Proposal for a Data Management API within the GridRPC

GRAAL Research Team

February 11, 2005

Abstract

This document follows the document produced by the GridRPC-WG on GricRPC Model and API for End-User applications [1]. This new document aims at completing the GridRPC API with a Data Management mechanisms and API. This document also focuses on the client side and does not include features and capabilities for building data management middle-wares.

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 those issues related to data management inside a GridRPC platform or on a data storage server.

The motivation of the data management extension is to provide explicit functions to manipulate the data exchange between a data storage service, a GridRPC platform, and the client. The GridRPC API defines a RPC mechanism to access Network Enabled Servers. However, a computation is composed of a program, the application in our case, and input and output data. 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 where the data used in the computation are stored: on a 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. For this reason, the GridRPC API must be extended to provide functions for explicit and simple data management.

In the first section, we present a motivation for the data management and in Section 3, the data management model proposed is developed. The main contribution of this document is given in Section 4 where we describe our proposal for a data management API. Current implementations of this model or part of this model are described in the joint document.

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 GridRPC model. Using this explicit management, the client will avoid useless transfer of these large data. However, the client may not want to, or may not know how to manage data then the basic functions must not be modified by this extension. To illustrate the motivation of data management, we give some examples describing when it could be used.

In a GridRPC environment, data could be placed either on the client host, a data storage server, a computational server or the GricRPC platform, as shown in Figure 1. When clients do not need to manage their

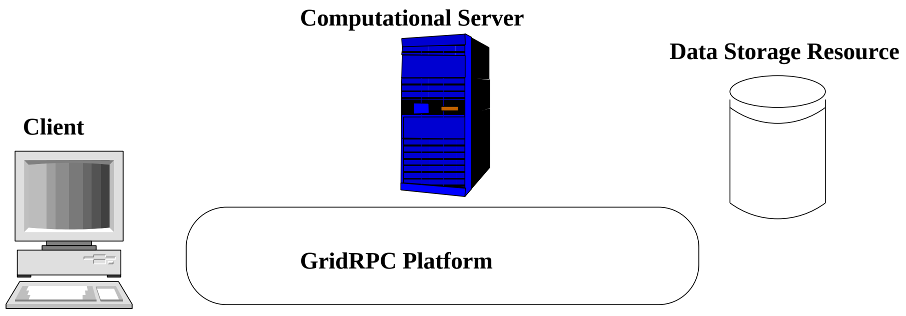

Figure 1: Data positions in GridRPC model

data, then the basic GridRPC API is sufficient. On each `grpc_call`, data will be exchanged between a client and the computational server used. However, to minimize data transfers, clients need data management functions.

Two cases could be considered: internal data and external data. Internal data are managed inside the GridRPC platform. Their placement depends on computations and it is transparent to clients. 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 $C =^t (A * B)$, 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: $W = A * B$ 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 could be found when the results of a simulation are sent to a graphical viewer as done in most Problem Solving Environments. An example of internal data management is the DTM infrastructure used in DIET [5]. This approach is suitable, but not limited to, for intermediate results to be reused in case of request sequencing.

External data are placed on servers, like data repositories. These servers are not registered inside the platform but can be directly accessed to get/put the data. The use of such data implies several data transfers if the client uses the basic API: the client must get the data and then send it to the platform when issuing its `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 server to the data storage. Examples of such Data storage servers are IBP [2] or SRB [3]. An example of this approach is the Distributed Storage Infrastructure of NetSolve [6]. This approach is well suited for, but not limited to, long life persistent data.

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. Client can use already existing data by just initializing a data reference with the data identification given in the storage service. The GridRPC platform provides functions to transfer data between clients and storage servers.

In the internal data case, the data management service tries to load/store the data inside the GridRPC platform, on the computational servers. Two different cases must be distinguished: explicit knowledge of the computational server and transparent data management. The first case can be managed with the same function defined for the external data case as the client knows where to place or transfer its data.

In the second case, data placement lays on the GridRPC platform. Data are also identified by references. The GridRPC platform provides functions to initialize these references and get the data. Transfers between servers are done in a transparent way.

These two approaches are complementary in the data management model proposed here. The GridRPC platform and the data storage interact to implement the three levels model (client, internal, external). 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 an other data storage server.

In both cases, it is mandatory to identify the data as there will not be any data pointer to the client's data left. Data external to the client, stored either in the platform or on storage servers, will be identified by **Data Handles**. When a computational server receive a data handle, it must knows where is the data placed and where the client want to save it after computation. Thus data handles must record the original location of the data and the destination of the data.

4 Data Management API

Defining the *Data Handle* and the functions used to manage it.

4.1 Data Handle

We introduce the notion of `data handle` which includes both information about the data themselves (e.g. type, size) as well as information on their location (e.g. URL of a specific server, link with a Storage Resource Broker). A data handle is essentially a reference to a data that may resides anywhere and a reference of the future data storage for the data. Data and data handle can be created separately. By managing data handles, clients do not have to know where the data is currently stored.

grpc_data_handle_t

Variables of this data type represent a specific data bound to a specific location. They are 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 the following type.

grpc_data_t

Variables of this type represent a specific data which could be local or remote. Remote Data could be identified on a model "url:data".

grpc_data_storage_t

Variables of this type represent a specific data storage (storage server or computational server).

4.2 Data Management Functions

As data handle records two informations, what data it references and where this data will saved to after computation, the data management API propose two functions to initialize indecently these values. The **init** function sets the data handle to the data it identifies. The **bind** function sets the data handle to location where the data will be saved. Using these two functions, 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** or **write** functions. The data handle can also be **inspected** to get more information about the status of the data or its location. Finally, one can **free** the data. This will unbind the handle and the data.

To be discussed: 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.

To avoid useless complexity, the data management API must be used only when necessary. Thus, the function we propose have no interaction with the functions of the basic GridRPC API. We propose to use the init/bind operations only when data will be stored inside the platform or in storage resources.

Examples of the use of this API is given in Section 4.3.

4.2.1 The init function

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

Function prototype:

```
grpc_error_t grpc_data_handle_init(
                        grpc_data_handle_t *dh,
                        grpc_data_t data);
```

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

4.2.2 The bind function

The **bind** function provides a way for the client to store a data on a specified server after a computation. The server may be either a storage server or a computational server if the client plans to reuse this data on the server. It is not mandatory to bind data handles referencing input parameters if the data are not persistent and must be destroyed after computation. Data handles referencing output parameters must be bound to know where this data will be stored.

Depending on the destination of the data, the storage parameter will be used differently:

client host clients does not need to call *bind*, or it could be set to `NULL` or to the client address.

storage server This can be a remote storage resource as IBP, SRB, NFS. In this case, the location may be identified on a url-like base. The client will give the storage server identifier.

computational server This can be the function_handle_t argument which is determined with the `grpc_function_handle_init` function. The client will give the computational server identifier.

platform This is the case where the computational server is dynamically chosen by the middleware (using the `grpc_function_handle_default` function). The location argument is set to `GRPC_DATA_MNT_DEFAULT`. In this case, data movements inside the platform are transparent to the client.

Function prototype:

```
grpc_error_t grpc_data_handle_bind(
                     grpc_data_handle_t *dh,
                     grpc_data_storage_t storage);
```

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

4.2.3 The write function

This function writes a data identified by `dh` to the specific location given by `storage`.

Function prototype:

```
grpc_error_t grpc_write(grpc_data_handle_t dh, grpc_data_storage_t storage);
```

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

To be discussed: implicit bind when writing a data. If we write a data to some place, we need to register this location in the data handle. Implicit bind will avoid using both functions on every call to write.

4.2.4 The read function

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

Function prototype:

```
grpc_error_t grpc_read(grpc_data_handle_t dh, grpc_data_t *data);
```

After calling the `grpc_read` function, the data identified by `dh`, will be available in the data type given by `data`.

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

4.2.5 The free_data function

This function frees the data identified by `dh` data handle. There is no particular assumption on what to do with the data but `dh` data handle will not identify it anymore. This function may be used to explicitly erase the data on a storage resource.

Function prototype:

```
grpc_error_t free_data(grpc_data_t dh);
```

After calling this function, the `dh` data handle does not references 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

The inspect_data function This function inspects the data identified by `dh` data handle. This functions returns information on data characteristics, status, and location.

Function prototype:

```
grpc_error_t grpc_inspect_data(grpc_data_t dh, grpc_data_t *data);
```

After calling this function, the `data` structure will be filled with up-to-date information on the data. The data will not be moved by using this function.

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 Examples of Use

In this section, we give use examples of the use the data management API 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.

4.3.1 Simple examples without data persistence

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 resources.

Input Data Here, we illustrate the way to send a local data or a data stored in a storage resource to the GridRPC platform without any persistence. In this example, the client issues a call with three input data `A`, `B` and `C`.

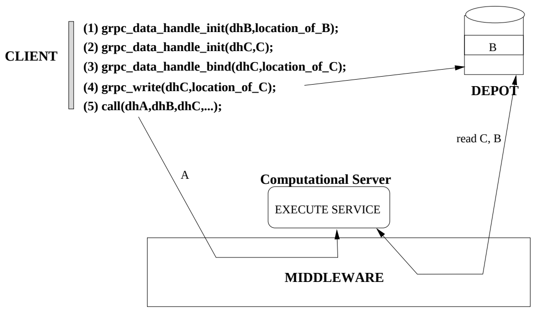
CLIENT
(1) grpc_data_handle_init(dhB,location_of_B);
(2) grpc_data_handle_init(dhC,C);
(3) grpc_data_handle_bind(dhC,location_of_C);
(4) grpc_write(dhC,location_of_C);
(5) call(dhA,dhB,dhC,...);
B
DEPOT
read C, B
A
Computational Server
EXECUTE SERVICE
MIDDLEWARE

Figure 2: Simple RPC call with input data.

In this example, A is local to the client, only an `init` operation is useful because A won't be managed by the platform (see (1)- 2). B data is already stored on a storage resource external to the GridRPC middleware. The `init` function allows to give the location of this data to the platform (see (2)-2). It is also sufficient because there is no need for data persistence. The third input data is copied by the client to a storage resource before being sent to the platform. In this case, the `init` operation coupled with the `write` operation is necessary(see (3),(4)-2) .

Output Data All output data are sent back to the client when no data conservation is needed by the client when he builds its problem. In this example, the client issues a call with `A` as input data and `C` as output data.

The Figure 3 describes the way to define output data. Because no conservation is required, all persistent operations are not required.

4.3.2 Add input data

In these cases, Input Data is originally placed on the client host. The client needs to place its data to optimize traits use either in a storage resource or on a specific server host or just want to leave it inside the GridRPC

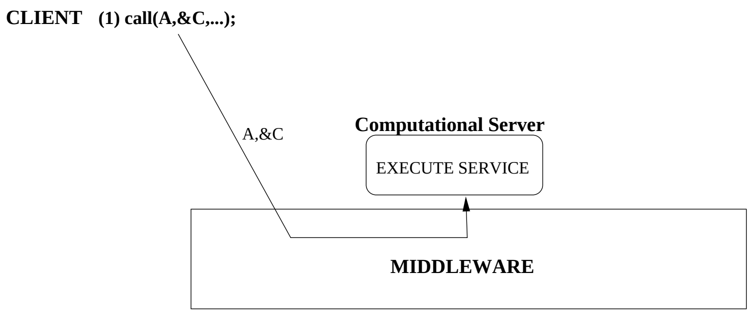

Figure 3: simple RPC call with output data

platform. These three possibilities are described in the followings paragraphs.

Add input data on a storage resource Figure 4 describes how to use the API when the data is written to a storage resource prior to a call to the `grpc_call` function.

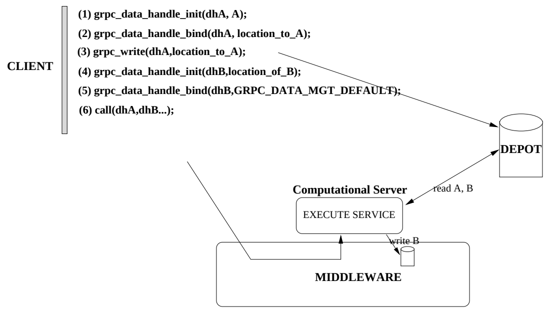
CLIENT
(1) grpc_data_handle_init(dhA, A);
(2) grpc_data_handle_bind(dhA, location_to_A);
(3) grpc_write(dhA,location_to_A);
(4) grpc_data_handle_init(dhB,location_of_B);
(5) grpc_data_handle_bind(dhB,GRPC_DATA_MGT_DEFAULT);
(6) call(dhA,dhB...);
DEPOT
read A, B
Computational Server
EXECUTE SERVICE
write B
MIDDLEWARE

Figure 4: Input mode and url

In this figure, A is initially stored in the client local machine and then sent to a storage resource (see (1,2,3)- 4). The B data, is already stored on a external machine but need to be bound inside the platform. So, the `init` function assign a data handle to this data location and then the `bind` operation binds it to the platform,(see (4,5)- 4) . The `data handle of` B refers to its location inside the platform.

In this case, the data may be used several times by the client by using the `dhA` data handle in further calls.

Add input data on a specific server Figure 5 describes how to use the API when the input data (A) is written on a specific server before the `grpc_call` function.

When using a `grpc_function_handle_init` to initialize a function handle, before a `grpc_call`, the client will know the server that will be chosen to compute its problem.

If this server and the data are used several times on a request sequencing mode, the client will benefit of placing and leaving its data on the desired server.

Add input data in the platform In this case, the client owns a data `A` locally on its host but he does not know where the computation will be executed. This is the case when a function handle is initialized with `grpc_function_handle_default`. Here, the data handle is created and bound to the platform (see Figure 6).

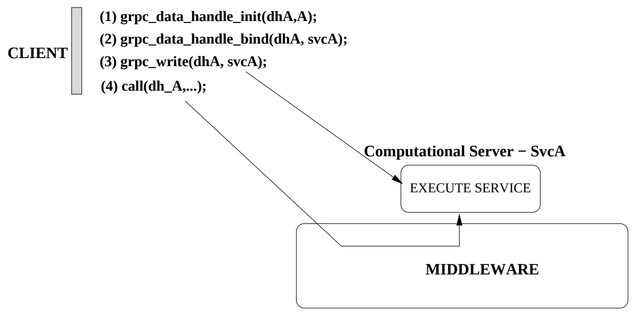
CLIENT
(1) grpc_data_handle_init(dhA,A);
(2) grpc_data_handle_bind(dhA, svcA);
(3) grpc_write(dhA, svcA);
(4) call(dh_A,...);
Computational Server − SvcA
EXECUTE SERVICE
MIDDLEWARE

Figure 5: Input mode and specific server.

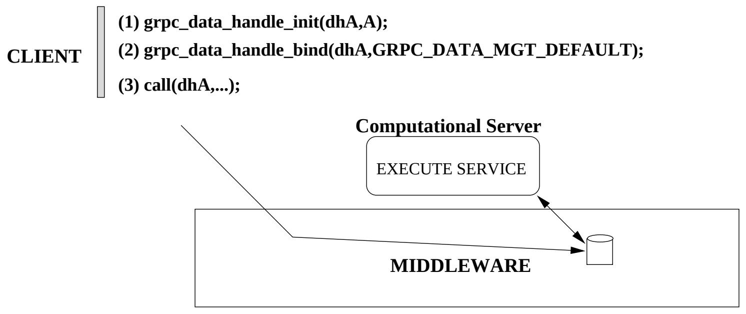
CLIENT
(1) grpc_data_handle_init(dhA,A);
(2) grpc_data_handle_bind(dhA,GRPC_DATA_MGT_DEFAULT);
(3) call(dhA,...);
Computational Server
EXECUTE SERVICE
MIDDLEWARE

Figure 6: Input mode et platform.

The `dhA` data handle will identify the `A` data after issuing the `write` call. But the data is still on the client host until a call to `grpc_call` is issued as the destination server is unknown. The data transfer is done during the `grpc_call`, when the destination server is chosen.

Discussion These figures show how to add an input data using the GridRPC API. An other aspect of the data management is the way to use a data stored onto a specific server, a storage resource or the middleware in next sessions or by others clients. If we consider that a data can survive to a client session, a data already stored inside the platform and not freed after the session can be used later. The way the the data handle is stored and restored it is not of our concern. It is clearly middleware dependent. So, when a data is already handled, it is not necessary to bind it one more time. The handle will be directly passed in the `grpc_call` function.

4.3.3 Output data

As we explained in beginning of the document, data management is mainly useful for temporary data in case of request sequencing. Temporary data are typically output data of a first request used as input data of a second request. Output data are generated by a computation on a server of the GridRPC platform. They may be kept on the same server, moved to a storage resource or read by the client.

Write an output data into the platform with no return Most of the time, temporary results are not needed by the client. In this case, the result of the computation is not returned to the client (see Figure 7).

By setting the `GRPC_DATA_MGT_DEFAULT` location for its data, the client will leave its placement to the GridRPC platform. Then, he will just use the `dhA` data handle as a reference to the data.

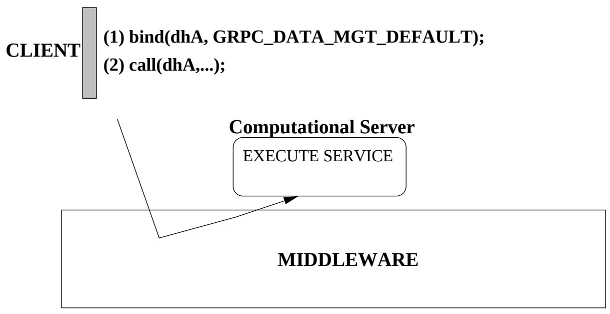

Figure 7: OUT mode and platform.

Write an output data into the platform with return In this case, the result of the computation is returned to the client, (see Figure 8). The `read` operation is necessary to get a copy of the data stored.

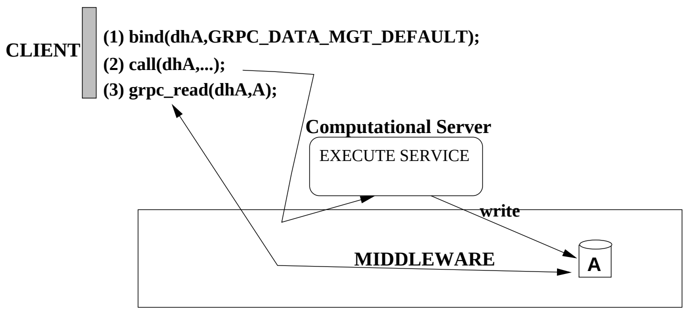

Figure 8: OUT mode and platform.

In this case, the data `A`, identified by `dhA` is generated by the computational server and leaved in place for a future computation. However the client needs the intermediate results and read it from the platform. He just needs the `dhA` data handle to get or manage this data.

Write an output data to a specific server with return In this case, the result of the computation is returned to the client, (see Figure 9). The `read` operation is necessary to get a copy of the data stored.

Write an output data in a storage server with return Figure 10 describes how to bind a data handle with a storage resource with the data is the result of a computation. The `read` operation is necessary to get a copy of the stored data.

Discussion We distinguish an out parameter with return value (copy) to the client with the out parameter with no return. In fact, this view is particularly helpful for managing intermediate results as seen Section 2. So, by default the output data is not returned to the client. When a data has to be returned to the client a copy is done when the explicit `read` function is used.

4.4 Server side

The API is build for a end-user. But, the middleware has to manage some data features for example the data movement between computational servers. Particularly, when data is moved between servers, the

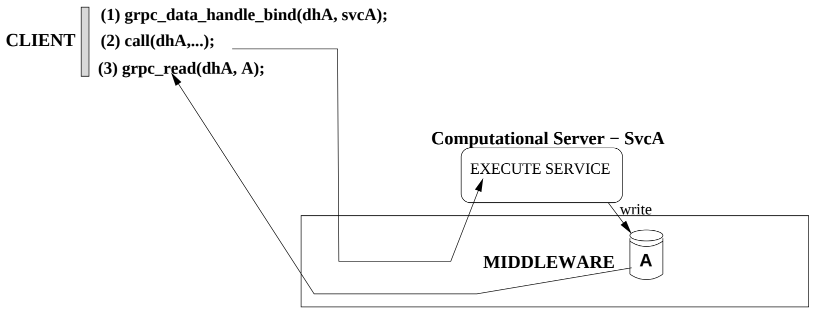

Figure 9: Input mode and url

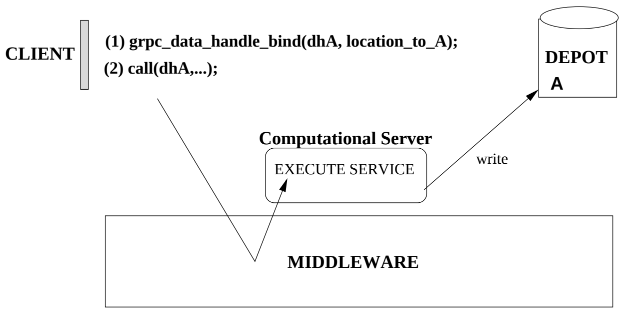

Figure 10: OUT mode and URL.

data handle associated to the data is no more valid since it is bound to a location that has changed. So, the platform must provide a mechanism to locate data or to change dynamically the data handle in order to keep dependencies between the handle bound in the client level and the new data location. This is particularly important for the `free_data` function.

4.5 Code Examples

Here we see how a code could be written.

without data persistence This is a standard case. See document titled `The End-user and Middleware APIs for GridRPC`, page 5.

simple RPC call where the input and output Data remain on the server Let consider a simple matrix multiplication $C = A * B$ where input data will be stored using the GridRPC data API. The three possible cases are described above.

The three examples describe a simple RPC code example to add input and output data in the following context: data sent to the platform (see Figure 11), to a storage resource (see Figure 12) or to a specific server (see Figure 13). In all cases, the way that the middleware manages the data is not of our concern. In fact, particularly when data is sent to the platform or in a storage resource, the way that computational servers get the data is left to the responsibility of the middleware.

In the first code example (see Figure 11), A and B are persistent inside the platform whereas the result of the computation C is persistent but isn't copied back to the client.

In the second code example(see Figure 12), A is local to the client but sent to a storage resource, B is already stored on a storage resource but not handled by the gridRPC API. Note that for B, there is no need of

```
...
double *A=NULL, *B=NULL, *C=NULL;
grpc_function_handle_t *handle;
grpc_data_handle_t *dh_A, *dh_B, *dh_C;
... // A,B initialization
grpc_initialize();

grpc_function_handle_default(handle,''*'');
grpc_data_handle_init(dh_A,A);
grpc_data_handle_bind(dh_A,GRPC_DATA_MGT_DEFAULT); // this lets the platform manages the data
grpc_data_handle_init(dh_B,B);
grpc_data_handle_bind(dh_B,GRPC_DATA_MGT_DEFAULT);
grpc_data_handle_bind(dh_C,GRPC_DATA_MGT_DEFAULT);
grpc_call(handle,dh_A,dh_B,dh_C); // the write operation is realized at a lower level( inside the call
...
grpc_free_data(dh_A);
grpc_free_data(dh_B);
grpc_free_data(dh_C);

grpc_finalize();
```

Figure 11: Simple RPC call with A and B sent into the platform.

the `bind` operation since B wouldn't be moved from the storage server to the platform or a computational server, the `bind` should be use only if B will be persistent inside the platform. C will be persistent inside the platform and copied back to the client.

```
...
double *A=NULL, *B=NULL, *C=NULL;
grpc_function_handle_t *handle;
grpc_data_handle_t *dh_A, *dh_B,*dh_C;
... // A,B initialization
grpc_initialize();

grpc_function_handle_default(handle,''*'');
grpc_data_handle_init(dh_A,A); // A is local
grpc_data_handle_bind(dh_A,''location_to_A'');
grpc_write(dhA,''location_to_A'')); // write data to the storage resource handled by dh_A
grpc_data_handle_init(dh_B,''location_from_B'');
grpc_data_handle_bind(dh_C,GRPC_DATA_MGT_DEFAULT);
grpc_call(handle,dh_A,dh_B,dh_C);
...
grpc_read(dh_C,C); // read C
grpc_free_data(dh_A);
grpc_free_data(dh_B);
grpc_free_data(dh_C);

grpc_finalize();
```

Figure 12: Simple RPC call with A and B persistent and stored on a storage resource.

In the third code example (see Figure 13, A and B are sent to a specific server and will be persistent, C will also be persistent on this server with no copy back to the client.

Two successive RPCs on the same server Let consider the following sequence $C = A * B$ then $D = C + A$. An optimization consist in storing C and A inside the platform. Here, the `grpc_function_handle_init` function provides a function handle on a specific server. Data transfers are explicitly managed by the client. The C matrix is an intermediate results and so it has not to be sent back to the client. In this example, the

```
...
double *A=NULL, *B=NULL, *C=NULL;
grpc_function_handle_t *handle;
grpc_data_handle_t *dh_A, *dh_B, *dh_C;
... // A,B initialization
grpc_initialize();
grpc_function_handle_init(handle,''srvA'',''*'');
grpc_data_handle_init(dh_A,A);
grpc_data_handle_bind(dh_A,''srvA'');
grpc_write(dhA,''srvA''); // write data to the storage resource handled by dh_A
grpc_data_handle_init(dh_B,B);
grpc_data_handle_bind(dh_B,''srvA'');
grpc_write(dh_B,''srvA'');
grpc_data_handle_bind(dh_C,''srvA'');
grpc_call(handle,dh_A,dh_B,dh_C);
...
grpc_free_data(dh_A);
grpc_free_data(dh_B);

grpc_finalize();
```

Figure 13: Simple RPC with A and B sent to a specific server.

utilization of the `read` function shows that a client must explicitly tell the platform to bring back a data. This is particularly interesting for intermediate results. We also notice that B is not persistent. So, none of the data management functions are useful for B. In the example, B is sent inside the call but depending on the platforms implementations it can also be sent before the call. D is set as persistent and copied back to the client.

Two successive RPCs on different servers Let consider the following sequence $C = A*B$ then $D = C+A$. An optimization consist in storing C and A inside the platform. The code example described below, we note that the D matrix is not persistent. So, we do not need to use the `bind` function. Data transfers between the computational servers are performed by the platform and so data do not need to be explicitly sent by the client. We point out that this solution needs a data movement mechanism implemented by the platform. Moreover, the handle of the transferred data is no more valid since it refers to a location that has changed. We bring three possible solutions: first, if the platform does not provide a sophisticated data movement mechanism, all data must be transferred back to the client (`read` function) and then sent from the client to the platform (new `bind` and `write` operations). Then, a data location infrastructure must be developed in order to keep a link between the handle and the new location of the data (for example a corresponding table between the initial handle and its new internal references). Finally, the data `handle` generated must be bound to a location. However this information is necessary but not sufficient. For example, when the platform has already implemented its own data management policy, a "local" data identifier already exists. Thus, the definition of what contains a `data handle` is let to the responsibility of the developers of each GridRPC platform.

References

[1] Nakada, H. and Matsuoka, S. and Seymour, K. and Dongarra, J. and Lee, C. and Casanova, H. : A GridRPC model and API for End-Users Applications, Global Grid Forum, December 2003, GWD-R

[2] J. Plank and M. Beck and W. Elwasif and T. Moore and 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 and R. Moore and A. Rajasekar and M. Wan : The SDSC Storage Resource Broker, In Procs. of CASCON'98, Toronto, Canada, 1998

```
...
double *A=NULL, *B=NULL, *C=NULL, *D=NULL;
grpc_function_handle_t *handle;
grpc_data_handle_t *dh_A, *dh_C, *dh_D;
... // A,B initialization
grpc_initialize();

grpc_function_handle_init(handle,''srvA'',''*'');
grpc_data_handle_init(dh_A,A);
grpc_data_handle_bind(dh_A,''srvA'');
grpc_write(dhA,''srvA''); // write data to the storage resource handled by dh_A
grpc_data_handle_bind(dh_C,''srvA'');

grpc_call(handle,dh_A,B,dh_C);


grpc_function_handle_init(handle1,''srvA'',''+'');

grpc_data_handle_bind(dh_D,''srvA'');
grpc_call(handle1,dh_A,dh_C,dh_D);
...
grpc_read(dh_D,&D);

grpc_free_data(dh_A);
grpc_free_data(dh_B);
grpc_free_data(dh_C);
grpc_free_data(dh_D);


grpc_finalize();
```

Figure 14: Two successive calls on the same server.

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

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

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

```
...
double *A=NULL, *B=NULL, *C=NULL, *D=NULL;
grpc_function_handle_t *handle;
grpc_data_handle_t *dh_A, *dh_C;
... // A,B initialization
grpc_initialize();

grpc_function_handle_init(handle,''srvA'',''*'');

grpc_data_handle_init(dh_A,A);
grpc_data_handle_bind(dh_A,''srvA'');
grpc_write(dhA,''srvA''); // write data to the storage resource handled by dh_A

grpc_data_handle_bind(dh_C,''srvA'');

grpc_call(handle,dh_A,B,dh_C);

grpc_function_handle_init(handle1,''srvB'',''+'');

grpc_call(handle1,dh_A,dh_C,&D);

grpc_free_data(dh_A);
grpc_free_data(dh_B);
grpc_free_data(dh_C);


grpc_finalize();
```

Figure 15: Two successive calls on different servers.