SRM Design Considerations

Arie Shoshani, Alex Sim, Junmin Gu

August, 2001

1. Introduction

In this document, we describe the consideration for the design of Storage Resource Managers (SRMs) in terms of their functionality. (SRMs) are grid middleware components that manage dynamically how client can access and what should reside on the storage resource under their control. We distinguish between two main types of SRMs: Disk Resource Managers (DRMs) that manage access (reading and writing) to a shared disk cache, and Hierarchical Resource Managers (HRMs) that manage access to a disk cache but also manage the access to tape through a mass storage systems, such as HPSS.

We consider two different aspects for the design: read functionality and write functionality. These are discussed in the context of both DRMs and HRMs below. However, we start by describing the basic requirements.

2. Basic requirements

a) SRMs access generality
SRM should support either a direct access by a client's program, or by an "agent program" on behalf of the client. In a grid architecture, the access to multiple files by a user can be delegated to an agent program that communicates to SRMs to coordinate space allocation and file transfer requests. SRMs can queue such request depending on the request load, and call back the agent program when transfer is completed. However, the user should also be able to support "thin clients" that access SRMs directly. In this case, the SRM cannot call back the client. Therefore, a "status" function must be provided. SRMs should also be able to communicate with each other in order to coordinate space allocation and file replication.

b) SRMs should get files when they are not in local disk
When a request for a file is made to an SRM and the file is not found in its disk cache, there are two options: notify the requester that the file is not in local cache (in which case the requester can invoke a grid file transfer call), or the SRM can take upon itself to get the file into its cache from some source location. We chose to make the second approach a requirement, since it simplifies the client's interaction with the grid middleware. In particular, this is an essential functionality for "thin clients" that expect the SRMs to provide the service of getting files they need. An SRM should manage its space, coordinate with the

source site that a file is about to be transferred, invoke the file transfer, and monitor its successful completion.

c) SRMs should support a "push" mode for writes
When a request is made to an SRM to get a file and the file is not in local disk, the SRM needs to first allocate the space for that file, and then either "pull" the file from its source location, or ask the source location to "push" it to its space. "Pull" is performed by an file transfer "get", and "push" is performed by asking the source site to performs a file transfer "put". Normally, only "get/pull" needs to be supported, because a client can always perform a get. However, "get/push" can be useful for caching a large number of files, or when delays are expected. For, example, a client can set a request to an SRM to push some files into its space. The client program can then quit, and the request will be fulfilled by the SRM. On the other hand, if a client program wants to put a file into the SRM disk cache, and it only has a client file transfer program, the SRM must permit the client to push a file to its space. This can be problematic for the SRM, as it has to monitor that the client is not overwriting the allocated space. Normally, it is simpler for clients to have the SRM pull the file when writing to an SRM, but that requires that a file transfer server exists at the client site. The APIs are designed to support both "push" and "pull" modes for both read and writes, but a particular implementation may choose to support only the basic "get/pull" and "put/push" mode.

d) SRM APIs should be uniform for DRMs and HRMs
When requesting a file from a DRM and the file is not in its disk cache, the DRM will request the file from a remote location. This causes a delay. For large files, the transfer may take many minutes or even hours depending on the available network bandwidth. Similarly, when requesting an HRM to get a file and the file is not in its disk, the HRM will schedule a staging request to the MSS to move the file from tape to its disk. This also causes a delay. For large files, this delay may be many minutes as well. In case that the MSS is busy, such a request may even be queued by the HRM, causing further delay. Realizing that from a client's point of view the only difference is the reason for the delay, one can design a uniform API for both DRMs and HRMs although they perform different functions. This design requirement also simplifies the way of communicating with SRMs, as well as SRMs communicating with each other regardless of type of the SRM.

e) SRMs should support pinning of files
SRMs manage what is in their disk cache depending on the popularity of the files and the access patterns to them. Files that can be removed are considered "volatile". If volatile files are to be accessed by clients or other DRMs, then a mechanism that makes sure they files are not deleted when they are about to be transferred or even in the middle of a transfer. This mechanism is called "pinning" a file. Pinning is not an essential requirement, since if a file does not transfer properly or is not found, the requester can request it from another source, or in the worst-case go to the original source where the file is not removable (i.e.

it is “permanent”). However, the operation of the grid is generally better if such mishaps can be avoided. In addition, pinning guarantees to the client that a file will be kept for a period of time. Therefore, we chose to make “pinning” a requirement. If pinning is supported, then it is also necessary to have a “pinning time-out” period. This is needed in order to avoid permanent pinning in case that the file is not released by the client.

3. Functionality

We now describe the functionality of SRMs for the “read” and “write” cases. We discuss the functionality expected from DRMs as well as HRMs.

3.1 Read functionality

Case 1: the file is in the disk cache

Consider a client requesting a file from a DRM. By a client we mean here a user’s program, or a agent program run on behalf of the client. The request should include a “logical file name” (which is a unique grid file name). The DRM may already have the file in its cache. In this case it returns the address of the file in its cache. The client can then read the file directly from the disk cache (if it has access permission), or can “pull” the file into its local disk (using some file transfer service such as FTP). In either case, the DRM will be expected to pin the file in cache for the client for a period of time. This Thus, a “time out” is associated with every file pinned. A well-behaved client will be expected to “release” the file when it is done with it. This case is the same for an HRM as well given that a file if found in its disk cache.

So far we introduced the following concepts:
1) requesting a file;
2) pinning a file;
3) releasing a file;
4) pinning time-out.

Case 2: the file is not in the disk cache

Now, if the client makes a request for a file, and the file is not in disk cache, the SRM will be expected to get the file from its source location. For a DRM this means getting the file from some remote location. Since that may take a relatively long time, the client call is non-blocking. When the file arrives there needs to be a way to notify the client. This requires the function of a call_back in the APIs. In case that the client cannot be called back since it does not have a server, we must provide a “status” function call that the client can use to find out when the file arrives. The status function can return estimates on the file arrival time if the file has not arrived yet.

For an HRM, the action is different. The HRM must maintain a queue for scheduling the file staging from tape to disk by the MSS. This is especially needed if the MSS is busy. If a queue exists, then the HRM puts the request at the end of the queue. Otherwise, it schedules it staging immediately. Like a DRM, and HRM needs to notify the client that the file was staged by issuing a call_back, or the client can find that out by using :status".

How does an SRM know where to get the file from? It is assumed that the client has consulted a Replica Catalog and made a decision where to get the file from. In principle, one can argue that the SRM should perform the function of consulting the replica catalog and making a choice where to get the file from if there are multiple replicas. However, we chose to leave this decision outside the SRM since such decisions may depend on global load balancing decisions. The SRM is also passed the "logical file name" in addition to the location of the source replica. This is because the file may be in cache as a result of another client requesting it earlier. In this case, the logical file name is used to find this out.

3.2 Write functionality

Consider a write request to a DRM first. The DRM has to be given the file size in the write request in order to allocate the space. The next action depends on the mode. If push mode is used, then the location on disk where the file has to be written is returned to the client. When the client finished writing the file (by using a file transfer "put"), it is expected to notify DRM. This brings up several design questions. What if the client writes more than the size provided by the request? What if the client crashes in the middle of the write? What if the client never writes? What if the client does not notify DRM that it finished writing? The implementation design needs to have some mechanisms to support these exceptions. There needs also be some mechanism that files written are not removed prematurely because of time-outs or other conditions. For this reason, we introduced the function requesting DRM to make a file "durable". A durable file is a permanent file under the control of the client. In contrast, "permanent" file are usually the original copies that can only be removed by the dataset administrator. The client can choose to use all of its allocated quota for durable files.

The other option that avoids most of the problems mentioned above is to request writing a file in a pull mode. In this case, the DRM is given the location of the file in the client's space, and it is expected to pull the file. This can be accomplished only if the client has a file transfer server.

In the case of an HRM, the file is first written to its disk cache in exactly the same way as the DRM description above. The HRM notify the client that the file has arrives to its disk using a call_back, then it schedules it to be archived to tape by the MSS. After the file is archived by the MSS, the SRM notifies the client again using a call_back. Thus, the HRM's disk cache is serving as a temporary buffer for files being written to tape.

The write case needs to support similar functions as the read case for releasing a file, aborting a transfer, and responding to “status” calls. But in addition it needs to supprt moving a file from a durable status to a volatile status.

In summary, the following functions are supported by the APIs:
• Request to get a file
—Request_to_get (push/pull modes)
—Release
—Abort
—Status
—Call_back (when file is available)

• Request to put a file
—Request_to_put (push/pull modes)
—Release
—Abort
—Status
—Call_back_1 (when file is transferred to disk)
—Call_back_2 (when file is transferred to tape – for HRM)