Description of SRM interface
and
C++ bindings of SRM IDLs

Arie Shoshani, Alex Sim, Junmin Gu
Scientific Data Management Research Group
Lawrence Berkeley National Laboratory

Table of Contents

Introduction

This document describes a single uniform interface for Storage Resource Managers (SRMs). It is designed so that it can be applied to both Disk Resource Managers (DRMs) and Hierarchical Resource Managers (HRMs). A DRM manages only its own disk cache. A HRM manages both its own disk cache as well as access to a robotic tape through a Mass Storage System (MSS). We can view the access to DRMs and HRMs uniformly since from the requesting client point of view the difference is only in the cause of the delay (or latency) of getting a file. In the case of a DRM, if the file is not available in the disk cache, the delay is caused by the network transfer form a remote location. In the case of a HRM, if the file in not in the local cache, the delay is caused by having to stage it from tape.

By and large, all the function calls and parameters are the same for DRMs and HRMs. However, since a HRM manages both a disk cache as well as access to a robotic tape, there are a few cases that apply only to HRMs. Specifically, there are a few return codes that refer to the state of the MSS (such as MSS_down). Also, in addition to a call_back when a file has been transferred to a disk cache successfully, there is an extra return code for a call_back when a file has been archived by the MSS.

This document contains both CORBA style IDLs for the API, as well as the equivalent C++ API bindings. The IDLs has three main parts: 1) definitions, 2) the SRM function calls, and 3) call_backs. Call_backs are useful when an action may take a while, such as queuing a request when the system is busy. Call_backs can only be used by clients that have CORBA server. Otherwise, clients can find the status of their request by calling the status function. There are also two additional parts of the IDL, one for an administrative use and one for information inquiry about the SRM.

Description of SRM interface IDLs

srmDefs.idl

```
#ifndef SRM_DEFS_IDL
#define SRM_DEFS_IDL

module SRM {
```

- Namespace SRM

```
  exception SRMException {
     string why;
  };
```

- Definition of Exception for SRM

```
  typedef long long TIME_OUT_T;
```

- Definition of time-out in seconds. Time-out is associated with each file, and its value depends on the local policy.

```
  typedef string USER_ID_T;
```

- Definition of User-ID. Currently User ID is a string, and it can be in any form that SRM can understand. For instance, an email address or user account name. Note: This version does not include security input – such as a proxy. If this were added, another variable would be necessary.

```
  typedef string REQUEST_ID_T;
```

- Definition of Request ID. Request ID is created in the client, and passed from the client to SRM. SRM does not generate Request ID for clients. The client can use this Request ID to issue status, release, and abort calls. Also, SRM uses this Request ID for call backs.

```
  typedef string FID_T;
```

- Definition of File ID, a logical file name.

```
  typedef sequence<FID_T> FIDSET_T;
```

- Set of logical file names

```
  typedef string HINT_T;
```

- Definition of hints. This hint is used for site-specific request additions. For instance, the tape ID where a file id stored, or instructions of staging files as a group (i.e. stage all files of the request together).

```
  enum MODE_T {
    PULL, PUSH
  };
```

- Definition of request mode from the file initiator's point of view. The mode types are either PULL or PUSH. Its usage is described later in this document.

```
  struct REQUEST {
      FID_T fid;
      string source_url;
      string target_url;
```

```
    long long size;
    HINT_T hint;
};
```

- Definition of a file request. It consists of logical file name, Source file name (site-specific file name), Target file name (destination file name), File size in bytes, Hint about the file. The target file name can be null depending on the mode type.

```
typedef REQUEST REQUEST_T;
typedef sequence<REQUEST_T> REQUEST_SET_T;
```

- Definition of Set of file requests

```
enum RETURN_CODE_T {
```

- Definitions of return codes

```
  FILE_REQUEST_QUEUED,
```

- It is used when the request is queued. This code is used as one of the return codes in request_to_get, request_to_put or status.

```
  FILE_TRANSFER_IN_PROGRESS,
```

- It is used when the file transfer is in progress.

```
  FILE_PINNED,
```

- It is used when file is pinned (reserved) for reading. With the case of request_to_get with PULL mode, it means that the file is at the destination (possibly after a transfer from another site) and is pinned for the client . This code can be as a result of request_to_get, and also, it can be used in the callback to the client from SRM.

```
  FILE_IN_CLIENT_CACHE,
```

- It is used when the file gets transferred successfully to the client's destination, in case of request_to_get with PUSH mode.   This code is also used by status or a call back after the transfer takes place.

```
  FILE_IN_SRM_CACHE,
```

- It is used when the file gets transferred successfully from client's source location to SRM's cache, in case of request_to_put with PULL mode.
- It can represent the second return code after SPACE_ALLOCATED, in response to "request_to_put" with PUSH mode. With the "request_to_put" with PUSH mode, SRM will allocate the space and return SPACE_ALLOCATED to the client. Upon the returned message, the client will start transferring the file. After the file transfer is done, client is expected to call "file_transfer_done" to SRM (but in some cases, it may not be always true). SRM may call back to the client with FILE_IN_SRM_CACHE to confirm the file transfer.

```
  FILE_ARCHIVED,
```

- It means that the file is successfully archived on MSS. When the file is migrated to MSS, in response to request_to_put, it is returned to the client after the successful migration. This response is only valid for a HRM.

```
  SPACE_ALLOCATED,
```

- When the client "request to put" with PUSH mode, space on SRM needs to be reserved for the file transfer. After the space for the particular file is allocated, this value can be returned.

```
  FILE_TIMED_OUT,
```

- It is used when the file is pinned, it is associated with time-out value, and file "status" will show how much time left on the file. The time-out value depends on the local policy.

```
  SPACE_TIMED_OUT,
```

- It is used when the allocated space is timed out.

```
  FILE_SIZE_MISMATCH,
```

- It is used when the expected file size is different from the actual file size.

```
  NOT_ENOUGH_SPACE,
```

- It is used when there is a file request and space cannot be allocated due to the not-enough space, this value is returned. It can be treated as one of request failures or FILE_REQUEST_QUEUED, although being handled as FILE_REQUEST_QUEUED is desirable.

    - If it is handled as a failed file request, the client can either re-submit the request later, or find some other SRM to use.
    - If it is handled as FILE_REQUEST_QUEUED, this is just an advisory, and the client can either abort the request, or wait until file request gets fulfilled. "status" will show any time estimates if possible.

```
  USAGE_LIMIT_REACHED,
```

- It is used when requested files exceed the limit on the number of files or the limit on the spaces allocated for the user by the policy. It can be treated as one of request failures or FILE_REQUEST_QUEUED, although being handled as FILE_REQUEST_QUEUED is desirable.
    - If it is handled as a failed file request, the client can either re-submit the request later, or find some other SRM to use.
    - If it is handled as FILE_REQUEST_QUEUED, this is just an advisory, and the client can either abort the request, or release some of his requests. "status" will show any time estimates if possible.

```
  FILE_DOES_NOT_EXIST,
```

- It is used when the requested file does not exist in the source.

```
  DUPLICATE_REQUEST_ID,
```

- For duplicate request ID.

```
  USER_NOT_AUTHORIZED,
```

- It is used when the user is not authorized to request files. Also, when the user does not have read or write permissions on the source or target file respectively, this value gets returned.

```
  MSS_DOWN,
```

- When MSS is down, this value returned. It is advisory purpose only, and there is no expected actions from the client. HRM will retry the request. Then, the client can abort the request. This response is only valid for a HRM.

```
  MSS_ERROR
```

- When MSS is in error, this value gets returned, and It is advisory purpose only, but some actions from the client is desirable, such as aborting the request and changing the source. This response is only valid for a HRM.

```
};

typedef sequence<string> PROTOCOL_SET_T;
```

- Definitions of the set of transfer protocols

```
struct RETURN_STATUS {
    FID_T fid;
    RETURN_CODE_T code;
    string code_explanation;
};
```

- Definitions of the return status; it consists of logical file ID, return code, some detailed explanation of code

```
typedef RETURN_STATUS RETURN_STATUS_T;
typedef sequence<RETURN_STATUS_T> RETURN_STATUS_SET_T;
```

- Definition of the set of return status

```
struct FILE_STATUS {
    FID_T fid;
    string return_url;
    long long time_to_service;
    TIME_OUT_T time_out;
    long long current_file_size;
    long long expected_file_size;
    boolean in_srm_cache;
    RETURN_CODE_T code;
    string code_explanation;
```

```
  };
```

- Definition of file status; it consists of
    - logical file ID,
    - Return URL (transferURL with protocol in EU): This is empty when the file has not been ready yet,
    - Time estimates to fulfill the request in seconds (This value is 0 when the file request is fulfilled),
    - Time-out associated with the pinned file (This value is 0 when the file request is not fulfilled yet),
    - Current File Size in bytes,
    - Expected File Size in bytes that client passed to SRM at the request time,
    - Boolean value for indicating if the file is in SRM cache,
    - Return code,
    - Detailed explanation about the code, when needed.

```
  typedef FILE_STATUS FILE_STATUS_T;
  typedef sequence<FILE_STATUS_T> FILE_STATUS_SET_T;
```

- Definition of the set of file status

```
};


#endif
```

srm.idl

```
#ifndef SRM_IDL
#define SRM_IDL

#include <srmDefs.idl>
#include <srm_call_back.idl>

module SRM {
  interface SRMServant {
```

- Definition of SRMServant class

```
    string systemStatus() raises (SRMException);
```

- System status returns current information available.

```
    FILE_STATUS_SET_T request_to_get(
                        in USER_ID_T uid,
                        in REQUEST_ID_T request_id,
                        in client_call_back ref,
                        in PROTOCOL_SET_T pset,
                        in REQUEST_SET_T fset,
                        in MODE_T mode)
                        raises (SRMException);
```

❖ Request to get files: requires
- User ID,
- Request ID,
- Client callback reference (If client does not support callbacks, this may be an optional argument. In that case, client should check the status for the result),
- Set of protocols that client can support (this set of protocols can be negotiated for the return URL later (this is referred as "transport URL" by the EU Datagrid)),
- Set of file requests
- Request mode (default is pull. In pull mode, the target is not defined; it is assigned by the SRM locally, and returned as return URL. In push mode, the target needs to be defined.)

```
    FILE_STATUS_SET_T request_to_put(
                        in USER_ID_T uid,
                        in REQUEST_ID_T request_id,
                        in client_call_back ref,
                        in PROTOCOL_SET_T pset,
                        in REQUEST_SET_T fset,
                        in MODE_T mode)
                        raises (SRMException);
```

❖ Request to put file: requies
- User ID,
- Request ID,
- Client callback reference (if the client does not support callbacks, this is an optional argument. In that case, the client should check the status to check if the request executed properly),
- Set of protocols that the client can support (this set of protocols will be used when the mode is pull (SRM initiates the file transfer from the source in the client's domain)),
- Set of files request (Definitions of the file request is defined in srmDefs.idl.),
- Request mode. The default mode is push. In push mode the size is required. In case of a DRM, the target is not defined; it is assigned by the DRM locally, and returned as return URL with protocol. Then the client can push (transfer) the files to the target (returned) URL disk. In case of a HRM, the target URL is required, since HRM needs to know where to archive the file. However, HRM still returns a disk return URL to be used for pushing the file to its disk. Note: This "push" mode requires the monitoring of the file sizes at the target location by SRM.

In general, it is desirable to use "pull" mode. In the case of a HRM, when we use pull mode, the source file will be transferred by HRM to the HRM managed disk location, and its location will be passed to the client as a return URL.

```
    void abort( in USER_ID_T uid,
                in REQUEST_ID_T request_id,
                in FIDSET_T fset)
                raises (SRMException);
```

- Abort request: requires User ID, Reqeust ID, Set of logical file IDs.

```
    void release( in USER_ID_T uid,
                  in REQUEST_ID_T request_id,
                  in FIDSET_T fset)
                  raises (SRMException);
```

- Release request: requires User ID, Request ID, Set of logical file IDs.

```
    FILE_STATUS_SET_T status(in USER_ID_T uid,
                             in REQUEST_ID_T request_id,
                             in FIDSET_T fset)
                             raises (SRMException);
```

- Status request: requires User ID, Request ID, Set of logical file IDs, Returns the status of each file, containing time estimates, time-outs, and return URL.

```
    PROTOCOL_SET_T getSupportedProtocols() raises (SRMException);
```

- getSupportedProtocols returns the set of protocols that SRM supports for file transfers.

```
    FILE_STATUS_T file_transfer_done(in USER_ID_T uid,
                                     in REQUEST_ID_T request_id,
                                     in FID_T fid)
                                     raises (SRMException);
```

- File_transfer_done is called by the client when the request_to_put with push mode is approved for a file, and after the client completed its transfer to the target location. It requires User ID, Request ID, Logical file ID

```
    boolean set_file_durable(in USER_ID_T uid, in FID_T fid);
```

- For setting a file to be durable in SRM cache. It requires the user ID and the file name.

```
    boolean set_file_volatile(in USER_ID_T uid, in FID_T fid);
```

- For setting a file to be volatile in SRM cache. It requires the user ID and the file name.

```
  };
};

#endif
```

srm_call_back.idl

```
#ifndef SRM_CALL_BACK_IDL
#define SRM_CALL_BACK_IDL

#include <srmDefs.idl>

module SRM {
    interface client_call_back {
```

- Definition of client_call_back class

```
        oneway void message(in REQUEST_ID_T request_id,
                            in FILE_STATUS_SET_T status_set);
```

- Callback message: It is one way call so that SRM server does not depend on the result of the call. It requires Request ID, Set of file status

```
        boolean areyoualive() raises (SRMException);
```

- Areyoualive returns a Boolean value. It is a way of checking if client callback works.

```
    };
};

#endif
```

srm_info.idl

```
#ifndef SRM_INFO_IDL
#define SRM_INFO_IDL

#include <srmDefs.idl>

module SRM {
  interface SRMInfo {
      ❖      Definition of SRMInfo class
    double getTransferRate() raises (SRMException);
           o  For transfer rate in bytes/sec
    double getTransferTimeEstimate(in FIDSET_T fset) raises (SRMException);
           o  For transfer time estimate on a file in seconds, regardless of the request ID. This could be for
              a file that has already been requested, or for a file that is expected to be requested.
    short getNumberTransferPending() raises (SRMException);
           o  For the number of pending transfers in SRM
    short getNumberTransferAllowed() raises (SRMException);
           o  For the limit on the number of transfers in SRM (depending on the system)
    short getNumberRequestsQueued() raises (SRMException);
           o  For the number of total requests on the SRM’s queue
    long long getCacheSizeUsed() raises (SRMException);
           o  For the size of the used cache
    long long getCacheSizeAllocated() raises (SRMException);
           o  For the size of the allocated cache
  };
};

#endif
```

srm_admin.idl

An administrative interface is not required as a common interface because administrative functionalities may be different depending on the site and the implementation of SRM itself. The following is what the LBNL implementation of SRM uses to control its internal values and to collect some information about the system.

```
#ifndef SRM_ADMIN_IDL
#define SRM_ADMIN_IDL

#include <srmDefs.idl>
#include <srm.idl>
#include <srm_info.idl>

module SRM {
      interface SRMAdmin : SRMServant, SRMInfo {
            struct FSIZE_T {
                  FID_T fid;
                  long long fsize;
            };
```
- Simple structure that consists of logical file name and its file size in bytes

```
            typedef sequence<FSIZE_T> FSIZE_SET_T;
            boolean setFilePermanent(in USER_ID_T uid, in FID_T fid);
```
- For setting a file to be permanent in SRM cache. It requires the administrator’s ID and the file name.

```
            double getDefaultTransferRate();
```
- For getting the default transfer rate in SRM. Default transfer rate is needed for transfer estimation. Transfer rate is dynamically adjusted as transfers go on.

```
            void setDefaultTransferRate(in double dtr);
```
- For setting the default transfer rate in bytes/sec.

```
            void setFileClusteringByTapeFactor(in short fcbt);
```
- For setting the number of file cluster value. It is used for efficient tape mountings.

```
            void setNumberTransferAllowed(in short npa);
```
- For setting the number of network connections for file transfer.

```
            double getLastTransferOverhead();
```
- For getting the last transfer overhead

```
            boolean reclaimCacheSize(in double rcs);
```
- In case the SRM cache size needs to be adjusted, the new cache size in bytes needs to be passed to this interface.

```
            void currentCachedFileList(out FSIZE_SET_T fset);
```
- For the list of the files in SRM cache

```
            boolean makeDirectoryOnMSS(in string path);
```
- SRM assumes the target path already existing prior to the file migration. If not, SRM will return USER_NOT_AUTHORIZED. To create a directory on MSS, client needs to access to the MSS by themselves, or access this interface and SRM creates the path on behalf of the client, including subdirectories recursively.

```
      };
};

#endif
```

RETURN CODE description

The return codes described in srmDefs.idl can be organized in the following hierarchical form:

1) RETURN_CODE_T is used in three aspects:
   a) return values for users calls to request_to_get or request_to_put.
      - REQUEST_QUEUED (which means either request is queued or is being processed)
      - FILE_TRANSFER_IN_PROGRESS which means the request is being processed
      - Request was completed successfully
        - for request_to_get with pull mode, FILE_PINNED may be returned.
        - for request_to_get with push mode, FILE_IN_CLIENT_CACHE may be returned.
        - for request_to_put with pull mode, FILE_IN_SRM_CACHE may be returned when SRM manages disks, or FILE_ARCHIVED when HRM manages MSS.
        - For request_to_put with push mode, SPACE_ALLOCATED may be returned first, and after file transfer is completed by the client, FILE_IN_SRM_CACHE and/or FILE_ARCHIVED may be returned by a HRM.
      - Request cannot be fulfilled and it could mean one of the following:
        - FILE_DOES_NOT_EXIST
        - USAGE_NOT_AUTHORIZED
        - <others>
   b) return values for calls to status .
      - if the request is DONE or FAILED, then same as return value as in a).
      - if the file is successfully transferred, but its transferred file size does not match as expected, then FILE_SIZE_MISMATCH would be returned. It depends on the local policy implementation of SRM to keep or remove the size-mismatched file.
      - if the request is QUEUED, in addition to REQUEST_QUEUED, we may return:
        - MSS_DOWN
        - MSS_ERROR
      - The following code can be handled in two ways, when either the request is queued or the request is failed:
        - USAGE_LIMIT_REACHED (which may put the request in the queue of user resources.)
        - NOT_ENOUGH_SPACE (which may cause a request to be put back of the queue)

        The actual handling of these cases depends on the local policy.
   c) callback values to clients .
      - if request is DONE or FAILED, then same as return value as in a).
      - if the file is successfully transferred, but its transferred file size does not match as expected, then FILE_SIZE_MISMATCH would be returned. It depends on the local policy implementation of SRM to keep or remove the size-mismatched file.
      - if request is QUEUED, in addition to REQUEST_QUEUED, we may return:
        - MSS_DOWN
        - MSS_ERROR
      - The following code can be handled in two ways, when either the request is queued or the request is failed:
        - USAGE_LIMIT_REACHED (which may put the request in the queue of user resources.)
        - NOT_ENOUGH_SPACE (which may cause a request to be put back of the queue)

2) Exception can be applied in addition to return code:
   - DUPLICATED_REQUEST_ID (if one or more users have active requests with the same id)

Description of C++ API bindings of SRM IDLs

srm_client_errors.h

```
// error codes for various states of SRM

#define    SRM_FILE_QUEUED 0
#define    SRM_FILE_IN_CLIENT_CACHE 1
#define    SRM_FILE_IN_SRM_CACHE 2
#define    SRM_NOT_ENOUGH_SPACE 3
#define    SRM_FILE_DOES_NOT_EXIST 4
#define    HRM_MSS_DOWN 5
#define    HRM_MSS_ERROR 6
#define    SRM_FILE_TIMED_OUT 7
#define    SRM_FILE_PINNED 8
#define    SRM_USAGE_LIMIT_REACHED 9
#define    SRM_USER_NOT_AUTHORIZED 10
#define    SRM_FILE_ARCHIVED 11
#define    SRM_SPACE_ALLOCATED 12
#define    SRM_SPACE_TIMED_OUT 13
#define    SRM_FILE_SIZE_MISMATCH 14
#define    SRM_FILE_TRANSFER_IN_PROGRESS 15
```

srm_client_attr.h

```
#include <iostream.h>
#include <vector>
#include <srm_client_errors.h>
#include <srm_client.h>
#include <srmDefs.h>
#include <srm.h>

class srm_client {
public:
      srm_client(bool debug=false);
      ~srm_client();
      bool setUserInfo(char* uid);
      bool setRequestType(SRM_CLIENT_REQUEST_TYPE t=SRM_READ,
                          SRM_CLIENT_REQUEST_TYPE_MODE m=PULL);
      bool addRequestProtocol(SRM_CLIENT_PROTOCOL_TYPE p=GSIFTP);
      int addFileToRequest(char* logical_file_name,
                        char* source_url,
                        char* target_url,
                        long long size,
                        char* hint);


private:
      bool verbose;
      char* userID;
      SRM_CLIENT_REQUEST_TYPE type;
      SRM_CLIENT_REQUEST_TYPE_MODE mode;
      std::vector<SRM_CLIENT_PROTOCOL_TYPE> protocols;
};
```

srm_client.h

```
// request type: read or write
enum SRM_CLIENT_REQUEST_TYPE {
      SRM_READ, SRM_WRITE
};

// request mode: pull or push
enum SRM_CLIENT_REQUEST_TYPE_MODE {
      SRM_PULL, SRM_PUSH
};

// protocols
enum SRM_CLIENT_PROTOCOL_TYPE {
      GSIFTP, FTP, HTTP, HTTPS, FILE, SCP
};

class srm_client;

// SRM CLIENT API initialization
bool srm_client_init();
// SRM CLIENT API destruction
bool srm_client_destroy();
// SRM CLIENT HANDLE initialization with verbose flag
bool srm_client_attr_init(srm_client*& handle,
                          bool verbose=false);
// SRM CLIENT HANDLE destruction
bool srm_client_attr_destroy(srm_client* handle);
// SRM CLIENT to set the user information
bool srm_client_set_user_info(srm_client* handle,
                                    char* userID);
// SRM CLIENT to set the request type and mode
bool srm_client_set_request_type(srm_client* handle,
                                    SRM_CLIENT_REQUEST_TYPE type,
                                    SRM_CLIENT_REQUEST_TYPE_MODE mode);
// SRM CLIENT to set the request protocols
//     can be called multiple times so that more protocols can be handled
bool srm_client_add_request_protocol(srm_client* handle,
                        SRM_CLIENT_PROTOCOL_TYPE protocol);
// SRM CLIENT to add file information to the request
//     can be called multiple times
//     will return the number of files in the request each time
int srm_client_add_file_to_request(srm_client* handle,
                        char* logical_file_name,
                        char* source_url,
                        char* target_url,
                        long long size,
                        char* hint);
// SRM CLIENT to execute the request to SRM
char* srm_client_send_request(srm_client* handle);
// SRM CLIENT to add files to the list of abort call
//    this list does not have to be called at all, and in that case
//    all the files in the request will be aborted
int srm_client_add_file_to_abort_list(srm_client* handle,
                                    char* logical_file_name);
// SRM CLIENT to execute abort to SRM
bool srm_client_abort_request(srm_client* handle);
```

```
// SRM CLIENT to add files to the list of release call
//    this list does not have to be called at all, and in that case
//    all the files in the request will be released
int srm_client_add_file_to_release_list(srm_client* handle,
                                    char* logical_file_name);
// SRM CLIENT to execute release to SRM
bool srm_client_release_request(srm_client* handle);
// SRM CLIENT to add files to the list of status call
//    this list does not have to be called at all, and in that case
//    all the files in the request will be asked
int srm_client_add_file_to_status_list(srm_client* handle,
                                    char* logical_file_name);
// SRM CLIENT to call request status to SRM
//    status result can be reviewed from the one of the handle's functions
bool srm_client_status(srm_client* handle);
```