Proposal for a Data Management API within the GridRPC

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

February 6, 2008

Status of This Memo

This document (**version 0.5**) provides a recommendation to the Grid community on a proposed model and API for data management to GridRPC. Distribution is unlimited.

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.

Contents

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.

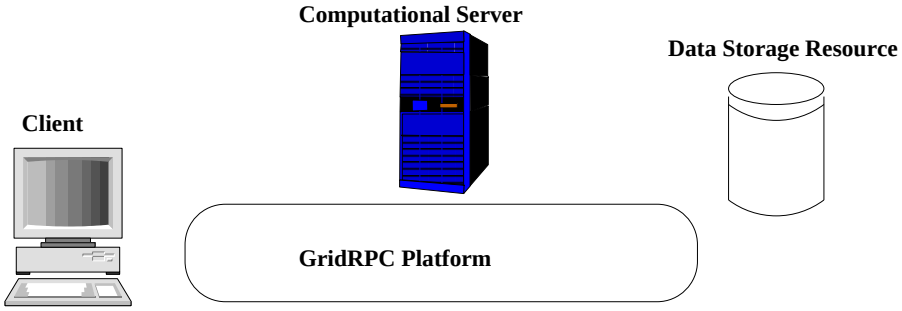

Figure 1: Data locations in the GridRPC model

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, *Data Handles* and *Storage Information*. 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. It relies on a data, or on a **grpc_data_handle_t** type and a **grpc_data_storage_info_t** type to access it.

Consequently, the **grpc_data_t** type can be seen as a structure containing the data itself and/or a **grpc_data_handle_t**. The **grpc_data_storage_info_t** type can also be stored in the **grpc_data_t** structure or it can also be stored and managed inside the GridRPC data middleware.

4.1.2 The grpc_data_handle_t type

A variable of this type represents a specific data. It is allocated by the user. After a *data handle* is initialized, it may be used in a server invocation. The lifetime of a *data handle* is determined when the user invalidates it. Data handles are created/allocated by simply creating a variable of this type.

4.1.3 The grpc_data_storage_info_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 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, or GRPC_STICKY, in which case the data cannot migrate but can be replicated.
- Information concerning the type of the data, as well as its size.

grpc_data_type_t	Definition
GRPC_INT	integer
GRPC_DOUBLE	double
GRPC_DOUBLE	complex
GRPC_ARRAY_OF_INT	array of int
GRPC_ARRAY_OF_DOUBLE	array of double
GRPC_ARRAY_OF_COMPLEX	array of complex
GRPC_MATRIX_OF_INT	matrix of int
GRPC_MATRIX_OF_DOUBLE	matrix of double
GRPC_MATRIX_OF_COMPLEX	matrix of complex
GRPC_CONTAINER_OF_GRPC_DATA	container of grpc_data_t

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 5.
  - char * protocol: one of the token {"NFS", "LOCAL_MEMORY", "IBP", "LOCAL_FS", "MIDDLEWARE", "HTTP"}, and gives some information on how to access the data (the list can be extended).
  - 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.
- The management mode is an enumerated type `grpc_data_mode_t` defined by the client. It is related to the following policy values:
  - GRPC_VOLATILE: used when the data is not kept inside the platform after a computation (the default usage for GridRPC API).
  - 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.
  - GRPC_STICKY_RETURN: used when a data is kept inside the platform but a copy is sent back to the client. Potential coherency issues may arise. The data item stays on the server where the computation has been done.
  - GRPC_PERSISTENT: used when a data has to be kept on the platform. The data is not sent back to the client and potential coherency issues may arise. Moreover, the data item can migrate or be replicated between servers depending on scheduling decisions.
  - GRPC_PERSISTENT_RETURN: used when a data is kept inside the platform but a copy is sent back to the client. This mode is useful when the client needs intermediate results. Potential coherency issues may arise. The data item can migrate or be replicated between servers depending on scheduling decisions.
- 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_ARRAY_OF_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.

4.1.4 The grpc_data_info_type 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
- GRPC_SIZE
- GRPC_TYPE (used to know to which type of the language the data corresponds)
- GRPC_LOCATIONS_LIST
- ...

Note: All information concerning a data can be stored within the GridRPC Data Management middleware, or within the `grpc_data_t` type. Nonetheless, this document does not focus on implementation. Furthermore, some information describing all the locations of a data can also be stored, for performance reasons, but that kind of features may be proposed later as an extension of the present document.

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=LOCAL_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).
- 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 **read** and **write** functions. The GridRPC data can also be **inspected** to get more information about

the status of the data or its location. 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 Section 5.

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:

```
grpc_error_t grpc_data_init( grpc_data_t * data,
                             char * URI_input,
                             char * URI_output,
                             grpc_data_type_t variable_type,
                             grpc_data_mode_t storage_mode );
```

input and **output** parameters are strings, which give the location on where to transfer the data from, and the location on where to possibly transfer the data to as explained previously.

Note that some of the parameters, such as the output parameter, can be unset. Furthermore, 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.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_TYPE	Specified type is not valid
GRPC_INVALID_MODE	Specified location is not valid
GRPC_OTHER_ERROR_CODE	Internal error detected

4.3.2 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:

```
grpc_error_t grpc_data_container_add( grpc_data_t * container, int rank,
                                      grpc_data_t * data );
grpc_error_t grpc_data_container_get( 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.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_TYPE	Specified type is not valid
GRPC_OTHER_ERROR_CODE	Internal error detected

4.3.3 The write function

This function writes a GridRPC data identified by `data` to the output location set during the init call in the output parameters fields. For commodity reasons, a diffusion mode and a list of additional servers on which the data has to be uploaded can be provided. In that case, the protocol defined during the init call is used. 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.

Function prototype:

```
grpc_error_t grpc_data_write(grpc_data_t data, grpc_data_diffusion_t mode,
                             <char * server_name>);
```

The `grpc_data_diffusion_t` can be defined by the set { GRPC_UNICAST, GRPC_BROADCAST }, which can be extended.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is not valid
GRPC_OTHER_ERROR_CODE	Internal error detected

4.3.4 The read function

This function reads a data stored inside the platform or on a specific storage server.

Function prototype:

```
grpc_error_t grpc_data_read(grpc_data_t * data);
```

After calling the `grpc_data_read` function, the data will be available in the GridRPC data type `data`, which will also still contain the Data Handle.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is not valid
GRPC_OTHER_ERROR_CODE	Internal error detected

4.3.5 The unbind function

When the user does not need a handle anymore, but knows that the data is 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:

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

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

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_OTHER_ERROR_CODE	Internal error detected

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:

```
grpc_error_t grpc_data_free(grpc_data_t data, 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, the `data` GridRPC data does not reference the data anymore.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is invalid
GRPC_OTHER_ERROR_CODE	Internal error detected

4.3.7 A function to get information on a grpc_data_t variable

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

Function prototype:

```
grpc_error_t grpc_data_getinfo(grpc_data_t data, grpc_data_info_type info_tag,
                               char ** info);
```

The kind of information that the function gets is defined by the **info_tag** parameter. 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.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is invalid
GRPC_OTHER_ERROR_CODE	Internal error detected

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:

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

Even if the GridRPC `data` contains the data in addition to data management information (data handle, size, type, etc.), only data information is saved in the location.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is invalid
GRPC_OTHER_ERROR_CODE	Internal error detected

5 Examples of Use

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.

5.1 Basic example

```
1  grpc_function_handle_init(handle1,"karadoc.aist.go.jp","*");
2  grpc_data_init(&dhA,"LOCAL_MEMORY://britannia.ens-lyon.fr/&A", NULL, DOUBLE, NULL);
3  grpc_data_init(&dhB,"NFS://britannia.ens-lyon.fr/home/user/B.dat", NULL, DOUBLE, NULL);
4  grpc_data_init(&dhC, NULL,"NFS://britannia.ens-lyon.fr/home/user/C.out", DOUBLE, NULL);
5  grpc_call(handle1,dhA,dhB,&dhC);
```

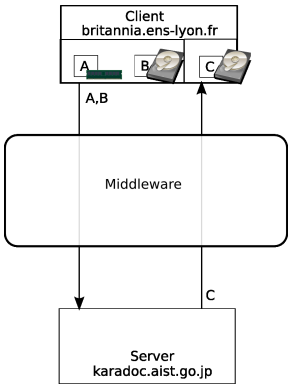

Figure 2: Simple RPC call with input and output data.

In this example (see Figure 2), we show 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.

5.1.1 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.

5.1.2 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.

5.2 Storage 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.

```
grpc_function_handle_init(handle12,"karadoc.aist.go.jp","*");
grpc_data_init(&dhA,"IBP://kaamelott.cs.utk.edu/1212#A.dat/ReadKey/READ", "NFS://britannia.ens-lyon.fr/home/user/A.dat", GRPC_DOUBLE, NULL);
grpc_data_init(&dhB,"SRB://carmelide.ens-lyon.fr/COLLECTION/Simulations/B.dat", "IBP://kaamelott.cs.utk.edu/1213#B.dat/WriteKey/WRITE", GRPC_DOUBLE, NULL);
grpc_data_init(&dhC, NULL,"NFS://britannia.ens-lyon.fr/home/user/C.out", GRPC_DOUBLE, NULL);
grpc_call(handle1,dhA,dhB,&dhC);
```

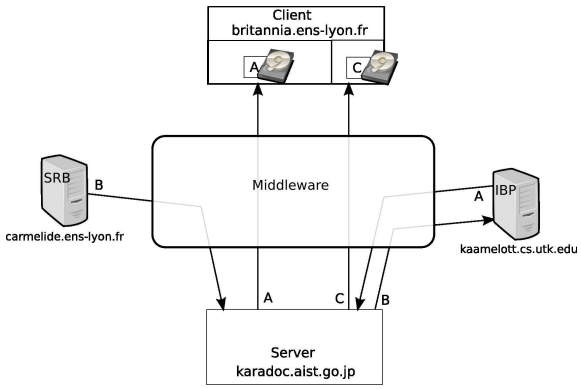

Figure 3: Simple RPC call with input and output data using external storage resources.

5.2.1 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.

5.2.2 Output Data

Output data `C` is sent back to the client.

5.3 Example with GRPC_STICKY mode

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*.

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.

5.3.1 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 `grpc_data_init` for this data requires only an input location and the GRPC_STICKY mode.

5.3.2 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 `grpc_call()`.

```
grpc_function_handle_init(handle13,"karadoc.aist.go.jp","*");
grpc_data_init(&dhA,"LOCAL_MEMORY://britannia.ens-lyon.fr/&A","LOCAL_MEMORY://karadoc.aist.go.jp", GRPC_DOUBLE, GRPC_STICKY);
grpc_data_init(&dhC, "NFS://britannia.ens-lyon.fr/home/user/C.in", "LOCAL_MEMORY://karadoc.aist.go.jp", GRPC_DOUBLE, GRPC_STICKY);

for(i=0;i<n+1;i++)
{
   if( i==1 )
      grpc_data_init(&dhC, "LOCAL_MEMORY://karadoc.aist.go.jp",NULL, DOUBLE, STICKY);
   if( i==n )
      grpc_data_init(&dhC, "LOCAL_MEMORY://karadoc.aist.go.jp","NFS://britannia.ens-lyon.fr/home/user/C.out", GRPC_DOUBLE, GRPC_VOLATILE);

   grpc_call(handle1,dhA,dhC,dhC);
}
grpc_data_free(dhA);
grpc_data_free(dhC);
```

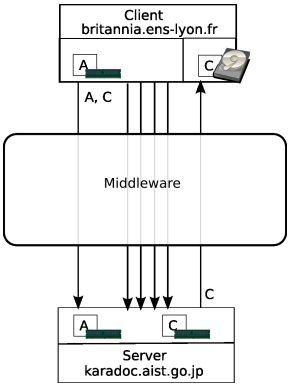

Figure 4: GridRPC call with data management using persistence through the GRPC_STICKY mode.

5.4 Example with GRPC_PERSISTENT mode

In this example (see Figure 5), 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 `grpc_call()` are performed, two on server *karadoc* and one on server *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 $C = A \times C$.

5.4.1 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 5.3). 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`.

5.4.2 Output Data

Output data `C` is created on server `karadoc` (`grpc_call()` line 7). At Line 8, `C` moves (or is duplicated) from server *karadoc* to server *perceval*. Line 9, `C` moves (or is duplicated) from server *perceval* to server *karadoc* and at the end, data `C` is sent back to the client.

Author Contact Information

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

```
grpc_function_handle_init(handle1,"karadoc.aist.go.jp","*");
grpc_function_handle_init(handle2,"perceval.rush.aero.org","*");
grpc_function_handle_init(handle3,"karadoc.aist.go.jp","*");
grpc_data_init(&dhA,"LOCAL_MEMORY://britannia.ens-lyon.fr/&A", NULL, GRPC_DOUBLE, GRPC_STICKY);
grpc_data_init(&dhB,"NFS://britannia.ens-lyon.fr/home/user/B.dat", NULL, GRPC_DOUBLE, GRPC_PERSISTENT);
grpc_data_init(&dhC, NULL, "LOCAL_MEMORY://karadoc.aist.go.jp", GRPC_DOUBLE, GRPC_PERSISTENT);
grpc_call(handle1,dhA,dhB,&dhC);
grpc_data_init(&dhC, "LOCAL_MEMORY://karadoc.aist.go.jp", "LOCAL_MEMORY://perceval.rush.aero.org", GRPC_DOUBLE, GRPC_PERSISTENT);
grpc_call(handle2,dhB,dhC,&dhC);
grpc_data_init(&dhC, "LOCAL_MEMORY://perceval.rush.aero.org","NFS://britannia.ens-lyon.fr/home/user/C.out", GRPC_DOUBLE, GRPC_PERSISTENT);
grpc_call(handle3,dhA,dhC,&dhC);
```

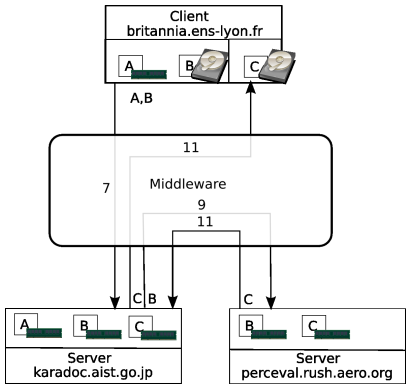

Figure 5: Three RPC call with data management using persistence.

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

Full Copyright Notice

References

[1] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee and H. Casanova. : A GridRPC model and API for End-Users Applications, Global Grid Forum, July 21, 2005, GFD-R.052

[2] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany and R. Wolski : The Internet Backplane Protocol: Storage in the network, Storage in the network. In NetStore '99: Network Storage Symposium. Internet2, October 1999.

[3] C. Baru, R. Moore, A. Rajasekar and M. Wan : The SDSC Storage Resource Broker, In Procs. of CASCON'98, Toronto, Canada, 1998

[4] D.C. Arnold, D. Bachmann and J. Dongarra : Request Sequencing: Optimizing Communication for the Grid, Lecture Notes in Computer Science 2003, vol 1900, pp 1213

[5] B. Del Fabbro, D. Laiymani, J.-M. Nicod, and L. Philippe: A Data Persistency Approach for the DIET Metacomputing Environment, Int. Conf. on Internet Computing, IC'04, 2004

[6] M. Beck, D.C. Arnold, A. Bassi, F. Berman, H. Casanova, J. Dongarra, T. Moore, G. Obertelli, J. Plank, M. Swany,S. Vadhiyar and R. Wolski: Middleware for the Use of Storage in Communication, IN Parallel Computing, vol 28, number 12, pp 1773-1788, 2002

[7] Y. Aida, Y. Nakajima, M. Sato, T. Sakurai, D. Takahashi and T. Boku : Performance Improvement by Data Management Layer in a Grid RPC System, IN the First International Conference on Grid and Pervasive Computing (GPC2006), pp.324-335, Taiwan, May 3-5, 2006

[8] G. Antoniu, L. Bougé and M. Jan. : JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid, IN Scalable Computing: Practice and Experience, Vol. 6(3):45-55, September 2005