Proposal for a Data Management API within the GridRPC

Y. Caniou, E. Caron, F. Desprez, G. Le Mahec, H. Nakada and Y. Tanimura

February 18, 2009

Status of This Memo

This document (version 0.9) provides a recommendation to the Grid community on a proposed model and API for data management to GridRPC. Distribution is unlimited.

Copyright Notice

Copyright © Open Grid Forum (2007). All Rights Reserved.

Abstract

This document follows the document produced by the GridRPC-WG on GridRPC Model and API for End-User applications [1]. This new document aims at completing the GridRPC API with Data Management mechanisms and API. This document is not intended to provide features and capabilities for building data management middleware. The goal of this document is to complete the GridRPC set of functions and definitions to allow users to manipulate their data. The motivation for this document is to provide explicit functions to manipulate the exchange of data between a GridRPC platform and a client since (1) the size of the data used in Grid applications may be large and useless data transfers must be avoided; (2) data are not always stored on the client side but may be made available either on a storage resource or within the GridRPC platform.
1 Introduction

The goal of this document is to define a data management extension to the GridRPC API for End-User applications. As for the GridRPC API document [1], it is out of the scope of this document to discuss the implementation of data management mechanisms inside a GridRPC platform or on a data storage server.

The motivation of the data management extension is to provide explicit functions to handle data exchanges between a data storage service, a GridRPC platform, and the client. The GridRPC API defines a RPC mechanism to access Network Enabled Servers. However, an application needs data to run and generates some output data, which have to be transferred. As the size of the data may be large in grid environments, it is mandatory to optimize the transfers of large data and avoid useless exchanges. Several cases may be considered depending on where data are stored: on an external data storage, inside the GridRPC platform or on the client side. In all these cases, the knowledge of “what to do with these data?” is owned by the client. Then, the GridRPC API must be extended to provide functions for explicit and simple data management.

We firstly present a motivation for the data management and in Section 3, the proposed data management model is introduced. The main contribution of this document is given in Section 4 where we describe our proposal for a data management API.

2 Data Management motivation

The main motivation of the data management extension is to provide a way to explicitly manage the data and their placement in the GridRPC model. With the help of this explicit management, the client will avoid useless transfers of large data. However, the client may not want to, or may not know how to manage data. Then, the default behavior of the GridRPC Data Management extension must be in accordance with the GridRPC API document. To illustrate the motivation of data management, we give now some examples describing when it can be used.

![Diagram of data locations in the GridRPC model](image)

In a GridRPC environment, data can be stored either on a client host, on a data storage server, on a computational server or inside the GridRPC platform, as shown in Figure 1. When clients do not need to manage their data, then the basic GridRPC API is sufficient. On each `grpc_call()`, data is transferred between a client and the computational server used. Once the computation performed, results are sent back to the client. However, to minimize data transfers, clients need data management functions.

Next, we explain two different cases concerning external data and internal data:

- External data are placed on servers, like data repositories. These servers are not registered inside the platform but can be directly accessed to read/write data. The use of such data implies several data transfers if the client uses the basic GridRPC API: the client must download the data and then send...
it to the GridRPC platform when issuing the call to grpc_call(). One of these transfers should be avoided: the client may just give a data reference (or handle) to the platform/server and the transfer is completed by the platform/server. Consider a client, with small data storage capacities, that needs to issue a request on large data stored in a data storage server. It is costly, and it may be not possible to send the data first to the client before issuing the call. The client could also need to directly store its results from the computational server to the data storage, or share an input or output data with other users of the GridRPC platform. Examples of such Data Storage servers are IBP [2] or SRB [3]. Among the different available examples of this approach, we can cite: (1) the Distributed Storage Infrastructure of NetSolve [6]; (2) the utilization of JuxMem in DIET [8]. This approach is well suited for, but not limited to, long life persistent data.

- Internal data are managed inside the GridRPC platform. Their placement depends on computations and it may be transparent to clients: in this case, the GridRPC middleware can manage data. Temporary data, generated by request sequencing [4], are examples of internal data. For instance, a client issues two calls to solve the same problem and the second call use input or output data from the first call. This is the case if we solve $C =^{t} (A \times B)$, where A and B are matrices. If the client do not find a solver which computes these two operations in one step, then he must issue two calls: $W = A \times B$ and $C =^{t} W$. But the value of $W$ is of no interest for him. Then, this matrix should not be sent back to the client host. Temporary data should be leaved inside the platform, close to or on the computational server, when clients do not need it. Other cases of useless temporary data occur when the results of a simulation are sent to a graphical viewer as done in most Problem Solving Environments. Among the examples of internal data management, we can cite the Data Tree Management infrastructure used in DIET [5], and the data layer omniStorage in omniRPC [7]. This approach is suitable for, but not limited to, intermediate results to be reused in case of request sequencing.

### 3 GridRPC Data Management model

As exposed in the previous section, we consider two different types of data: external and internal data.

In the external data case, data are explicitly stored on a storage depot. Clients manage explicitly their data. When clients invoke a server, they give a data reference to identify the data used for the computation. A client can use already existing data by just providing the data identification given by the storage service.

In the internal data case, the data management service tries to read/write the data inside the GridRPC platform, from the client side or on a computational server. In this case, either the client knows where the data is stored and can manage the transfer (he can use the same calls than the ones to manage external data stored on a storage server), either the data is transparently managed by the GridRPC middleware, in which case the middleware provides mechanisms for transfers between computational servers.

These two approaches are complementary in the data management model proposed here. The GridRPC platform and the data storage interact to implement data transfers. Note that some additional functionalities which are not addressed in this document, can be designed, such as: reusable data generated by a computation could be stored during a TTL (Time To Leave) on the computational server before being sent to data storage servers; or when the storage capacity of a computational server is overloaded, it may be sent to another data storage server.

In both cases, it is mandatory to identify each data. All data stored either in the platform or on storage servers will be identified by Data Handles and Storage Information. Without lack of generality, we define the GridRPC data type as either the data used for a computational problem, either both a Data Handle and storage information. Indeed, when a computational server receives a GridRPC data which does not contain the computational data, it must know the unique name of the data with the Data Handle, and must know its location to get it and where the client wants to save it after the computation. Thus storage information must record the original location of the data and the destination of the data.
4  Data Management API

In [1], data used as input/output parameters are provided within the <varargs> notation of the grpc_call() and grpc_call_async() functions. Without lack of generality, and in order to propose an API independent of the language of implementation, we refer to grpc_data_t as the type of such variables. Thus, in the following, a grpc_data_t is any kind of data, or contains a reference on the computational data, which we call a Data Handle, as well as some Storage Information.

Next, we firstly define some data types for GridRPC data management. We present afterwards the different functions to managed them, composing the proposed API for GridRPC Data Management.

Note that in the following, we refer to “GridRPC data” to designate the generic data which is used in the GridRPC calls and “data” to designate the content data.

4.1 GridRPC data types

We introduce here the notion of a GridRPC data which at least includes the data or a data handle, and may contains some information about the data itself (e.g., type, size) as well as information on its location and the protocol used to access it (e.g., the URI of a specific server, a link with a Storage Resource Broker, containing the correct protocol to use). A data handle is essentially a unique reference to a data that may reside anywhere. Data and data handles can be created separately. By managing GridRPC data with data handles, clients do not have to know where data are currently stored.

4.1.1 The grpc_data_t type

A data in a GridRPC middleware is defined by the grpc_data_t type. Variables of this type represent information on a specific data which can be local or remote. It is at least composed of:

- Two NULL-terminated lists of URIs, one to access the data and one if the data has to be stored somewhere from this server (for example, an OUT parameter to transfer at the end of a computation).
- Information concerning the mode of management. For example, data management is defaulted to the one of the standard GridRPC paradigm, but it can be noted for example as GRPC_PERSISTENT, which corresponds to a transparent management by the GridRPC middleware.
- Information concerning the type of the data, as well as its size.

<table>
<thead>
<tr>
<th>grpc_data_type_t</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPC_INT</td>
<td>integer</td>
</tr>
<tr>
<td>GRPC_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>GRPC_COMPLEX</td>
<td>complex</td>
</tr>
<tr>
<td>GRPC_STRING</td>
<td>string</td>
</tr>
<tr>
<td>GRPC_FILE</td>
<td>file</td>
</tr>
<tr>
<td>GRPC_CONTAINER_OF_GRPC_DATA</td>
<td>container of grpc_data_t</td>
</tr>
</tbody>
</table>

Table 1: Definition of grpc_data_type_t codes
Details on Storage Information

- URI: it defines the location where a data is stored. It can be built like “protocol://machine_name:port/path_to_data” and thus, contains at least four fields. Some straightforward examples are given in Section 4.2 and several full examples of utilization can be found in Section A.
  - char * protocol: one of the token {“file”, “nfs”, “memory”, “ibp”, “local_fs”, “middleware”, “http”}, and gives some information on how to access the data (the list is not exhaustive).
  - char * hostname: the name of the server on which resides the data.
  - int port: the port to use to access the data.
  - char * path: the full path of the data or an ID.

- The management mode is an enumerated type grpc_data_mode_t defined by the client. It is useful to set the behavior of the data on the platform. It is related to the following policy values. If the middleware does not handle the given behavior, it throws an error.
  - GRPC_VOLATILE: used when the data may not be kept inside the platform after a computation (can be considered as the default usage for GridRPC API). Still, the Data Management middleware can keep the data in the system, migrate and replicate it, for later use. Hence, potential coherency issues may arise.
  - GRPC_UNIQUE_VOLATILE, used when the data must not be kept inside the platform after a computation for security reason for example (can be considered as the default usage for GridRPC API).
  - GRPC_PERSISTENT: used when a data has to be kept on the platform. The data is not sent back to the client. Moreover, the data item can migrate or be replicated between servers depending on scheduling decisions, and potential coherency issues may arise.
  - GRPC_STICKY: used when a data is kept inside the platform but cannot be moved between the servers. In that case the data is not given back to the client after computation. This is used if the client needs that data in the platform for a second computation on the same server for example. Note that in this case, the data can be replicated and that potential coherency issues may arise.
  - GRPC_UNIQUE_STICKY: used when a data is kept inside the platform but cannot be moved between the servers. In that case the data is not given back to the client after computation. This is used if the client needs that data in the platform for a second computation on the same server for example. Note that in this case, the data cannot be replicated for security reason for example.
  - GRPC_END_LIST: this is not a type, but just marker, which is used to terminate grpc_data_mode_t lists.

- The type of the data is an enumerated type grpc_data_type_t defined by the client: it describes the type of the data, for example GRPC_DOUBLE, GRPC_INT, etc., as exposed in Table 1. We can note that a special grpc_data_t can contain other grpc_data_t. That way, the user relies on the GridRPC Data Middleware to transfer a set of grpc_data_t. The matter of how to implement it by an array, a list or anything else is GridRPC Data Middleware dependant, then not in the purpose of this document.

4.1.2 Function specific types

In this section, we describe some types that are only used in a given function. They are enumerated types, and are given here for commodity reasons.
The `grpc_completion_mode_t` type. This type is used in `grpc_data_wait()` and is defined by the enumerated type `{ GRPC_WAIT_ALL, GRPC_WAIT_ANY }` which can be extended. It is used to detail the behavior of the waiting process: the function can wait for one or all transfers concerning data involved during the call to `grpc_data_wait()`.

The `grpc_data_info_type_t` type. This type is only used with the `grpc_data_getinfo()` function to define the wanted information. It is an enumerated type defined with the following values (which can be extended):

- `GRPC_HANDLE`
- `GRPC_INPUT_URI`
- `GRPC_OUTPUT_URI`
- `GRPC_MANAGEMENT_MODE` (used to know if a data is for example `GRPC_VOLATILE`, etc.)
- `GRPC_SIZE`
- `GRPC_TYPE` (used to know to which type of the language of implementation the data corresponds)
- `GRPC_LOCATIONS_LIST`
- `GRPC_STATUS` (used to know if a `grpc` data is “GRPC_IN_PLACE” or “GRPC_TRANSFERRING”)
- `GRPC_COHERENT` (used to know if the considered data is managed by a Data Management middleware that ensures coherency in all replicas (and then, if this replica may or is up-to-date) – should be dependent on the locations)

Note: Information is managed by the GridRPC Data Management API, which relies on at least one Data Management middleware. Then, information concerning a data can be stored within the GridRPC Data Management middleware, and/or within the `grpc_data_t` type. Nonetheless, this document does not focus on implementation.

4.2 Examples of use

- A GridRPC data corresponding to an input matrix stored in memory can partly be constructed with the information of `protocol=memory`, `port` is a null string, the machine name is the one of the localhost and `path_to_data` is the path used to access the data in memory (such as a pointer in C language, or a key that lets the GridRPC API either make the correspondance with the correct input to give to the data middleware, either the key used by the data middleware).

- The URI “http://myName:/myhome/data/matrix1” corresponds to the location of a file named matrix1, which we can access on the machine named myName, with the `http` protocol. Typically, the data, stored as a file, can be downloaded with a command like “`wget http://myName/myhome/data/matrix1`”

4.3 Data Management functions

Data handles are provided by the GridRPC Data Management middleware. They must be unique, and the middleware must record some information about the data, such as the location, the size, etc. The `init` function sets the data handle to the data it identifies, while the user provides needed information concerning the location on where the data is stored and where it has to be stored after the computation. Using this function, all the semantic needed to provide data management and data persistence can be covered.
Data exchanges between client and explicit locations (computational servers or storage servers) are done using the asynchronous read and asynchronous write functions. Consequently, the GridRPC data can also be inspected, or probed, to get more information about the status of the data or its location. Functions are also given to wait after the completion of some transfers. Finally, one can unbind the handle and the data, and free the GridRPC data.

To provide identification of long lived data, data handles should be saved and restored, for instance in a file. This will allow two different users to share the same data. Security and data life cycle management issues are not of the API concerns.

Examples of the use of this API are given in Appendix A.

4.3.1 The init function

The init function initializes the GridRPC data with a specific data. This data may be available locally or on a remote storage server. Both identifications can be used. GridRPC data referencing input parameters must be initialized with identified data before being used in a grpc_call(). GridRPC data referencing output parameters do not have to be initialized.

Function prototype:

```c
grpc_error_t grpc_data_init(grpc_data_t * data,
    const char ** list_of_URI_input,
    const char ** list_of_URI_output,
    const grpc_data_type_t variable_type,
    const size_t * dimensions_sizes,
    const grpc_data_mode_t * list_of_data_mode);
```

list_of_URI_input and list_of_URI_output parameters are NULL-terminated lists of strings, which give the different locations on where to transfer the data from, and the locations on where to possibly transfer the data to, as explained previously. Hence, a list describes all the available locations known by the client, in order for the GridRPC Data Management middleware to possibly implement some efficient and fault-tolerant mechanisms to perform a choice among all the proposed selections (and the ones eventually known by the Data Management middleware). In sake of simplicity, one can imagine that the default behavior would be a sequential try until the transfer to/from one of them can be achieved.

Remarks:

- If GRPC_UNIQUE_VOLATILE is used at the same time than a resolution_mode equal to GRPC_NO_RESULT, then if List_of_URI_output is NULL, the result is lost.
- storage_mode is by default set to GRPC_VOLATILE. If another value is given, for example GRPC_STICKY, then the storage management is applied to all locations if needed. If a user wants to use the same handle in order to use the same data possibly being managed differently on numerous locations (for example GRPC_STICKY on given resources and GRPC_VOLATILE on others), then he has to do it when calling the grpc_data_write() function.
- Some of the parameters, such as the output parameter, can be unset.
- If the function is called with a grpc_data_t which has been used in a previous call, fields corresponding to information already given are overwritten.

<table>
<thead>
<tr>
<th>Error code identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPC_NO_ERROR</td>
<td>Success</td>
</tr>
<tr>
<td>GRPC_NOT_INITIALIZED</td>
<td>grpc_initialize was not called in advance</td>
</tr>
<tr>
<td>GRPC_NOT_SUPPORTED</td>
<td>the capability is not supported</td>
</tr>
<tr>
<td>GRPC_INVALID_TYPE</td>
<td>Specified type is not valid</td>
</tr>
<tr>
<td>GRPC_INVALID_MODE</td>
<td>Specified location is not valid</td>
</tr>
<tr>
<td>GRPC_MALFORMED_URI</td>
<td>One of the URI is not valid</td>
</tr>
<tr>
<td>GRPC_OTHER_ERROR_CODE</td>
<td>Internal error detected</td>
</tr>
</tbody>
</table>

8
4.3.2 Functions specific to a special mode or data

Mappings of memory location to given names.

If he wants to use a data which is in memory, the user must provide some name in the URIs in the input or output fields which has to be understood by the GridRPC Data Management layer in the GridRPC system, in addition of the use of the memory protocol. For this reason, we provide here two functions:

Function prototype:

```c
grpc_error_t grpc_data_memory_mapping_set(const char * key, void * data);
grpc_error_t grpc_data_memory_mapping_get(const char * key, void ** data);
```

The function `grpc_data_memory_mapping_set()` is used to make the relation between a data stored in memory and a `grpc_data_t` data when the memory protocol is used: the aim is to set a keyword that will be used in the URI used for example during the initialization of the data.

Containers of `grpc_data_t` management functions.

In order to facilitate the use of some special structures like lists or arrays of `grpc_data_t` variables, the two following functions let the user manipulate them at a higher level and without knowing the contents of the structures.

Function prototype:

```c
grpc_error_t grpc_data_container_set(grpc_data_t * container, int rank,
                                     const grpc_data_t * data);
grpc_error_t grpc_data_container_get(const grpc_data_t * container, int rank,
                                     grpc_data_t ** data);
```

The variable `container` is necessarily a `grpc_data_t` of type `GRPC_CONTAINER_OF_GRPC_DATA`. `rank` is a given integer which acts as a key index, and `data` is the data that the user wants to add in or get from the container. Note that getting the data does not remove the data from the container. Furthermore, the container management is free of implementation.

<table>
<thead>
<tr>
<th>Error code identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPC_NO_ERROR</td>
<td>Success</td>
</tr>
<tr>
<td>GRPC_NOT_INITIALIZED</td>
<td>grpc_initialize was not called in advance</td>
</tr>
<tr>
<td>GRPC_NOT_SUPPORTED</td>
<td>the capability is not supported</td>
</tr>
<tr>
<td>GRPC_INVALID_TYPE</td>
<td>Specified type is not valid</td>
</tr>
<tr>
<td>GRPC_INVALID_RANK</td>
<td>Specified rank is not valid</td>
</tr>
<tr>
<td>GRPC_OTHER_ERROR_CODE</td>
<td>Internal error detected</td>
</tr>
</tbody>
</table>

4.3.3 The transfer function

This function writes a GridRPC data identified by `data` to the output locations set during the init call in the output parameters fields. For commodity reasons, a list of additional servers on which the data has to be uploaded can be provided.

A user may want to be able to transfer data while computations are done. For example, if a computation can begin as soon as some data are downloaded but needs all of them to finish, the management of data must use asynchronous mechanisms as default behavior. Then, this function initiates the call for the transfers and returns immediately after.

Function prototype:
grpc_error_t grpc_data_transfer(grpc_data_t * data,
    const char ** list_of_input_URI,
    const char ** list_of_output_URI,
    const grpc_data_mode_t * list_of_input_modes,
    const grpc_data_mode_t * list_of_output_modes);

list_of_[input/output]_modes are NULL-terminated lists with the same number of items than list_of_[input/output]_URI. For each URI describing the hostname, the protocol used to access the data, etc., a management mode can be specified: a data can be flagged GRPC_STICKY on given resources. Hence, list_of_[input/output]_modes can be used to set different management policies on some resources (for example, set the data as GRPC_STICKY to a set of a resources and GRPC_PERSISTENT to the others) while possibly benefiting of an “aggressive” write as the data is the same everywhere.

Remarks:

- If one of the list_of_[input/output]_modes is set to NULL, the management mode of the data is the one specified during the initialization of the data, else the management mode is overridden.
- No information is given as when the transfer will indeed begin.
- If a user needs to know if the transfer is completed on one or another server (or all), he can use the grpc_data_getinfo() function.
- If a user wants to wait of the completion of one or more transfers, he can use the grpc_data_wait() function.
- If the data middleware (e.g., the GridRPC middleware or the data middleware on which it relies) does not manage coherency between the duplicates on the platform, a correct call to this function can be useful to ensure that all copies are up to date.

<table>
<thead>
<tr>
<th>Error code identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPC_NO_ERROR</td>
<td>Success</td>
</tr>
<tr>
<td>GRPC_NOT_INITIALIZED</td>
<td>grpc_initialize was not called in advance</td>
</tr>
<tr>
<td>GRPC_NOT_SUPPORTED</td>
<td>the capability is not supported</td>
</tr>
<tr>
<td>GRPC_INVALID_DATA</td>
<td>Specified data is not valid</td>
</tr>
<tr>
<td>GRPC_MALFORMED_URI</td>
<td>One of the URI is not valid</td>
</tr>
<tr>
<td>GRPC_OTHER_ERROR_CODE</td>
<td>Internal error detected</td>
</tr>
</tbody>
</table>

To discuss:

- A special ERROR if protocol not set or not correct should be given?
- A special ERROR if at least on server does not respond?
- Functions read/write to perform the IN/OUT transfers?
- A “diffusion mode” definition? Some broadcast/multicast mechanisms can then be implemented in the GridRPC data middleware in order to improve performance. The diffusion mode can be used by more intelligent data middleware to diffuse a data in a broadcast manner for example.

4.3.4 The grpc_data_wait() function

For convenient reasons, the function grpc_data_transfer() is asynchronous. Hence, a user have the possibility to perform overlap transfers with computation and try to realize transfers in parallel. This function can then be used by the user to wait for the completion of one or several transfers.

Function prototype:
grpc_error_t grpc_data_wait(const grpc_data_t ** list_of_data,
                        grpc_completion_mode_t  mode);

Depending on the value of mode (GRPC_WAIT_ALL or GRPC_WAIT_ANY), the call returns when all or
one of the data listed in list_of_data is transferred, which means that for a given data, all transfers involved
for the input or output part are finished.
Remarks:

- If list_of_data is NULL, then either one or all data (depending on the value of mode) are transferred since the call to grpc_initialize().

- The use of this function can be done in such a way that the server can test if data are in place (i.e., that transfers involved in the grpc_data_transfer() on the client part have been completed) before doing anything. If the user performs a grpc_data_transfer() of a grpc_data_t whose transfer has not been completed, the behavior is depending on the data middleware which manage the data: if the middleware implements some stamps mechanisms, then no problem will occur.

- This function considers only the information that the user is aware of: if the data is shared between different users, then a call to grpc_data_wait() returns depending on the input of the user that has performed the call. Hence, the call will not depend on an other user action.

<table>
<thead>
<tr>
<th>Error code identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPC_NO_ERROR</td>
<td>Success</td>
</tr>
<tr>
<td>GRPC_NOT_INITIALIZED</td>
<td>grpc_initialize was not called in advance</td>
</tr>
<tr>
<td>GRPC_NOT_SUPPORTED</td>
<td>the capability is not supported</td>
</tr>
<tr>
<td>GRPC_INVALID_HANDLE</td>
<td>Specified handle is invalid</td>
</tr>
<tr>
<td>GRPC_INVALID_DATA</td>
<td>Specified data is not valid</td>
</tr>
<tr>
<td>GRPC_OTHER_ERROR_CODE</td>
<td>Internal error detected</td>
</tr>
</tbody>
</table>

4.3.5 The unbind function

When the user does not need a handle anymore, but knows that the data may be used by another user for example, he can unbind the handle and the GridRPC data by calling this function without actually freeing the GridRPC data on the remote servers.

Function prototype:

```c
grpc_error_t grpc_data_unbind(grpc_data_t * data);
```

After calling this function, data does not reference the data anymore.

<table>
<thead>
<tr>
<th>Error code identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPC_NO_ERROR</td>
<td>Success</td>
</tr>
<tr>
<td>GRPC_NOT_INITIALIZED</td>
<td>grpc_initialize was not called in advance</td>
</tr>
<tr>
<td>GRPC_NOT_SUPPORTED</td>
<td>the capability is not supported</td>
</tr>
<tr>
<td>GRPC_OTHER_ERROR_CODE</td>
<td>Internal error detected</td>
</tr>
</tbody>
</table>

4.3.6 The free function

This function frees the GridRPC data identified by data on a subset or on all the different locations where the data is stored, and unbind the handle and the data. This function may be used to explicitly erase the data on a storage resource.

Function prototype:

```c
grpc_error_t grpc_data_free(grpc_data_t * data, const char ** URI_locations);
```

If URI_locations is NULL, then the data is erased on all the locations where it is stored, else it is freed on all the location contained in the list of URI.

After calling this function, data does not reference the data anymore.

<table>
<thead>
<tr>
<th>Error code identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPC_NO_ERROR</td>
<td>Success</td>
</tr>
<tr>
<td>GRPC_NOT_INITIALIZED</td>
<td>grpc_initialize was not called in advance</td>
</tr>
<tr>
<td>GRPC_NOT_SUPPORTED</td>
<td>the capability is not supported</td>
</tr>
<tr>
<td>GRPC_INVALID_DATA</td>
<td>Specified data is invalid</td>
</tr>
<tr>
<td>GRPC_OTHER_ERROR_CODE</td>
<td>Internal error detected</td>
</tr>
</tbody>
</table>
4.3.7 A function to get information on a grpc_data_t variable

This function let the user access information about an instantiation of a grpc_data_t. It returns information on data characteristics, status, locations, etc.

Function prototype:

```c
grpc_error_t grpc_data_getinfo(const grpc_data_t * data, 
                               grpc_data_info_type_t info_tag, 
                               const char * URI, 
                               char ** info);
```

The kind of information that the function gets is defined by the info_tag parameter. A server name can be given to get some data information dependent on the location of where is the data (like GRPC_STICKY).

Remarks:

- For values equal to GRPC_INPUT_URI and GRPC_OUTPUT_URI, the returned list is considered to be information on the grpc_data in the system, not only the information got locally for the handle (or stored in the grpc_data_t).
- server_name can be set to NULL (default behavior). In that case, if the user tries to access the information of the mode (GRPC_STICKY for example) and the data has different management mode on the platform, then the value GRPC_UNDEFINED may be returned.
- If info_tag equals to GRPC_STATUS, then info can be one of the “GRPC_IN_PLACE” and “GRPC_TRANSFERING” value.

Note that in case of info_tag is set to GRPC_HANDLE, information is of no use to manage data with the given API: handles are initialized in the init call function, stored in the grpc_data_t.

<table>
<thead>
<tr>
<th>Error code identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPC_NO_ERROR</td>
<td>Success</td>
</tr>
<tr>
<td>GRPC_NOT_INITIALIZED</td>
<td>grpc_initialize was not called in advance</td>
</tr>
<tr>
<td>GRPC_NOT_SUPPORTED</td>
<td>the capability is not supported</td>
</tr>
<tr>
<td>GRPC_INVALID_DATA</td>
<td>Specified data is invalid</td>
</tr>
<tr>
<td>GRPC_OTHER_ERROR_CODE</td>
<td>Internal error detected</td>
</tr>
</tbody>
</table>

4.3.8 The load_data and save_data functions

In order to communicate a reference between Grid users, for example in case of large size data, one should be able to store a GridRPC data. The location can then be shared, for example by mail, and one can be able to load the corresponding information.

Function prototype:

```c
grpc_error_t grpc_data_load(const grpc_data_t * data, const char * URI_input); 
grpc_error_t grpc_data_save(const grpc_data_t * data, const char * URI_output);
```

These functions are used to load/save the data descriptions. Even if the GridRPC data contains the data in addition to metadata management informations (data handle, size, type, etc.), only data informations have to be saved in the location. The format used by these functions is left to the developer’s choice. The way the informations are shared by different middleware is out of scope of this document and should be discussed in an interoperability recommendation document.
<table>
<thead>
<tr>
<th>Error code identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRPC_NO_ERROR</td>
<td>Success</td>
</tr>
<tr>
<td>GRPC_NOT_INITIALIZED</td>
<td>grpc_initialize was not called in advance</td>
</tr>
<tr>
<td>GRPC_NOT_SUPPORTED</td>
<td>the capability is not supported</td>
</tr>
<tr>
<td>GRPC_INVALID_DATA</td>
<td>Specified data is invalid</td>
</tr>
<tr>
<td>GRPC_OTHER_ERROR_CODE</td>
<td>Internal error detected</td>
</tr>
</tbody>
</table>

References

A  Examples of use of the API

In this section, we give examples of the data management API usage to illustrate its interest. Depending on the passing mode of the arguments (data), we show how to optimize data placement and avoid useless transfers. We do not consider these examples as an exhaustive list but they can help to understand the way to build a data management API in GridRPC middleware.

A.1  Basic example

```c
grpc_function_handle_init(handle1,"karadoc.aist.go.jp","*”); 
grpc_data_init(&dhA, 
{const char * []}{"memory://britannia.ens-lyon.fr/#A", NULL}, 
NULL, 
GRPC_DOUBLE, (size_t []){3, 3, 0}, 
{grpc_data_mode_t []}{GRPC_VOLATILE, GRPC_END_LIST});

grpc_data_init(&dhB, 
{const char * []}{"nfs://britannia.ens-lyon.fr/home/user/B.dat", NULL}, 
NULL, 
GRPC_DOUBLE, (size_t []){3, 3, 0}, 
{grpc_data_mode_t []}{GRPC_VOLATILE, GRPC_END_LIST});

grpc_data_init(&dhC, 
NULL, 
{const char * []}{"nfs://britannia.ens-lyon.fr/home/user/C.out", NULL}, 
GRPC_DOUBLE, (size_t []){3, 3, 0}, 
{grpc_data_mode_t []}{GRPC_VOLATILE, GRPC_END_LIST});

grpc_call(handle1, &dhA, &dhB, &dhC);
```

![Diagram](Figure 2: Simple RPC call with input and output data.)

In this example (see Figure 2), we show an example on how to use the GridRPC data management functions when the data does not need to be stored inside the platform or on a storage resource. This example corresponds to the default behavior of the data management performed in the GridRPC paradigm, but conducted by the client with data handles.
A.1.1 Input data

Here, we illustrate the way to send a local data (in memory and on disk) to the GridRPC platform. In this example, the client issues a call with two input data $A$ and $B$. $A$ and $B$ are local to the client. Note that we use the &A notation in the URI for commodity reason, but the real memory address should be given here.

A.1.2 Output data

Output data $C$ is sent back to the client because in our case, no data conservation is needed. In this example, the client issues a call with $A$ and $B$ as input data and $C$ as output data. We note that the example uses the GRPC_NO_RESULT resolution mode during the initialization of the grpc_data_t because the client manage the output data: the given URI is used to manage the data. If GRPC_RESULT_RETURN was used, there could be two transfers: one more because of the normal GridRPC behavior of the GridRPC middleware which transfers output data locally on the client (in that case, the client may have to question the GridRPC middleware to get how to access the data).

A.2 Example with external storage resources

In this example we show how to use the GridRPC data management when the data is stored on an external data repository.

Figure 3 shows how to manage external data repository as IBP or SRB.

```c
grpc_function_handle_init(handle12,"karadoc.aist.go.jp","*");
grpc_data_init(&dhA,
(const char * []){"IBP://kaamelott.cs.utk.edu/1212#A.dat/ReadKey/READ", NULL},
(const char * []){"NFS://britannia.ens-lyon.fr/home/user/A.dat", NULL},
GRPC_DOUBLE, (size_t []){3, 3, 0},
(grpc_data_mode_t []){GRPC_VOLATILE, GRPC_END_LIST});
grpc_data_init(&dhB,
(const char * []){"SRB://carmelide.ens-lyon.fr/COlLECTION/Simulations/B.dat", NULL},
(const char * []){"IBP://kaamelott.cs.utk.edu/1213#B.dat/WriteKey/WRITE", NULL},
GRPC_DOUBLE, (size_t []){3, 3, 0},
(grpc_data_mode_t []){GRPC_VOLATILE, GRPC_END_LIST});
grpc_data_init(&dhC, NULL,
(const char * []){"NFS://britannia.ens-lyon.fr/home/user/C.out", NULL},
GRPC_DOUBLE, (size_t []){3, 3, 0},
(grpc_data_mode_t []){GRPC_VOLATILE, GRPC_END_LIST});
grpc_call(handle1,dhA,dhB,&dhC);
```

A.2.1 Input data

Here, we illustrate the way to send a remote data stored on SRB or IBP server to the GridRPC platform. In this example, the client issues a call with two input data $A$ and $B$. $A$ is available on SRB repository and $B$ is available on IBP repository. With the input and output parameters from the grpc_data_init() function we can move the data from a repository to another:

- $A$ is read from SRB server and will be sent to the client.
- $B$ is read from IBP server and will be sent to SRB server.

A.2.2 Output data

Output data $C$ is sent back to the client.
A.3 Example with persistence

In this example we show how to re-use data on a specific server without resending them. Client wants to compute\( C = C \times A^n \) using the service "\(*\)" on server karadoc.

```c
double * A;
grpc_data_memory_mapping_set("A", A); // set mapping for memory scheme

grpc_function_handle_init(handle1,"karadoc.aist.go.jp","*");

grpc_data_init(&dhA,
(const char * []){"memory://britannia.ens-lyon.fr/A", NULL},
(const char * []){"memory://karadoc.aist.go.jp", NULL},
GRPC_DOUBLE, (size_t []){3, 3, 0},
(grpc_data_mode_t []){GRPC_STICKY, GRPC_END_LIST});

grpc_data_init(&dhC,
(const char * []){"NFS://britannia.ens-lyon.fr/home/user/C.in", NULL},
(const char * []){"memory://karadoc.aist.go.jp", NULL},
GRPC_DOUBLE, (size_t []){3, 3, 0},
(grpc_data_mode_t []){GRPC_STICKY, GRPC_END_LIST});

for(i=0;i<n+1;i++)
{
    if( i==1 )
    {
        grpc_data_init(&dhC,
        (const char * []){"memory://karadoc.aist.go.jp", NULL}, NULL,
        GRPC_DOUBLE, (size_t []){3, 3, 0},
        (grpc_data_mode_t []){GRPC_STICKY, GRPC_END_LIST});
    }
    if( i==n )
    {
        grpc_data_init(&dhC,
        (const char * []){"memory://karadoc.aist.go.jp", NULL},
        (const char * []){"NFS://britannia.ens-lyon.fr/home/user/C.out", NULL},
        GRPC_DOUBLE, (size_t []){3, 3, 0},
        (grpc_data_mode_t []){GRPC_STICKY, GRPC_END_LIST});
    }
}```

Figure 3: Simple RPC call with input and output data using external storage resources.
Figure 4: GridRPC call with data management using persistence through the GRPC_STICKY mode.

In this example (see Figure 4), we show how to use the GridRPC data management functions when the data needs to be stored inside the platform. In this example we consider the client needs to keep the data on the same server. The GRPC_STICKY mode provides this behavior.

A.3.1 Input data

Data \( A \) will be used and will remain on server karadoc, we can use the GRPC_STICKY parameter to keep the data on server karadoc. Data \( C \) is an input/output data. The first \texttt{grpc_data_init} for this data requires only an input location and the GRPC_STICKY mode.

A.3.2 Output data

Output data \( C \) is generated on server karadoc but only the last result is useful for the client. Thus, to send the final result to the client we update the output location just before the last \texttt{grpc_call()}.

A.4 Example with prefetching

In this example we show how the user can deal with the GridRPC to manage the data and thus performs the prefetching of data. Client wants to compute \( C = A \times B \) on server karadoc and \( C = A + C \) on server perceval.

\begin{verbatim}
grpc_function_handle_init(handle1,"karadoc.aist.go.jp", "+");
grpc_function_handle_init(handle2,"perceval.rush.aero.org","*");
\end{verbatim}
/* Data initialization */
grpc_data_init(&dhA,
(const char *[]){"IBP://kaamelott.cs.utk.edu/1212#A.dat/ReadKey/READ", NULL},
(const char *[]){"NFS://britannia.ens-lyon.fr/home/user/A.dat", NULL},
GRPC_DOUBLE, (size_t[]){3, 3, 0},
(grpc_data_mode_t[]){GRPC_VOLATILE, GRPC_END_LIST});
grpc_data_init(&dhB,
(const char *[]){"SRB://carmelide.ens-lyon.fr/COLLECTION/Simulations/B.dat", NULL},
(const char *[]){"IBP://kaamelott.cs.utk.edu/1213#B.dat/WriteKey/WRITE", NULL},
GRPC_DOUBLE, (size_t[]){3, 3, 0},
(grpc_data_mode_t[]){GRPC_VOLATILE, GRPC_END_LIST});
grpc_data_init(&dhC, NULL,
(const char *[]){"NFS://perceval.rush.aero.org/home/user/C.out", NULL},
GRPC_DOUBLE, (size_t[]){3, 3, 0},
(grpc_data_mode_t[]){GRPC_PERSISTENT, GRPC_END_LIST});
grpc_data_init(&dhD,
(const char *[]){"NFS://perceval.rush.aero.org/home/user/C.out", NULL},
(const char *[]){"NFS://britannia/home/user/C.out", NULL},
GRPC_DOUBLE, (size_t[]){3, 3, 0},
(grpc_data_mode_t[]){GRPC_PERSISTENT, GRPC_END_LIST});

/* Write the data using dhA handle on a NFS server. */
grpc_data_transfer(&dhA,
NULL,
(const char *[]){"NFS://perceval.rush.aero.org/home/user/A2.dat", NULL},
NULL,
(grpc_data_mode_t[]){GRPC_STICKY, GRPC_END_LIST});
grpc_call(handle1,dhA,dhB,&dhC);

/* The data transfer of A has been asked previously */
grpc_call(handle2,dhA,dhC,&dhC);

/* Waiting for the end of the C data transfer proceeded after C computation. */
grpc_wait({dhC, NULL}, GRPC_WAIT_ALL);

/* The data is written on its output destination (NFS://britannia/home/user/C.out) */
grpc_data_transfer(&dhD, {NULL}, {NULL});

In this example (see Figure 5), we show how to use the GridRPC data management functions to prefetch
the data.

A.4.1 Input data

Data A is stored on the Kaamelot IBP server and will be used on karadoc to compute the first operation. This
data is also used as an entry to the second operation. Data B is located on the SRB server and will be used
as the second entry of the first operation only. The data prefetching is done by transferring A from the IBP
server to the perceval server in parallel of the first computation. The second operation will use the output
data C as input parameter with A that could be prefetched when the computation starts.

A.4.2 Output data

Data C is used as output for the two operations. It is first generated on karadoc and updated after the call on
perceval. It is not directly sent back to the client. To obtain it, the client proceed to an explicit data transfer
using the grpc_data_read() function.
A.5 Example with data migration

In this example (see Figure 6), we show how to use the GridRPC data management functions when the data needs to be stored inside the platform (or on a storage resources, this point depends on the middleware implementation). In this example we consider that the persistence data is kept in memory. Three \texttt{grpc\_call()} are performed, two on server \textit{karakdoc} and one on server \textit{perceval} working on the same data. The goal of the code here is to compute \( C = A \times (B + A \times B) \), which is done by doing the steps \( C = A \times B \), then \( C = B + C \) and finally \( C = A \times C \).

```c
grpc_function_handle_init(handle1,"karakdoc.aist.go.jp","*"ompson D
grpc_function_handle_init(handle2,"perceval.rush.aero.org","*");
grpc_function_handle_init(handle3,"karakdoc.aist.go.jp","*");
grpc_data_init(&dhA,
{\texttt{const char * [])}}}"memory://britannia.ens-lyon.fr/A", NULL),
GRPC_DOUBLE, (size_t [])\{3, 3, 0\},
{\texttt{grpc\_data\_mode\_t []}}\{GRPC\_STICKY, GRPC\_END\_LIST\});

grpc_data_init(&dhB,
{\texttt{const char * [])}}}"NFS://britannia.ens-lyon.fr/home/user/B.dat", NULL),
GRPC_DOUBLE, (size_t [])\{3, 3, 0\},
{\texttt{grpc\_data\_mode\_t []}}\{GRPC\_PERSISTENT, GRPC\_END\_LIST\});

grpc_data_init(&dhC,
{\texttt{const char * [])}}"memory://karadoc.aist.go.jp/data/C", NULL),
GRPC_DOUBLE, (size_t [])\{3, 3, 0\},
{\texttt{grpc\_data\_mode\_t []}}\{GRPC\_PERSISTENT, GRPC\_END\_LIST\});

grpc_call(handle1,dhA,dhB,&dhC);

grpc_data_init(&dhC,
{\texttt{const char * [])}}"memory://karadoc.aist.go.jp/C", NULL),
```
(const char * []){"memory://perceval.rush.aero.org", NULL},
GRPC_DOUBLE, (size_t []){3, 3, 0},
(grpc_data_mode_t []){GRPC_PERSISTENT, GRPC_END_LIST});
grpc_call(handle2,dhB,dhC,&dhC);
grpc_data_init(&dhC,
(const char * []){"memory://perceval.rush.aero.org/&C", NULL},
(const char * []){"NFS://britannia.ens-lyon.fr/home/user/C.out", NULL},
GRPC_DOUBLE, (size_t []){3, 3, 0},
(grpc_data_mode_t []){GRPC_PERSISTENT, GRPC_END_LIST});
grpc_call(handle3,dhA,dhC,&dhC);

Figure 6: Three RPC call with data management using persistence.

A.5.1 Input data

Data A will be used only on server karadoc, we can use the GRPC_STICKY parameter to keep the data on server karadoc (see Section A.3). Thus A is already available when the third grpc_call() is performed. The data B is used on two servers. With the GRPC_PERSISTENT mode the second grpc_call() implies that the data moves (or is duplicated) from server karadoc to server perceval.

A.5.2 Output data

Output data C is created on server karadoc. C moves (or is duplicated) from server karadoc to server perceval. C moves (or is duplicated) from server perceval to server karadoc and at the end, data C is sent back to the client.
## B Table of Functions

<table>
<thead>
<tr>
<th>Category</th>
<th>Function Name</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>lifecycle</td>
<td>grpc_data_init</td>
<td>4.3.1</td>
</tr>
<tr>
<td></td>
<td>grpc_data_unbind</td>
<td>4.3.5</td>
</tr>
<tr>
<td></td>
<td>grpc_data_free</td>
<td>4.3.6</td>
</tr>
<tr>
<td>mapping</td>
<td>grpc_data_memory_mapping_set</td>
<td>4.3.2</td>
</tr>
<tr>
<td></td>
<td>grpc_data_memory_mapping_get</td>
<td>4.3.2</td>
</tr>
<tr>
<td>container</td>
<td>grpc_data_container_set</td>
<td>4.3.2</td>
</tr>
<tr>
<td></td>
<td>grpc_data_container_get</td>
<td>4.3.2</td>
</tr>
<tr>
<td>read/write</td>
<td>grpc_data_transfer</td>
<td>??</td>
</tr>
<tr>
<td>load/save</td>
<td>grpc_data_load</td>
<td>4.3.8</td>
</tr>
<tr>
<td></td>
<td>grpc_data_save</td>
<td>4.3.8</td>
</tr>
<tr>
<td>wait</td>
<td>grpc_data_wait</td>
<td>4.3.4</td>
</tr>
<tr>
<td>reflection</td>
<td>grpc_data_get_info</td>
<td>4.3.7</td>
</tr>
</tbody>
</table>

Table 2: Functions defined in this document.
## Table of Types

<table>
<thead>
<tr>
<th>Category</th>
<th>Type Name</th>
<th>Possible values</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>data structure</td>
<td>grpc_data_t</td>
<td>structured type</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td>grpc_data_type_t</td>
<td>GRPC_INT</td>
<td>??</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_DOUBLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_COMPLEX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_FILE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_CONTAINER</td>
<td></td>
</tr>
<tr>
<td>data management</td>
<td>grpc_completion_mode_t</td>
<td>GRPC_WAIT_ALL</td>
<td>4.1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_WAIT_ANY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>grpc_data_mode_t</td>
<td>GRPC_VOLATILE</td>
<td>??</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_UNIQUE_VOLATILE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_PERSISTENT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_STICKY</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_UNIQUE_STICKY</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_END_LIST</td>
<td></td>
</tr>
<tr>
<td></td>
<td>grpc_data_info_type_t</td>
<td>GRPC_HANDLE</td>
<td>4.1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_INPUT_URI</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_OUTPUT_URI</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_MANAGEMENT_MODE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_SIZE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_TYPE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_LOCATIONS_LIST</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_STATUS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_COHERENT</td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>grpc_error_t</td>
<td>GRPC_NO_ERROR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_INVALID_TYPE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_INVALID_MODE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_INVALID_DATA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_INVALID_RANK</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_MALFORMED_URI</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_OTHER_ERROR_CODE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GRPC_NOT_INITIALIZED</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Types defined in this document.
Author contact information

Yves Caniou
University of Lyon / CNRS / ENS Lyon / INRIA / UCBL
Yves.Caniou@ens-lyon.fr

Eddy Caron
University of Lyon / CNRS / ENS Lyon / INRIA / UCBL
Eddy.Caron@ens-lyon.fr

Hidemoto Nakada
National Institute of Advanced Industrial Science and Technology
hide-nakada@aist.go.jp

Intellectual property statement

The OGF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the OGF Secretariat. The OGF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this recommendation. Please address the information to the OGF Executive Director.

Full copyright notice

Copyright (C) Open Grid Forum (2007). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the OGF or other organizations, except as needed for the purpose of developing Grid Recommendations in which case the procedures for copyrights defined in the OGF Document process must be followed, or as required to translate it into languages other than English. The limited permissions granted above are perpetual and will not be revoked by the OGF or its successors or assigns. This document and the information contained herein is provided on an “AS IS” basis and THE GLOBAL GRID FORUM DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.”