming" model.

Both of these models have advantages. The advantage of the "on-demand" model is that it lets a client plan better its available space. This is useful, for example, when running a simulation and the amount of space for generated data can be predicted. However, managing allocations is complex, since the system can be abused with very large allocation requests that are not actively used. In order to prevent abuse, there must be some external per-user (or per-group) quota assignment mechanism, and the SRM managing usage (MB-hours) counted against it.

The advantage of the "streaming" model is that a client does not have to deal with allocations or quotas. When it submits a request for a large number of files, the system queues the file requests, and allocates a quota automatically to bring in part of the files up to the assigned quota. The quota assignment is a local SRM policy with the goal of

treating all clients who share the cache fairly. When the client releases files, additional files are brought in automatically. The "streaming" model can prevent space abuse by setting the allocation quota to be initially low, and only when the release rate is high, the quota increases. Similarly, the pin-lifetime prevents files from being kept in cache for unreasonable long time.

Recommendation 9: Permit both the "on-demand" and the "streaming" models in the basic version, but it is an implementation choice for each SRM.

We all agreed on the usefulness of the "streaming" model, but we felt that "on-demand" should be supported as well by SRMs that choose to do that. As mentioned above, while the streaming model is attractive for managing shared volatile storage, there are situations where pre-allocation of space is necessary, such as in the data generation phase. Also, some systems already support space management, especially for archival storage. For example, CASTOR permits space pre-allocation to tape, but not to disk. For these reasons, we chose to allow both models, but leave it to the specific storage system and SRM implementation as to what is actually supported. For example, an SRM that supports only the "streaming" model could always respond to a space allocation request with its local policy quota that would have been automatically assigned anyway.

Issue 10: Transfer protocol negotiation

When making a request to an SRM, the client needs to end up with a protocol for the transfer of the file that the storage system can support. In general, systems may be able to support multiple protocols and clients may be able to use multiple protocols depending on the system they are running on. One can choose a single protocol that any SRM client has to use (such as GridFTP), but we felt that is too restrictive. There could be some university researcher without access to GridFTP who wishes to access files through regular FTP, BBFTP, etc. There needs to be a way to match the transfer protocol that a client has with that of the SRM's storage system. The choices are to have either the client provide a list of protocols and SRM chooses one, or vice versa.

Recommendation 10: Client provides an ordered protocol list, and SRM select the protocol to be used.

We selected the option of the protocol choice made by SRM because depending where the file is stored a different protocol may be used. This is true of SRMs that manage multiple sub-systems, where each may be supporting a different protocol (for security or other reasons). Alternatively, sites having SRMs may advertise the protocols they support in some resource directory (such as MDS), so clients may choose the desired protocol ahead of time and submit only that protocol.

Issue 11: File names and URLs

The terminology introduce by various groups was predictably different and confusing. After long discussions we reached an understanding and an agreement of the meaning of

terms. Below are the terms and meanings that emerged. Some terms were suggested by people from PPDG and some by the EDG.

LFN- is a logical file name that is globally unique for a given dataset. Thus, the dataset name is usually the first part of the LFN. It is the choice of the dataset designer how to assign these names. If the files are organized in directories then the directory names are part of the LFN. The dataset name and directory names are separated by "/". An example of an LFN is: "CERN-dataset-7/run17/part1/file-123".

SFN- is a file name assigned by a site to a file. Normally, the site file name will consist of a "machine:port/directory/LFN", but the site can choose to use another string instead of the LFN. An example of an SFN is: "sleepy.lbl.gov:4000/tmp/foo-3000". In this example we used the simple file name "foo-3000" to simplify the example.

SURL – is a "site URL" which consists of "protocol://SFN". The protocol for communicating with an SRM is simply "srm". An example of an SURL for a file managed by SRM is: "srm://sleepy.lbl.gov:4000/tmp/foo-3000".

TFN – is the "transfer" file name of the actual physical location of a file that needs to be transferred. It has a format similar to an SFN.

TURL – is the "transfer URL" that an SRM returns to a client for the client to "get" or "put" a file in that location. It consists of "protocol://TFN", where the protocol must be a specific transfer protocol selected by SRM from the list of protocols provided by the client (see recommendation 9). If the physical storage location matches the one provided by the SURL, then only the protocol is replaced in the TURL. For the above SURL example, and assuming the protocol is "gridftp", the TURL will be: "gridftp://sleepy.lbl.gov:4000/tmp/foo-3000". However, the physical file location can be anywhere at that site, giving the freedom for the site manager to change the physical locations of files without having to change the SURL or update the replica catalog. If for the above example the physical location of the file is on another machine (e.g. "dm.lbl.gov", another path (e.g. "/home /level1"), and even another file-name (e.g. "abc-3000") then the TURL will be: "gridftp://dm.lbl.gov:4000/home /level1/abc-3000".

StFN – is the "storage" file name that a client may request SRM to use when it stores the file. This is useful for telling SRMs where to archive a file. Normally, SRMs that archive files, such as an HRM, may choose to honor that request. But, the SRM may choose to give it another name, and return that to the client.

Recommendation 11: Use the terms LFN, SFN, TFN, SURL, TURL, and StFN as explained above.

Issue 12: Site management

There are sites that manage multiple storage resources. An example is the SAM system at Fermi Lab which uses a metadata directory to map "site file names" (SFNs) to storage

file names (StFN). This gives the site the flexibility to move files around from one storage device to another without the SFN changing. Thus, the SFN does not have to correspond to the storage location of the file. This is consistent with the definition of SFNs above. In this case, the TURL that SRM returns to the client provides the client with an TFN=StFN name for transfer purposes. In this case, the TURL returned is simply the StFN prepended with the protocol.

In addition, some sites may want to have multiple SRMs on multiple machines, but a single entry point to the site. To support this capability, we propose to support re-direction. For example, suppose that the single entry point may be the machine called "main" at Fermi, and the machine that actually manages this file is called "auxiliary1". An SURL for a file might be: "srm://main.fermi.gov:4000/tmp/foo-3000". The redirect sent to the requester will be: "srm://auxiliary1.fermi.gov:4000/tmp/foo-3000". This instructs the client to contact the auxiliary machine directly.

Finally there is the issue of running multiple SRMs on the same machine. How would each be referred to? Two suggestions were made: 1) use a different name for each and register it with the DNS service; and 2) use different port numbers. We note that the first choice is not sufficient, since the DNS server will map the names to the same IP address. So, different ports must be used, even if different names are given to each.

Recommendation 12: 1) Permit an SURL that have different file path and names than the returned TURL for that file; that is, the SFN in the SURL does not have to be the same as the TFN in the TURL. 2) Provide redirect capability for site entry managers for sites that choose to use that. 3) Use different ports for multiple SRMs on the same machine.

Issue 13: Fundamental file movement requests

The most basic request that a client can make to an SRM is to "get" and "put" files. However, unlike the FTP (or GridFTP) "get" and "put" a request to an SRM entails space allocation by the SRM, and getting files from other source locations if the file does not exist locally. Actually, SRMs do not move files at all, but after they allocate the space, they call file transfer servers (FTP, GridFTP) to transfer files. To avoid the confusion, we labeled the functions "SrmGet" and "SrmPut".

When a request to get files is made (SrmGet), the SRM needs to check for each file if it is already in the disk cache (it could be there because another client asked for it), and if not, allocate the space necessary for the file and bring it (replicate it) from its source location. In the case of a DRM, it involves getting the file over the network if necessary from its source site, which may require communicating with another DRM or HRM. In the case of an HRM, it may need to get a file either from a remote site or from its MSS. Getting a file from its MSS may involve getting the file from tape. We note that a request for a file that resides in a remote site is not transitive. That is, if the remote site does not have the file, an error is returned; the remote site is not expected to get the file from another site.

When a request to put files is made (SrmPut), then for each file the SRM has to allocate space, and return to the client the location for the client to "put" the file into. In the case that a file is "permanent", putting the file implies archiving it. Thus, a "permanent SrmPut" to an HRM would move the file to tape, and a "permanent SrmPut" to a disk, would mark it as non-removable (except by its user or an administrator).

Another issue is whether to support a third-party SRM request. In this case the client does not want to get or put a file, but rather to move it from one SRM to another. The use case that inspired this is file generation. In this case the client wants to place the files in some "close-by" available disk cache, and then move the files to an archive in another system/site. Another common need for this capability is file replication. For example, replicating files from one archive (perhaps where the data was generated) to another archive (perhaps where the data will be used). We call this function "SrmCopy". The question was whether to include this in the basic version.

Recommendation 13: Support "SrmGet", "SrmPut", and "SrmCopy" in the basic version.

The main reason to support the third-party "SrmCopy" is that it allows clients to move files between storage resources on the grid. An SrmCopy has to have a "source SURL" and a "target SURL". After the file is "copied" from the source system to the target system, the target SRM sends a "release" to the source system. The source system will "release" the file if it was volatile. For removing durable and permanent files at the source site, we decided to add a "release-flag" in the SrmCopy function. If this flag is on, the SRM will advise the source location to release the file given that the user is the owner of the file. This capability is provided to accommodate moving of files from a temporary location to an archive.

There is some similarity between SrmGet and SrmCopy (pull mode), in that both are designed to get a file from a remote location. However, there are two differences between them: 1) SrmCopy will permit a "release-flag" as discussed above, while SrmGet is only for the purpose of replicating files and this option is not available with it. 2) If the file is already in SRM's cache, SrmCopy will transfer the file, write over the file in the cache, while for SrmGet, if the file is already in cache, no transfer occurs. The latter capability permits file sharing, an essential capability for efficient use of the SRM.

Issue 14: push vs. pull

Consider the need to put a file by a client into an SRM. There are two options: 1) the client makes the SrmPut request, gets back a TURL location, and proceed to perform a GridFTP "put" transfer to that location; or 2) the client provides the SRM with the location of its file, and the SRM performs a GridFTP "get" to transfer the file from the client's location. We call the first case a "push mode", and the second a "pull mode". Similarly, when requesting an SrmGet, it can be performed in either a "pull mode" (the client pulls the file from SRM) or a "push" mode" (the SRM pushes the file to the client).

One can choose to support both of these modes, and the case can be done for each. In general, a "pull mode" by the SRM is safer because the SRM is in control since it initiated the "pull" and it can abort the transfer in case a larger file than expected is being transferred. Otherwise, if a file is pushed into the SRM and it exceeds the size of the allocated space, it cannot be directly stopped. More complex schemes need to be devised for aborting a file transfer being pushed.

We did not feel that it is worth supporting both modes in the basic version. The issue of which modes to support was determined mainly considering firewalls. If an SRM is behind a firewall, then the obvious choice is to support SrmGet in a "pull mode" and SrmPut in a "push mode".

SrmCopy can be requested either from the source site or from the target site. If it is requested from the source site, then the file has to be "pushed" by the source SRM to the target site. Conversely, if it is directed to the target site, the file has to be "pulled" by target SRM. The first case, which requires a "push" may be a problem for SRM behind firewalls. For this reason, we recommend that only the "SrmCopy-by-target" will be supported by the basic version. If an "SrmCopy-by-source" is requested, then that source site can redirect the request to the target site. This practice is OK as long as the target site has an SRM. To accommodate the case that a target system does not have an SRM, the "SrmCopy-by-source" has to be implemented. We consider this an advanced capability.

Recommendation 14: for the basic version, support only SrmGet in pull mode, SrmPut in push mode, and SrmCopy-by-target (which implies a pull mode). It is an implementation choice whether to support SrmCopy-by-source (which implies a push mode), as it is considered an advanced capability.

Issue 15: Suspend and resume requests

Suppose that a client running a long analysis requiring many files fails. When the client returns, it needs to resume the analysis at the point where it stopped. If the client does not keep track of what files were already processed, it needs to start the entire analysis task over again. However, since the SRM keeps track of the files requested and released, it can resume the request from the point of its failure. Similarly, it may be necessary for a client to suspend the analysis waiting for compute resources. In this case a "resume" functionality will be useful as well.

At the time of failure or suspension the files fall into three categories 1) files that were provided to the client and released; 2) files provided but not released; and 3) files not provided yet. Should files in category 2 be re-scheduled? A similar question exists for files being written.

Recommendation 15: Support "suspend" and "resume" in the basic version. When we resume an SrmGet, files requested but not released are re-scheduled. For files being "put" as a result of SrmPut, if SRM does not get a notification that the "put" was completed for some files, they are removed.

Issue 16: what file status to assign to files for SrmPut

SRMs that do not support permanent file storage (such as a shared disk cache) should be designed to permit files written into them to be saved till they are moved to an archive. That implies that SRMs should support at least "durable" status.

Recommendation 16: SRMs should support either durable or permanent files or both. This is a requirement of the basic version.

Issue 17: should file sizes be required in SrmGet and SrmPut

If the wrong file size is provided with SrmGet and SrmPut, then the space SRM allocates for that file may overflow or underflow. An underflow is easy to support, but an overflow may interfere with other transfers, or may cause the SRM to fail if it does not guard against it. Thus, it is necessary to monitor the file transfer for overflows.

Given that this is done, it should be possible to handle request that do not specify the file size at all. This can be done by allocating a default file size (say one GB), and if the transfer exceeds that, allocate additional space till the transfer is finished or the maximum limit reached. We considered these options.

Recommendation 17: For the basic version require file sizes to be provided, and fail any transfer whose size is imprecise.

The reason for the above decision is that permanent files as well as replicated files requested by clients are expected to be registered in a file catalog along with their file sizes. Failure to provide the correct size indicates that they were registered incorrectly. Similarly, the sizes of files being written are known to the client, and should be accurate when provided to SRMs.

It is an implementation choice whether a more flexible approach will be accommodated.

Issue 18: communication protocol

We concluded that we should agree on a single protocol for all communications with SRMs. The choices considered are web-based, CORBA-based, extended FTP, etc.

Recommendation 18: Use soap/xml protocol over https as transport protocol; express functional spec in WSDL (described in http://www.w3.org/TR/wsdl).

The reason for this choice is that this protocol is becoming popular and is already supported by many vendors.

Issue 19: Should unix-like functions (mkdir, rmdir, ls, mv, rm) be supported?

For SRMs that support archiving, such functions are necessary. If SRMs do not permit such functions to be expressed, then the client will have to perform them directly with the archival system. However, the ability to provide support for these functions by SRM, depends on its ability to interact with the storage system at this level. For example, to perform such operations on HPSS will require HRM to have access to a client API or HSI. These may not always be available. CASTOR, Enstore, and JASMine all provide such access to SRMs.

Recommendation 19: Support these functions for archival systems. Supporting these in a non-archival system is an implementation choice.

Issue 20: providing estimates

Estimates are important both for a single file, and for an entire request for multiple files. The estimate of how long it will take to get a file depends on the source of the file and whether there are other files in the SRM's queue. The estimate includes the time to get the file out of a tape archive if necessary as well as the transfer time over the network. Estimation is very useful for determining where to get a file from if it exists in multiple sites, or even whether to run a request at all.

Accurate estimation is difficult to produce without good monitoring tools. But since estimation is so useful, an imprecise estimation is better than no estimation at all.

Recommendation 20: Because of their importance, we recommend support of estimation as best we can. These functions should be included in the SRM protocol.

Issue 21: Should SRMs be multi-threaded?

Recommendation 21: Yes, so that they can be called by multiple clients without blocking. They should be thread-safe.

Issue 22: Should time-out be provided by users for a request?

Recommendation 22: We consider this a useful but optional parameter that a user can provide. We consider that useful because resources can become unavailable after the request has started and before it has been completed (network failure, hardware reconfiguration or relocation...) and for some applications, it is better to get a timeout, rather than waiting for hours or days.

Issue 23: Should LFN be required in file request in addition to SURL?

SURL is required in specifying the source from which to get the file. It would be useful for SRMs to have the LFN, too, to permit SRM to share files. For example, suppose that LFN=L1, and three are 2 replicas SURL=S1, SURL=S2. Suppose that the SRM was provided only with SURL=S1, and got the file for a client. Then, if another client requests SURL=S2, there is no way for SRM to know that it has already the same file in

cache, and cannot hand the file to the second client. Instead it will get the same file a second time.

This problem can be avoided if we could rely on a file catalog to provide the reverse mapping SURL → LFN. When this becomes a service we can use, we would eliminate the parameter LFN altogether.

Recommendation 23: We chose to allow LFN to be provided optionally in addition to SURL.

This is a temporary measure until we can rely on a file catalog for the reverse mapping. We note that by allowing this there is a danger in case that a client gives the wrong LFN. Suppose that in the above example, Client 1 request SURL=S1 and also provides LFN=L2 (instead of L1). Then, if another client ask for SURL=S3 with a correct LFN=L1, it will get the wrong file (i.e. the file with SURL=S1). Thus, in the future LFN will not be provided.

Issue 24: Should we include a renew lifetime method?

The reason to allow this is to help deal with long delays, and lifetime expiration before a file is used by the client. For example, suppose that a client asks for 50 files, and the source SRM may be able to pin all 50 files. If the network is slow, it can happen that the lifetime limit arrives before all the files were transferred to the destination. In that case, renewing a filetime just before actual transmission will tell the source SRM to keep a file for an extended lifetime to allow for the file transfer before the lifetime expires.

There are two cases to consider. 1) The file is still in cache when renewal is requested, even if this renewal is after lifetime expired. In this case SRM re-pins the file, and all is well. 2) The file lifetime expired, and the file was removed because space was needed. In this case, the client will not transfer the file, and wait until it is brought in again.

The negative effect of this method is that it can be used repeatedly to extend the lifetime over and over. To avoid this possibility SRM can manage that by some policy they prefer. An example could be to limit the lifetime to a single renewal only, or some limited number. Another is to ignore the request unless it is made close to the end of lifetime. Such policies is an implementation choice.

Recommendation 24: Provide a Renew Lifetime method, and have individual implementations choose how to treat it.

3. Summary of methods and concepts

3.1 Summary of methods and their functionality

•SrmGet()
— Purpose: to bring a file into SRM's cache
— If the file is already in cache, pin, and return lifetime; otherwise
    - allocate disk space,
    - if file is in local archive, bring to cache, pin, and return lifetime
  Or
    - if the file is at a remote site, get it, pin, and return lifetime
— "Permanent", "durable", or "volatile" may be specified; default: "volatile"
— SRM may choose to refuse "permanent", "durable"

•SrmRelease()
—Release (or unpin) a file

•SrmPut()
— Purpose: a request for space allocation before performing a put
— Allocate space, return lifetime for space allocation
— Will overwrite the file if file is already in cache
— Default is "durable"
— SRM may choose to honor "permanent", or treat it as "durable"
— StFN may be supplied
    - if supplied, either agree and return TURL + SURL
    - or refuse (SRM does not accept StFN), return site assigned TURL + SURL
    - if StFN not supplied, return site assigned TURL + SURL

•SrmPutDone()
— Purpose: notify SRM that the "put" completed
— Setup file status according to its requested "type"

•SrmCopy() (pull mode)
— Purpose: third party file movement from a remote site to the SRM (pull)
— Allocate Space
— Get from a remote site – use SURL
— Put into SRM's space, possibly into archive
— Release-flag option: release source file after transfer for all file-status types
— StFN may be supplied
    - If supplied, either agree and return SURL
    - Or refuse (SRM does not accept StFN), return site assigned SURL
    - If StFN not supplied, return site assigned SURL

•SrmCopy() (push mode)
— Purpose: third party file movement from local SRM to a remote site (push)
— Get from SRM's space, possibly from archive – use SURL
— Put into a remote site
— Release-flag option: release source file after transfer for all file-status types
— StFN may be supplied
    -If supplied, either agree and return SURL

-Or refuse (SRM does not accept StFN), return site assigned SURL
-If StFN not supplied, return site assigned SURL
— An implementation choice for basic version: return "re-direct" to target site

•SrmTerminateRequest()
— Terminate an entire request for all files, even if not all file were requested or released

•SrmAbortFile()
— Abort request for that file, abort transfer if in the middle of transfer.

•SrmChangeFileStatus()
— Request to change from current status to new status (P/D/V)

•SrmRequestSuspend()
— Suspend request. All files currently in cache stay in cache till their lifetime expires.

•SrmRequestResume()
— Resume request. All file requested by not released are added to list of file still to be acted on.

The functions below are for the purpose of finding the status of request, getting estimates, and general administrative functions. Most are similar the functions discussed in srm.v1.0.doc (http://sdm.lbl.gov/srm/documents/joint.docs/srm.v1.0.doc).

•SrmGetRequestStatus()
— Which files of that request are in cache, lifetime remaining, files still queued, etc.

•SrmGetFileMetaData()
— For files not associated with a request, to find out if in cache and their status.

•SrmGetReqEstTime()
— Get best possible time estimate for that request. If request already launched it is an estimate for time remaining. To get an estimate of a request without launching it, the suspend/resume commands can be used as follows: make request, suspend immediately, request estimate; then request can be either resumed or aborted. Note: this command can be used to get an estimate on a single file, by making a request of a single file.

•SrmGetProtocols()
— Get the set of protocols that the SRM's site supports

•SrmAdvisoryDelete()
— A recommendation to SRM to delete a file as it will not be accessed again by the client. Can be ignored by SRM depending on the use of the files by other clients.

3.2 Summary of Srm Get, SrmPut, and SrmCopy differences

Table 1 highlights the differences between the three basic SRM functions: SrmGet, SrmPut, and SrmCopy. The first column represents the default file-status assigned, in case that a file-status is not specified with the function call. The second column refers to whether a "release-flag" can be set, to indicate whether the source files should be deleted. As can be seen this is available only for the SrmCopy. The third column indicates whether the file will be written over in case that it is already in the SRMs storage (including archival storage).

	Default status	**Release-flag**	**Write-over**
SrmGet	Volatile	No	No
SrmPut	Durable	No	Yes
SrmCopy	Permanent	Yes	Yes

Table 1: differences in behavior for SrmGet, SrmPut, and SrmCopy

3.3 Summary of SRM's actions based on file-status

Table 2 summarizes the action expected of SRMs when files are assigned one of the status types: volatile, durable, and permanent. The first column indicates whether the SRM will schedule the file to be archived. If the archival system involves a tape archive, an SRM make take some time to actually move a file to archive, depending on the system load. The archive may be any storage system, such as disk space reserved for archival purposes. The second column indicates whether a lifetime will be assigned to the file. Note that it is local policy choice what is the length of time assigned for "volatile" and "durable" files. The third column indicates whether the SRM may remove a file if space is needed. Additional details were discussed in Issue 7 above.

	Archive	**Assign lifetime**	**Remove if space needed**
Volatile	No	Yes	Yes
Durable	No	Yes	No
Permanent	Yes	No	No

Table 2: SRM's actions based on file-status type

Note that an SRM is not obliged to support all the file-status types, depending on the functionality it is intended to provide. For example, an SRM that manages only a disk (DRM) intended to be used as shared temporary space may choose to provide only "volatile" for SrmGet, and only "durable" for SrmPut and SrmCopy. Further, the durable space may be limited by a quota per user set by the administrator of that DRM. Similarly, an SRM that manages access to tape storage (HRM) may choose to use its cache only for the purpose of staging files into and out of the tape system. It may choose to provide only "volatile" for SrmGet, and only "permanent" for SrmPut and "SrmCopy". Further, it may refuse to perform SrmGet for files that are not local to its storage. The SRM design is intended to support any local choices made by various implementations of SRMs supporting various types of storage systems.

3.4 Summary of the source and target sites for SrmGet, SrmPut, SrmCopy

Table 3 is intended to summarize the source and target sites that each of the SrmGet, SrmPut, and SrmCopy can address. This reflects the intended use of these functions.

An SrmGet is intended to be used by a client or a program acting on behalf of a client to get files it needs. Therefore, the target SURL can be either specified to be local to the SRM, or unspecified. In case that it is unspecified, the SRM return the TURL where the file is. Even in the case that the local location is specified the SRM may return a TURL of its own choice. The source SURL can be either a URL local to the SRM or a remote URL. Under normal circumstances, once the SrmGet returns, it is expected that the client will access that file within the lifetime assigned to it, and then release the file.

An SrmPut is also intended to be used by a client or a program acting on behalf of a client to put files from its space into the SRM. The client expects space to be allocated, and then it performs the "put" transfer into the SRM's space. Therefore, the source SURL is not specified (i.e. null). The target SURL may be unspecified (in which case the SRM will return a TURL where to put the file), or may be specified. Even in the case that the target SURL is specified, the SRM may return a TURL of its own choice. It may also return, in addition, an SURL that the client has to use for future reference. This capability is provided for sites that manage multiple storage resource systems, and wish to have a different SURL and TURL for a file stored in their systems.

An SrmCopy is intended to be used by a client or a program acting on behalf of a client for moving files from one storage system to another. When the "release-flag" is set the file at the source will be removed after a successful copy. Under normal circumstances, it is not expected that the client will access that file immediately after the move. The SrmCopy can also be used for replicating files; this is simply achieved by not setting the "release-flag". An SrmCopy must be provided with a source SURL and a target SURL. In case that the SRM's site coincides with the source SURL site, then the SrmCopy is expected to "push" the file to the target site. Conversely, if the SRM's site coincides with the target SURL site, then the SrmCopy is expected to "pull" the file from the source site. As can be seen from the table, we have chosen to support only the SrmCopy in a "pull mode, and redirect the SrmCopy in a "push" mode.

	Source (SURL)	Target (SURL)	Supported
SrmGet	Local/remote	Null/local	Yes
SrmPut	Null	Null/local	Yes
SrmCopy (pull mode)	Remote	Local	Yes
SrmCopy (push mode)	Local	Remote	No (redirect instead)

Table 3: the source and target SURL for SrmGet, SrmPut, and SrmCopy

Acknowledgements

We thank Miron Livny for suggesting the suspend/resume functions that we ended up including in the SRM design. We also thank other members of the PPDG team for feedback on the design issues of SRMs.