Experiences in testing a Grid service in a production environment

Flavia Donno, Andrea Domenici

Experiences in testing a Grid service in a production environment

Flavia Donno
CERN, European Organization for Nuclear Research,
CH-1211 Geneva 23,
Switzerland.

Andrea Domenici
DIIEIT, University of Pisa,
v. Diotisalvi 2, I-56122 Pisa,
Italy.

***Abstract*—This paper reports on the problems and solutions related to the testing of several implementations for conformance to a standard storage management interface adopted in the Worldwide LHC Computing Grid.**

***Index Terms*—Grid, testing, Storage Resource Manager.**

I. Introduction

The Worldwide LHC Computing Grid (WLCG) is one of the largest Grid infrastructures dedicated to high-performance scientific computation, with more than 200 sites all over the world. Its data storage facilities, expected to sustain an amount of data in the order of 10–15 Petabytes per year, are heterogenous Mass Storage Systems (MSS), each based on different technologies and with different capabilities and interfaces. In spite of this diversity, Grid applications and Grid middleware expect to handle all data in a uniform manner across the Grid. It has then been necessary to develop a standard interface for all MSSs, called the *Storage Resource Manager (SRM) Interface* [1]. Existing MSSs have been adapted to SRM specification, and new ones have been developed anew according to the specification.

Testing MSSs for compliance to the SRM specification has been problematic. The specification defines an Application Programming Interface (API) comprising a large set of operations that often have many parameters of complex type. The semantics of this API are specified informally and subsume a complex behavior that must be tested thoroughly. Given the complexity of the specification, however, exhaustive testing is very time-expensive, in a production environment where storage resources are continually in use.

This paper reports on the various techniques used to test implementations of the SRM service and shows how the application of standard testing methodologies has been guided by specific knowledge of the operational environment where the systems under test are deployed.

In the following sections we introduce the basic concepts of the SRM (Sec. II), then in Sec. III we illustrate the ideas behind the testing of the SRM service, in Sec. IV we introduce the S2 testing language, and in Sec. V we summarize the results of the testing activity.

II. Grid Storage Elements

A *Storage Element* (SE) is a Grid Service implemented on a mass storage system (MSS) that may be based on a pool of disk servers, on more specialized high-performing disk-based hardware, or on a disk cache front-end backed by a tape system, or some other reliable, long-term storage medium. Remote data access is provided by a GridFTP service [2] and possibly by other data transfer services, while local (intra-cluster) access is provided by POSIX-like input/output calls. Authentication, authorization and audit/accounting facilities are also part of a SE, that uses proxy certificates handled by the Virtual Organization Management Service.

A Storage Element provides *spaces* where users create and access *files*. Storage spaces may be of different qualities, related to reliability and accessibility, and support different data transfer protocols. Different users may have different requirements on space quality and access protocol, therefore, in addition to the basic data transfer and file access functions, a SE must support more advanced resource management services, including dynamic space allocation.

Each SE supports the SRM service specification, whose main specification documents are *The Storage Resource Manager Interface Specification, Version 2.2* [1] and the *Storage Element Model for SRM 2.2 and GLUE schema description* [3]. Other relevant documents are [4], [5], [6], [7], [8], [9].

A. The Storage Resource Manager service requests

The *SRM Interface Specification* lists the service requests that a client application may issue, along with the data types for their arguments and results.

Request signatures are given in an implementation-independent language and grouped by functionality: *Space management* requests allow the client to reserve, release, and manage spaces, specifying or negotiating their qualitiy and lifetime; *Data transfer* requests have the purpose of getting files into SRM spaces either from the client's space or from other remote storage systems on the Grid, and to retrieve them; *Directory* requests create, populate, list, or delete directories; *Permission* requests set or list read and write permissions on files and directories; and finally, *Discovery* functions allow applications to query the availability and characteristics of the storage system behind the SRM interface.

Most SRM requests have a number of optional input and output parameters. The 'optional' qualification may have different meanings for different parameters and different requests:

for an input parameter, it may mean that the parameter has an implementation-dependent default value, or that it may be ignored by the implementation, possibly because the parameter refers to an unsupported feature; for an output parameter, it may mean that the implementation is allowed not to return a value, or that the value is not needed for some choices of input values.

Several requests accept or return one or more *SURL*'s as parameters. A SURL (*Site URL*) is a string that identifies a file, composed of the hostname of the Storage Element and of the file's name within the Storage Element.

All requests have a mandatory `returnStatus` output parameter, that reports about success or cause of failure for each request. It may also report on the state of advancement in the execution of *asynchronous* requests. The SRM processes asynchronous requests by queueing them and returning a *request token* to the client. The client, by passing the request token to other SRM calls, may subsequently query the system about request completion, or abort the request. This usage is common when a single request acts on several files or spaces. In this case, a request may return not just one return status but an overall *request-level* status plus a specific *file-level* status for each file.

B. Storage space properties

The execution of most SRM operations depends on a few properties of the involved files and of the spaces where their *copies* (i.e., possibly replicated physical instances) reside. In this paper we use as an example an SRM request that involves spaces and not files, so we only need to consider the property of *storage class*, that is defined in terms of two other properties, *retention policy* and *access latency*, and the properties of *connection type* and *access pattern*.

The properties of retention policy and access latency may or must be specified for most SRM requests involving the reservation or creation of spaces and files. Retention policy describes the reliability of a storage medium, while access latency says if data are immediately accessible or must be staged from a slow medium (e.g., tape) to a faster one.

Retention policy is a qualitative indication of the likelihood that a file copy may be lost in a given storage space. This likelihood may be high, intermediate, or low. A space with a high likelihood of file loss is said to have a *Replica* retention policy, since it is satisfactory for replicated files that can be accessed with a limited performance penalty if a single copy is lost. A space with an intermediate likelihood of file loss has an *Output* retention policy, since it is satisfactory for files that are not replicated but can be recreated as the output of some computation. Finally, the *Custodial* retention policy applies to storage that has a low likelihood of file loss, and is therefore appropriate for files whose recovery would be very costly or even impossible.

Access latency is a classification of storage media according to the timeliness of data access. A space where data are immediately accessible is *Online*, otherwise it is *Nearline*. A *Nearline* space is supported by a medium, such as tape or DVD libraries, that uses mechanical operations to retrieve the data, that are then staged to temporary disk storage.

In the WLCG only a few combinations, referred to as *storage classes*, of the above properties are supported, and they are called *Tape0Disk1* (*Replica*, *Online*), *Tape1Disk1* (*Custodial*, *Online*), and *Tape1Disk0* (*Custodial*, *Nearline*).

The properties of connection type and access pattern are, strictly speaking, properties of the client application and its connection to the SRM. In fact, the connection type, with values *Wan* and *Lan*, "indicates if the client is connected through a local or wide area network" [1] while the access pattern, with values *Transfer_mode* and *Processing_mode*, indicates whether the client is going to transfer a whole file in one operation (*transfer mode*) or access the file with several distinct operations (*processing mode*). However, this information is also passed to requests that reserve space before data transfers take place, so that the reserved space may be taken from a medium optimized for the specified connection type and access pattern.

III. Testing Storage Resource Manager service implementations

To illustrate the problem of testing an SRM service, let us consider one typical SRM request, `srmReserveSpace`, whose behavior is described in details in [1]. Fig. 1 shows the signature of the request, where the **boldface** parameters are mandatory.

Input parameters:

string	authorizationID
string	userSpaceTokenDescription
TRetentionPolicyInfo	**retentionPolicyInfo**
unsigned long	desiredSizeOfTotalSpace
unsigned long	**desiredSizeOfGuaranteedSpace**
int	desiredLifetimeOfReservedSpace
unsigned long[]	arrayOfExpectedFileSizes
TExtraInfo[]	storageSystemInfo
TTransferParameters	transferParameters

Output parameters:

TReturnStatus	**returnStatus**
string	requestToken
int	estimatedProcessingTime
TRetentionPolicyInfo	retentionPolicyInfo
unsigned long	sizeOfTotalReservedSpace
unsigned long	sizeOfGuaranteedReservedSpace
int	lifetimeOfReservedSpace
string	spaceToken

Fig. 1. The signature of `srmReserveSpace`.

To assess, with some approximation, the number of test cases that may be required, we start with a straightforward application of equivalence partitioning.

With this technique, the tests must cover all the values of the arguments with a finite domain. The values of such arguments, in fact, represent distinct behaviors, policies, or operating conditions of the system, while the arguments whose values range over potentially infinite domains only represent physical attributes of some entity, such as, for example, the

size or lifetime of a space. Such values do not affect the system's functionality, and need to be considered only for stress testing (to find implementation-dependent limitations) and performance testing. Therefore, we need consider only one valid value for each argument with an infinite domain. Similarly, one single value is representative of a class of invalid values.

Further, we must take into account the fact that the domain of some arguments may be the union of a theoretically infinite set (e.g., positive integers for space lifetime) and a finite set of special values, such as -1 to signify an unbounded lifetime. Finally, we must consider the absence of an argument from a request, either because the argument is optional, or because the request is ill-formed. Special arguments and the absence of an argument, denoted as NULL, are considered as extra values.

A test case for a request with N arguments is therefore an N-tuple of values, i.e., an element of the Cartesian product of the sets of values that each element may take, where these values are drawn from the union of the equivalence classes described above. We adopt the strategy of separating *clean* tests, expected to produce valid results, from *dirty* tests, expected to produce (correct) error codes. For the latter group of tests, we consider only tuples of invalid values, since valid arguments would not affect the test reults. In this way we rule out a large number of "mixed" test cases, i.e., containing both valid and invalid argument values. Such test cases are only useful when certain valid values for an argument are incompatible with values of other arguments, but these situations are rare and they can be ignored when extimating the needed number of test cases.

We may now define four equivalence classes for the values of each argument: (1) *valid* values; (2) *well-formed invalid* values (e.g, values not supported by an implementation); (3) *ill-formed invalid* values; and (4) the class consisting of the NULL value for mandatory arguments.

We observe that class 1 includes the NULL value, unless an argument is mandatory. Class 4 covers the cases where a mandatory argument is absent.

Let us consider class 1. Let $\mathcal{D}_i$ denote the domain of the i-th argument, and let $V_i = |\mathcal{D}_i|$ (the cardinality of $\mathcal{D}_i$) if $\mathcal{D}_i$ is finite, or $V_i = 1$ otherwise. V_i is the number of valid values that must be tested for the i-th argument. In order to consider the absence of an optional argument, we introduce V_i^*, that is equal to $V_i + 1$ for optional arguments, or to V_i otherwise. The number of possible test cases for class 1 is then $\prod_{i=1}^{N} V_i^*$.

Then we consider classes 2, 3, and 4. These classes are used for dirty testing, so their elements will not be combined with values of class 1 to form test cases. Since the range of each argument is reduced to three distinct values, one from each of the three classes, the number of these test cases is 3^N.

Adding the two terms we obtain

$$\prod_{i=1}^{N} V_i^* + 3^N \quad . \tag{1}$$

For the `srmReserveSpace` request, we study its signature as defined in the SRM specification, and by the criteria introduced above we find that V_i^* for its nine arguments ranges from 1 to 6, giving a number of test cases above twenty thousand. Considering that the SRM interface lists thirty-nine requests with an average of six input parameters, to a maximum of ten, it is clearly necessary to find a way to reduce the test case space to a manageable size without reducing its efficacy. In the rest of this section we discuss how to select a manageable number of test cases by a careful analysis of the actual operating conditions of the SRM service.

A. Reducing the test set size by use-case analysis

In order to reduce the number of possible test cases to execute, we made a few assumptions justified by the requirements of the WLCG users. These assumptions enable us to reduce the number of parameters whose combined values make up the input domain, with the remaining parameters set at fixed values, thus lowering the dimensionality of the test case space.

In order to reduce the size of the input domain for the `srmReserveSpace` request, we examine the requirements from the LHC experiments and determine the set of input arguments that are commonly needed by users and that most implementations do not ignore. First of all, this function must guarantee a given amount of space (`desiredSizeOfGuaranteedSpace`) of a certain quality (`retentionPolicyInfo`) for the required amount of time (`desiredLifetimeOfReservedSpace`). These three parameters are then the most relevant for testing.

As mentioned in Sec. II-B, only three storage classes are required to be supported in WLCG, so that the structured value `retentionPolicyInfo` can only be one of the pairs (*Replica*, *Online*), (*Custodial*, *Online*), and (*Custodial*, *Nearline*), plus the implementation-dependent system defaults (*Custodial*, NULL) and (*Replica*, NULL).

We now consider the `transferParameters` input. This parameter has a structured value that combines connection type and access patterns (Sec. II-B), plus some additional information on site connectivity. These parameters affect critically the performance of data transfers, therefore, even if the SRM specification assumes that they may be negotiated between clients and service, it is very unlikely that input and output buffers and specific protocols are left to the users to define and allocate during an `srmReserveSpace` call. Normally, Grid centers will statically configure space buffers with predetermined characteristics (such as protocols, connectivity type, etc.) that applications can reserve. Therefore, if users request a certain combination of `retentionPolicyInfo` and `transferParameters` for which no space buffer has been allocated at a certain location, the SRM server will just ignore the `transferParamenters` value.

The `authorizationID` is foreseen in order to specify a user identity and allow the system to grant permission for space reservation. However, in WLCG such a parameter is not used by any of the implementations taken into consideration since they all rely on the VOMS that handles authorization

without any need of passing credentials explicitly as argument values, since the role or group of a user is derived from his proxy. In WLCG only VO managers can reserve space and make it available to users of a VO. Therefore, the VOMS proxy is checked and appropriate privileges to reserve space are guaranteed. We may re-interpret Formula 1, that had been derived considering only explicit arguments, by representing environmental conditions, such as the presence of a valid proxy, as extra arguments.

For the user proxy we consider the following cases: (1) the user has a valid proxy that allows him/her to reserve space; (2) the user has a proxy that does not allow him/her to reserve space; and (3) the user does not have a valid proxy. Case 1 represents the equivalence class of valid values, while in cases 2 and 3 the SRM server returns error status codes for authorization and authentication failure, respectively. Since both error responses must be tested, we have two classes of invalid values. Therefore, in order to perform dirty testing we must consider 3^4 combinations for the four explicit arguments, times two possible invalid values for the *user proxy* pseudo-argument.

By the above considerationse, the only input parameters and conditions to be considered are `retentionPolicyInfo`, `desiredSizeOfGuaranteedSpace`, `desiredLifetimeOfReservedSpace`, `transferParameters`, and the user proxy.

Then we re-examine the possible values of `retentionPolicyInfo`. The Tape0Disk1 storage class must be supported by all implementations and therefore it is a case to be considered. Classes Tape1Disk0 or Tape1Disk1 are equivalent since the implementation can either support them or not. The cases (*Custodial*, NULL) or (*Replica*, NULL) are also equivalent since the server can either provide a default for the Access Latency or not. In the second case, the server must return an error status. Taking these considerations into account, we can limit ourselves to consider only four cases for `retentionPolicyInfo`, namely (*Replica*, *Online*), (*Custodial*, *Nearline*), (*Replica*, NULL), and NULL, i.e., the absence of the parameter.

Argument `desiredSizeOfGuaranteedSpace` is mandatory, therefore it cannot be NULL (invalid condition). Its value must be greater than zero and can be as great as allowed by the *unsigned long* type.

For the `transferParameters` input parameter, a similar analysis can be done.

As a result of the above consideration, we find out that the number of test cases can be safely reduced to about two hundred.

B. Modeling constraints and boundary conditions

In Sec. III-A we have dealt with the selection of test cases based on the signature of the service requests. However, we still have not considered the restrictions imposed by the specifications and the semantics of the function to validate. For instance, the `srmReserveSpace` method can be asynchronous (Sec. II-A). In this case the returned request token can be used as an input to the `srmStatusOfReserveSpaceRequest` request to check the status of the operation. The specification restrictions and the semantics of the request are expressed using cause-effect graphs. In Fig. 2 we list the causes and effects identified. In particular, the causes taken in consideration are numbered from 1 to 13, while the effects are numbered from 90 to 93.

As it can be noted, the list of causes does not include only conditions on the input arguments, as it normally happens in cause-effect analysis: Some environmental conditions (e.g., cause 2 in the table) are also taken into consideration in order to represent the restrictions imposed by the protocol.

Fig. 3 represents the analysis of the semantic content of the specification transformed into a Boolean graph linking causes and effects. This is the cause-effect graph for the `srmReserveSpace` function.

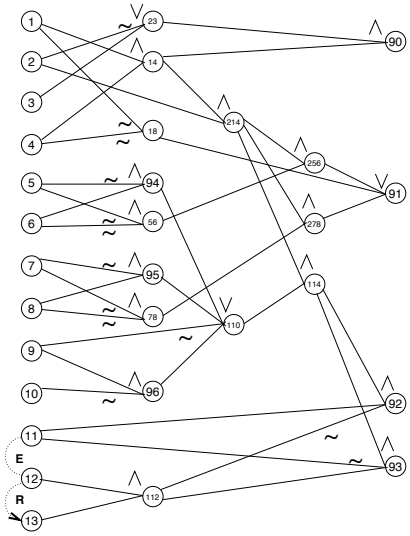

Fig. 3. Cause-effect graph for the `srmReserveSpace` request.

The graph nodes not listed in Fig. 2 (e.g., 56, 78, ...) have been created to facilitate the navigation of the graph. Nodes 94, 95, and 96 instead represent effects foreseen by the specification that must be checked explicitly by a specific test program. For instance, if the `desiredSizeOfTotalSpace` is NULL (negation of cause 5) and the SRM system supports a default for `desiredSizeOfTotalSpace` (cause 6), then the return value `sizeOfTotalSpace` must be checked to be non-NULL and greater than 0. Causes 11, 12, and 13 are constrained by the E and R constraints: constraint E (*Exclusive*) between nodes 11 and 12 expresses the fact that causes 11 and 12 cannot be simultaneously true; constraint R (*Requires*) between nodes 12 and 13 represents the fact that if 12 is true then 13 must be true.

The study above has allowed us to find inconsistencies in the specification. For instance, if the server did not support the `retentionPolicyInfo` value, both values for the `returnStatus` SRM_NOT_SUPPORTED and SRM_INVALID_REQUEST were considered correct

1 retentionPolicyInfo is not NULL
2 retentionPolicyInfo is supported by server
3 invalid input parameters
4 desiredSizeOfGuaranteedSpace is not NULL
5 desiredSizeOfTotalSpace is not NULL
6 default for desiredSizeOfTotalSpace is supported
7 desiredLifetimeOfReservedSpace is not NULL
8 default for desiredLifetimeOfReservedSpace is supported
9 transferParameters is not NULL
10 transferParameters is compatible with retentionPolicyInfo
11 requestToken is returned
12 spaceToken is returned
13 sizeOfGuaranteedReservedSpace and lifetimeOfReservedSpace are returned

90 returnStatus is SRM_NOT_SUPPORTED
91 returnStatus is SRM_INVALID_REQUEST
92 returnStatus is SRM_REQUEST_QUEUED | SRM_REQUEST_INPROGRESS
93 returnStatus is SRM_SUCCESS | SRM_LOWER_SPACE_GRANTED

94 sizeOfGuaranteedReservedSpace = default
95 lifetimeOfReservedSpace = default
96 transferParameters is ignored

Fig. 2. List of causes (1–13) and effects (90–93) for the `srmReserveSpace` request. Effects 94–96 are intermediate effects that need to be checked in test cases.

by some implementations, due to conflicting wordings in the specifications. After this issue was discovered, it was decided that the correct return code should be SRM_NOT_SUPPORTED. Furthermore, the specification only allowed for SRM_REQUEST_QUEUED to be returned when the method is asynchronous, while also SRM_REQUEST_INPROGRESS is possible for implementations that are fully parallel or threaded. Another example is the default lifetime for the reserved space: The original specification set it to infinite, while for practical implementation issues it was decided to leave the default to be an implementation choice, and the specification was changed accordingly.

C. Error guessing

We conclude our review of the methodologies employed in the SRM service testing with the approach we call *Error guessing*. This is simply the selection of test cases based on the testers' intuition and experience, especially the experience gained in the actual operation and maintenance of the system under test.

Some test cases were devised to clarify issues arisen during the development of a semi-formal model of the SRM [5], [6]. Some of the issues concerned the behavior of the systems when concurrent calls interfered with each other. For instance, in the case of an asynchronous `srmReserveSpace` we studied the effects of an `srmAbortRequest` issued before `srmReserveSpace` is completed. We also checked the information returned by `srmGetSpaceMetadata` (a request for information on space properties) and the effect of an `srmUpdateSpace` (a request to change space properties) once the `srmReserveSpace` is completed. We have used the cause-effect graphing method also in this case, expressing in a formal way the SRM protocol in case of interacting functions.

Furthermore, the error guessing methodology was used in order to find recurrent errors caused by diverging interpretations of the SRM interface. For instance, one implementation for SRM v2.2 used to return generic error codes even when more specific codes were clearly defined by the specification for erroneous situations. This was due to a broad interpretation of the SRM specification. For instance, in the case of SRM transfer methods, the developers interpreted the file level return status as a file characteristic that changed with the file state. In the specification, the file level return status is always connected to the request issued and can be different from request to request, even if the file state does not change. Special test cases were designed to discover situations of this kind. In the case of another implementation a synchronization problem between the SRM server and the backend storage system was found. If an `srmPrepareToPut` (a request that must be issued before a file transfer) was performed on a SURL after an `srmRm` (request to remove a file) on the same SURL, an `srmStatusOfPutRequest` returned first a failure and then a success on the same request. This was due to the fact that the remove operation was executed by the backend asynchronously while for the SRM this operation is synchronous. The SRM server was implemented not to consider the fact that `srmRm` is executed asynchronously, keeping into account the history of operations performed by the client on a specific SURL, before proceeding with satisfying further request on the same SURL.

IV. A TEST EXECUTION SCRIPTING LANGUAGE

Testing a service running on a production Grid composed of a very large number of nodes distributed across the planet requires appropriate tools. The S2 scripting language [10] has been specifically devised to test the SRM service.

An S2 script consists of *actions*, called *branches*. Actions have an *execution value* and an *outcome*. The execution value is 0 for successful execution, or a positive integer related to the severity of failure. The outcome is a logical value depending on the execution value: normally, it is TRUE for success and FALSE for failure, but it is possible to specify a threshold for the execution value under which execution has a TRUE outcome; in this way, testers may filter out less severe failures, or force the execution of subsequent actions that would be otherwise disabled by the failure of a previous action.

The fundamental kind of action is the execution of an SRM command. The S2 interpreter recognizes a set of expressions with a command-line syntax that exercise all the implemented SRM calls. The outputs of the SRM commands are character strings containing several fields whose contents can be extracted by means of Perl-style regular expressions and stored in variables. Other kinds of actions include tests involving the outputs of the SRM commands, and the execution of any operating system call.

Actions can be composed by means of iterative structures (similar to *for* and *while*), parallel execution, AND-sequential execution, and OR-sequential execution. In the two latter forms of composition, execution of each branch depends on the outcome of the previously executed one: AND-sequential execution terminates as soon as a branch has a FALSE outcome, OR-sequential execution terminates as soon as a branch has a TRUE outcome.

Fig. 4 shows a simple test example. In the example, the first branch, made of four lines, is the execution of the SRM request `srmLS`. The first line contains the name of the request followed by environment variables (set by the tester before executing the script) that specify the request arguments. The next three lines contain patterns against which the return values of the request are matched. Pattern matching has the side effect of assigning the matched string to variables. For example, consider the first pattern, "`requestToken=(?P<requestToken>.*)`", that matches a string of characters following the prefix "`requestToken=`" and stores them into the `requestToken` variable. Similarly, other fields of the output go into the variables `pathDetails` and `returnStatus`, with the two fields `explanation` and `statusCode`.

The second branch, executed if the previous one is successful, makes some tests on the values of `returnStatus` and `requestToken`, then it prints the results to a log file with the `echo` system command. Finally, the last branch, executed if any of the previous ones fails, prints an error message to the log file and exits.

S2 has allowed us to build a testing framework that supports the parallel execution of tests where the interactions among concurrent method invocations can be easily tested. The S2 test suite has allowed the early discovery of memory corruption problems by the CGSI security plug-ins in one implementation. Authentication errors were reported randomly when multiple concurrent requests were made to the server. The coding of such a test case required very little time (few minutes) with no errors made in the test program that exercised this test case.

V. Results of SRM service testing

The SRM has currently been implemented for five different Mass Storage Systems, namely CASTOR [11], dCache [7], DPM [12], DRM/BeStMan [13], and StoRM [14].

All these systems are being tested for compliance with the SRM Interface Specification. The analysis described in the previous sections has been performed on the 39 SRM requests defined in the specification, to design a set of test suites that validate the correctness of the implementations. These suites are called the *Availability* tests, that check the availability in time of the SRM service end-points, the *Basic* tests, that verify the basic functionality of the implemented SRM API, the *Use Case* tests that check boundary conditions, request interactions, and exceptions, the *Exhaustion* tests, that exercise all possible syntactic variations for the input arguments, including long strings, blank-padded strings, strings with non-printable characters, or malformed URL's, the *Stress* tests to identify race conditions and observe the behavior of the system when critical concurrent operations are performed, and finally the *Interoperability* tests, that perform remote operations (servers acting as clients) and cross-copy operations among several implementations.

The tests are run automatically six times a day. The data of the last run are stored together with the history of the results and their details. Plots are produced every month on the entire period of run to track the improvements and detect possible problems.

Figure 5 shows the results of the basic tests for all implementations. In particular, the number of failures over the number of total tests executed is reported over time. It is possible to observe that after a month of instability, the implementations converged over a certain set of correctly implemented requests (around the 11th of December 2006). In the week of the 15th of December 2006, a new WSDL description of the SRM Web service interface was introduced to comply with the decisions made about the issues that the new SRM model. At the same time the developers introduced new features and bug fixes that produced oscillations in the plot. After the Christmas break and the upgrade of the testing suite to the new WSDL description, the implementations started to converge again toward a correct behavior with respect to the specifications. We can notice that toward the end of the testing period the number of failures is almost zero for all implementations. That is when testing for a given implementation and a given release of the SRM interface could stop. However, both the interface development and the deployment of implementations on new sites continue, and so does the testing activity.

VI. Conclusions

Very large scale computational Grids, such as the WLCG, by their very size and complexity pose new challenges to the testing process. In the case of the SRM service, some characteristics of this service further stimulate the testers'

```
srmLs $ENV{ENDPOINT} SURL[$ENV{SRM_ENDPOINT}] numOfLevels=0
requestToken=(?P<requestToken>.*) pathDetails=(?P<pathDetails>.*)
returnStatus.explanation=(?P<returnExplanation>.*)
returnStatus.statusCode=(?P<returnStatus>.*)

&& TEST
   $MATCH{(SRM_SUCCESS | SRM_REQUEST_QUEUED
               | SRM_PARTIAL_SUCCESS | SRM_INPROGRESS)
   ${returnStatus}}
      SYSTEM echo "srmLs: OK: ${returnStatus}" >> $ENV{S2_LOG}
      && TEST !$DEFINED{requestToken}
         SYSTEM echo "srmLs: Ls is synchronous" >> $ENV{S2_LOG}
      || SYSTEM echo "srmLs: Ls is asynchronous" >> $ENV{S2_LOG}

|| SYSTEM echo "srmLs: KO: ${-returnStatus}${-returnStatus_explanation}"
>> $ENV{S2_LOG} && exit ${!}
```

Fig. 4. An S2 test script.

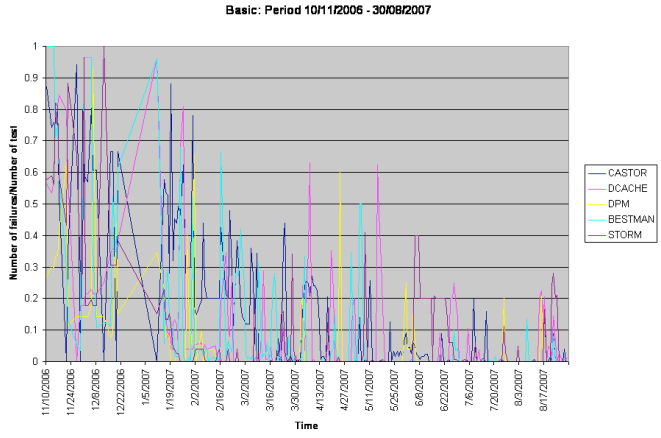

Fig. 5. Results of basic tests.

creativity. The service specification must accommodate for the functionalities of existing storage resources, for the needs of the organizations operating the resources, and for the requirements of high-end user applications. As a result, the specification is rather complex, and it must allow for semantic variations or different conformance levels by the implementations, thus making the choice of test cases more difficult.

From the experience gained in testing the SRM service for the WLCG, it may be learned that the well-established principles and methodologies of functional testing are fundamental, but they must be integrated with a judicious analysis of the environmental and operating conditions of the system. A thorough knowledge of the actual usage patterns enables testers both to reduce the number of test cases that could be obtained by a straightforward reading of the service interface, and to find more test cases whose utility might not be guessed from the same reading.

The SRM protocol is still evolving with new enhancements, and the same testing framework will be used again as part of the normal development and certification cycles.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the collective work of the SRM collaboration and the WLCG project, and in particular the contributions of Arie Shoshani, Alex Sim and Junmin Gu (LBNL), Jean-Philippe Baud, Paolo Badino, Maarten Litmaath (CERN), Timur Perelmutov (FNAL), Patrick Fuhrmann (DESY), Shaun De Witt (RAL), Ezio Corso (ICTP), Luca Magnoni and Riccardo Zappi (CNAF/INFN),

The authors have been supported by CERN and INFN, respectively.

REFERENCES

[1] A. Sim, A. Shoshani *et al.*, "The Storage Resource Manager Interface Specification, Version 2.2," http://www.ogf.org/documents/GFD.129.pdf, Open Grid Forum GSM-WG, Proposed Recommendation GFD-R-P.129, May 2008.

[2] W. Allcock *et al.*, "GridFTP protocol specification," GGF GridFTP Working Group, Document, September 2002.

[3] P. Badino *et al.*, "Storage Element Model for SRM 2.2 and GLUE schema description, v3.5," WLCG, Tech. Rep., Oct. 27, 2006.

[4] "Addendum to the SRM v2.2 WLCG Usage Agreement," http://indico.cern.ch/conferenceDisplay.py?confId=34806, May 2006.

[5] A. Domenici and F. Donno, "A model for the Storage Resource Manager," in *Grid Computing: International Symposium on Grid Computing, (ISGC 2007), Taipei, Taiwan*, ser. Computer Science, S. C. Lin and E. Yen, Eds. Springer US, 2007, pp. 99–105.

[6] ——, "Static and dynamic data models for the Storage Resource Manager v2.2," *Journal of Grid Computing*, vol. 7, no. 1, pp. 115–133, March 2009.

[7] M. Ernst *et al.*, "Managed data storage and data access services for data grids," in *Proceedings of the Computing in High Energy Physics (CHEP) conference*, Interlaken, Switzerland, September 27 – October 1, 2004.

[8] "SRM v2.2 WLCG Usage Agreement," http://cd-docdb.fnal.gov/0015/001583/001/SRMLCG-MoU-day2[1].pdf, May 2006, grid Storage Interfaces Workshop, Fermilab.

[9] A. Shoshani *et al.*, "Storage Resource Management: Concepts, Functionality, and Interface Specification," in *Future of Grid Data Environments: A Global Grid Forum (GGF) Data Area Workshop*, Berlin, Germany, March 9–13, 2004.

[10] F. Donno and J. Menćak, "The S2 testing suite," http://s-2.sourceforge.net, September 2006.

[11] O. Barring *et al.*, "Storage Resource Sharing with CASTOR," in *12th NASA Goddard/21st IEEE Conference on Mass Storage Systems and Technologies (MSST2004)*, U. of Maryland, Adelphi, MD, Apr. 13–16, 2004.

[12] J.-Ph. Baud and J. Casey, "Evolution of LCG-2 Data Management," in *Proceedings of the Computing in High Energy Physics (CHEP'04) conference*, Interlaken, Switzerland, September 27 – October 1, 2004.

[13] A. Sim *et al.*, "Berkeley Storage Manager (BeStMan) Administrative Guide," Lawrence Berkeley National Laboratory, Tech. Rep., February 2009, http://datagrid.lbl.gov/bestman/docs/bestman-guide.pdf.

[14] E. Corso *et al.*, "StoRM, an SRM Implementation for LHC Analysis Farms," in *Proceedings of the Computing in High Energy Physics (CHEP'06) conference*, Mumbai, India, Feb. 2006.