Data Management API within the GridRPC

Yves Caniou, Eddy Caron, Gaël Le Mahec, Hidemoto Nakada

GFD-R-P.186
GRIDRPC-WG
gridrpc-wg@ogf.org

Yves Caniou, University of Lyon / CNRS / ENS
Lyon / INRIA / UCBL
Eddy Caron, University of Lyon / CNRS / ENS
Lyon / INRIA / UCBL / SysFera
Gaël Le Mahec, University of Picardie
Jules Verne
Hidemoto Nakada, National Institute of
Advanced Industrial Science and Technology
June 6, 2011

Data Management API within the GridRPC

Status of This Document

Grid Final Draft (GFD)

Copyright Notice

Abstract

This document follows the document produced by the GridRPC-WG on GridRPC Model and API for End-User applications. This new document aims to complete the GridRPC API with Data Management mechanisms and API.

This document is not intended to provide features and capabilities for building data management middleware. Its goal 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 users, components of a GridRPC platform and storage resources 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. All functions in the API have been thought to be called by each part in a GridRPC platform (client, agent and server) if needed.

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 [**?**], it is out of the scope of this document to discuss the *implementation* of the API described in this document, which focuses on data management mechanisms inside a GridRPC platform.

The motivation of the data management extension is to provide explicit functions to handle data exchanges between data storage resources, components of a GridRPC platform and the user. 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 data may be large in grid environments, it is mandatory to optimize transfers of large data by avoiding 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 user 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.

1.1 Security Consideration

As the GRPC API is to be implemented on different types of Grid (and Cloud) middleware, it does not specify a single security model, but rather provides hooks to interface to various security models. A GRPC implementation is considered secure if and only if it fully supports (i.e. implements) the security models of the middleware layers it builds upon, and neither provides any (intentional or unintentional) means to by-pass these security models, nor weakens these security models' policies in any way.

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.

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 transfered 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

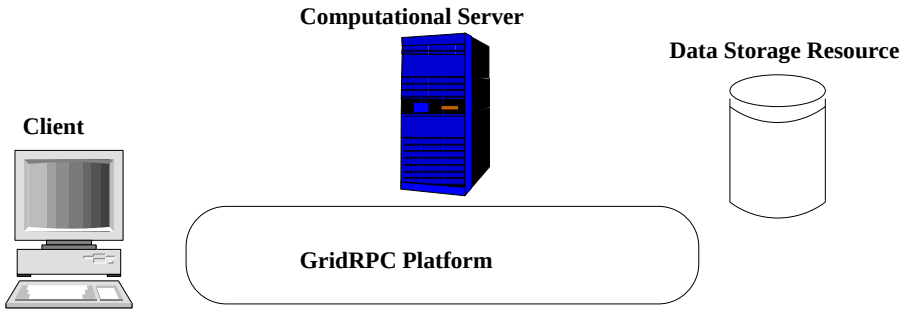

Figure 1: Data locations in the GridRPC model

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 not be 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 [**?**] or SRB [**?**]. Among the different available examples of this approach, we can cite: (1) the Distributed Storage Infrastructure of NetSolve [**?**]; (2) the utilization of JuxMem [**?**] or DAGDA [**?**] in DIET. 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 [**?**], 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 [**?**], and the data layer omniStorage in omniRPC [**?**]. 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 term *GridRPC data* 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 one 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 [**?**], 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 computational data, or contains a reference on the computational data, which we call a *Data Handle*, as well as some *Storage Information*.

In the following, 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 from now on 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 contain some information about the data itself (*e.g.,* type, dimension) as well as information on its locations and the protocols used to access it (*e.g.,* at least an 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:

- A unique identifier, of type `grpc_data_handle_t`. This is created and used internally by the GridRPC middleware to manage the data. Depending on the data management possibilities of the GridRPC middleware, the data itself can also be stored.

- Two NULL-terminated lists of URIs, one to access the data and one to record the data (for example an OUT parameter to transfer at the end of a computation, or to prefetch the data.)
- 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 dimensions.

Details on Storage Information

- *URI*: it defines the location where a data is stored. URI formalism is described in [**?**]. It can be built like "protocol:[//machine_name][:port]/path_to_data" and thus, contains at least four fields. Some fields are optionnal, depending on the requested protocol (*e.g.,* no hostname for "file"). Several full examples of utilization can be found in Section A.
  - char * protocol: a token like "ibp", "http", "ftp", "file", "memory", "LFS" (local file system), "DFS" (distributed file system) or "middleware" can be used. It gives some information on how to access the data (the list is not exhaustive). express the idea
  - 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.

  For example,

  - A GridRPC data corresponding to an input matrix stored in memory can partly be constructed with the information of `protocol` set to `"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 (a key that lets the GridRPC API make the correspondence with the correct input to give to the data middleware section 4.2.2).
  - 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`
- The *management mode* is an enumerated type `grpc_data_mode_t`. It is useful to set the behavior (see Table 1) of the data on the platform. It is given by the client, and 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 user explicitly manages the grpc_data_t data (location and contained data). Thus the data may not be kept inside the platform after a computation, and this can be considered as the default usage for GridRPC API. For coherency issues, note that an underlying data middleware can only consider using any internal copy if it verifies, with the help of checksum functions for example, that the data is indeed identical as the one specified in the URI provided by the user.
  - GRPC_STRICTLY_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_STICKY: used when a data is kept inside the platform but cannot be moved between the servers, except if the user explicitly ask for the migration. 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 also be replicated and that potential coherency issues may arise.
- GRPC_UNIQUE_STICKY: used when a data is kept inside the platform and cannot be moved between the servers. 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_PERSISTENT: used when a data has to be kept inside the platform. The underlying data middleware is explicitly asked to handle the data: the data can migrate or be replicated between servers depending on scheduling decisions, and potential coherency issues may arise if the user attempt to modify the data on his own.
- GRPC_END_LIST: this is not a type, but a marker, that is used to terminate grpc_data_mode_t lists.

grpc_data_mode_t	Keeping inside the platform	Replication	Migration
GRPC_VOLATILE	both possible	not applicable	not applicable
GRPC_STRICTLY_VOLATILE	NO	not applicable	not applicable
GRPC_PERSISTENT	YES	YES	YES
GRPC_STICKY	YES	YES	NO
GRPC_UNIQUE_STICKY	YES	NO	NO

Table 1: Summary of data management modes capabilities

- The *type* of the data is an enumerated type `grpc_data_type_t`: it is set by the client and describes the type of the data, for example GRPC_DOUBLE, GRPC_INT, as exposed in Table 2.

  We have defined a special `grpc_data_t` that can contain other `grpc_data_t`, namely the `GRPC_CONTAINER_OF_GRPC_DATA` (see section 4.2.2 on the management of containers). That way, the user relies on the GridRPC Data Middleware to transfer a set of `grpc_data_t` data. The matter of implementing it by an array, a list or anything else is GridRPC Data Middleware dependent, then not in the scope of this document.

grpc_data_type_t	Definition
GRPC_BOOL	boolean
GRPC_INT	integer
GRPC_DOUBLE	double
GRPC_COMPLEX	complex
GRPC_STRING	string
GRPC_FILE	file
GRPC_CONTAINER_OF_GRPC_DATA	container of grpc_data_t

Table 2: Definition of grpc_data_type_t codes

4.1.2 Function specific types

In this section, we describe some types that are internal to 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, only used in `grpc_data_getinfo()`, is used to define the wanted information. It is an enumerated type defined with the following values (which can be extended):

- GRPC_DATA_HANDLE
  used to have information on the handle.
- GRPC_INPUT_URI
  used to know the different protocols and locations where replica of the data can be accessed.
- GRPC_OUTPUT_URI
  used to know the different protocols and locations where the data has to be transfered.
- GRPC_MANAGEMENT_MODE
  used to know the management mode of a data, for example `GRPC_VOLATILE`.
- GRPC_DIMENSION
  used to know the dimensions of the data.
- GRPC_TYPE
  used to know to which type of the language of implementation the data corresponds.
- GRPC_LOCATIONS_LIST
  used to get all the locations known by the underlying data middleware where to access the data and the protocols used to do so.
- GRPC_STATUS
  used to know if a grpc data is "GRPC_IN_PLACE", "GRPC_TRANSFERING" or "GRPC_ERROR_TRANSFER" for example (exact outputs have to be discussed in an interoperable document).

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, and the syntax of outputs will be discussed in an interoperable document

4.2 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 locations, dimensions, etc. The **init** function sets the data handle to the data it identifies, while the user provides needed information concerning locations 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 the client and explicit locations (computational servers or storage servers) are done using the *asynchronous* **transfer** function. Consequently, the GridRPC data can also be **inspected**, 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.

4.2.1 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()`.

Function prototype:

```
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 data_type,
                            const size_t * data_dimensions,
                            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 required locations on where to transfer the data to. Hence, a list describes generally 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 if the handle has already been defined during a previous call). In sake of simplicity, one can imagine that the default behavior would be a sequential try until the transfer from one of them can be achieved.

Remarks:

- 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.
- Parameter `list_of_URI_input` and `list_of_URI_output` can be set to NULL if empty. Typically, input parameters will have their `list_of_URI_output` set to NULL, output parameter will have their `list_of_URI_input` set to NULL, and inout parameters will have their `list_of_URI_output` containing the `list_of_URI_input`.

  Note that the presence of `list_of_URI_input` and `list_of_URI_output` does not imply that the data is IN, INOUT or OUT. Of course, giving the same value to both list should give the same behavior than setting a data INOUT.
- If `list_of_data_mode` is NULL, the data is managed either with `GRPC_VOLATILE` or `GRPC_STRICTLY_VOLATILE` as default mode depending on the capability of the Grid middleware mode.

  If the data has to be managed differently on at least one another resource, for example with a `GRPC_STICKY` mode on a given resource, then the storage management has to be defined for all locations: this implies that the size of this list is the same as the size of the output list, since there is one mode for each output location.
- The `data_dimensions` parameter is a vector terminated by a zero value, containing the dimensions of the data. For example, an array of $[n\ m\ 0]$ would be used to describe a $n \times m$ matrix.

  The dimension is always known by the client: in case the number of results is not known for a service, then generally this service will return a GRPC_CONTAINER_OF_GRPC_DATA of dimension $[1\ 0]$. Each result can then be accessed inside the single container with `grpc_data_container_get()`, and its dimension known by a call to `grpc_data_getinfo()`.
- Error code identifiers and meanings are described in Table 3. When an argument is not valid, it can mean that either the user made a mistake when calling the function, either that the corresponding information is not supported by the implementation of the API.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_NOT_INITIALIZED	`grpc_initialize()` was not called in advance
GRPC_NOT_IMPLEMENTED	The function is not implemented
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_INVALID_TYPE	Specified type is not valid
GRPC_INVALID_MODE	Specified mode is not valid
GRPC_INVALID_URI	One of the URI is not valid
GRPC_OTHER_ERROR	Internal error detected

Table 3: Error codes identifiers and meanings for the init function.

4.2.2 Functions specific to a special mode or data

Mappings of memory location to given names.

If he wants to use a data stored in memory, the user must provide some name in the URIs in the input and/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:

```
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.

Error code identifiers and meanings are described in Table 4. Note that, like all function of this API, $grpc_initialize()$ as to be called previously of the use of these two functions.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_NOT_INITIALIZED	`grpc_initialize()` was not called in advance
GRPC_NOT_IMPLEMENTED	The function is not implemented
GRPC_OTHER_ERROR	Internal error detected

Table 4: Error codes identifiers and meanings for the memory mapping function.

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_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.

Error code identifiers and meanings are described in Table 5.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_NOT_INITIALIZED	`grpc_initialize()` was not called in advance
GRPC_NOT_IMPLEMENTED	The function is not implemented
GRPC_INVALID_DATA_HANDLE	Specified handle is invalid
GRPC_INVALID_TYPE	Specified type is not valid
GRPC_INVALID_RANK	Specified rank is not valid
GRPC_OTHER_ERROR	Internal error detected

Table 5: Error codes identifiers and meanings for container management functions.

4.2.3 The transfer function

This function writes a GridRPC data to the output locations set during the init call in the output parameters fields. For commodity reasons, additional URIs from which the data can be downloaded and to 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. Then the management of data must use **asynchronous mechanisms** as default behavior. 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_output_modes);
```

`list_of_output_modes` is a `GRPC_END_LIST`-terminated list with the same number of items as `list_of_output_URI`. For each URI describing the hostname, the protocol used to access the data, etc., a management mode can be specified. Hence, `list_of_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 resources and `GRPC_PERSISTENT` to the others) while possibly benefiting of an "aggressive" write as the data is the same everywhere.

Remarks:

- If `list_of_output_modes` is set to NULL, the management mode of the data is the one specified during the initialization of the data, or defaulted if it was not set as a unique mode.

  Note that if a user wants to change the management modes of a data, this function can be called with the fields `list_of_output` and `list_of_output_modes` correctly filled.

- 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 more locations, he can use the `grpc_data_getinfo()` function.
- If a user wants to wait after 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.

Error code identifiers and meanings are described in Table 6.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_NOT_INITIALIZED	`grpc_initialize()` was not called in advance
GRPC_NOT_IMPLEMENTED	The function is not implemented
GRPC_INVALID_DATA_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is not valid
GRPC_INVALID_MODE	Specified mode is not valid
GRPC_INVALID_URI	One of the URI is not valid
GRPC_OTHER_ERROR	Internal error detected

Table 6: Error codes identifiers and meanings for the transfer function.

4.2.4 Waiting for completion of transfers

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
                            grpc_data_t ** returned_data);
```

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.

Remarks:

- `returned_data` is the (or one of the) data which makes the function return: if no special error is returned and the mode was set to `GRPC_WAIT_ANY`, then the data is the one whose transfer has been completed. If an error occured, then there has been at least one error on the transfer of this data.
- 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 for example.
- 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 yet been completed, the behavior is depending on the data middleware that manages the data: if the middleware implements some stamps mechanisms, then no problem will occur.

Error code identifiers and meanings are described in Table 7.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_NOT_INITIALIZED	`grpc_initialize()` was not called in advance
GRPC_NOT_IMPLEMENTED	The function is not implemented
GRPC_INVALID_DATA_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is not valid
GRPC_TRANSFER_ERROR	At least one server is not responding
GRPC_OTHER_ERROR	Internal error detected

Table 7: Error codes identifiers and meanings for the wait function.

4.2.5 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:

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

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

Error code identifiers and meanings are described in Table 8.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_NOT_INITIALIZED	`grpc_initialize()` was not called in advance
GRPC_NOT_IMPLEMENTED	The function is not implemented
GRPC_INVALID_DATA_HANDLE	Specified handle is invalid
GRPC_OTHER_ERROR	Internal error detected

Table 8: Error codes identifiers and meanings for the unbind function.

4.2.6 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, 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.

Error code identifiers and meanings are described in Table 9.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_NOT_INITIALIZED	`grpc_initialize()` was not called in advance
GRPC_NOT_IMPLEMENTED	The function is not implemented
GRPC_INVALID_DATA_HANDLE	Specified handle is invalid
GRPC_INVALID_URI	One of the URI is not valid
GRPC_OTHER_ERROR	Internal error detected

Table 9: Error codes identifiers and meanings for the free function.

4.2.7 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(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 (defined page 8). By setting `URI`, a server name can be given to get some data information dependent on the location of where is the data (like `GRPC_STICKY`). `info` is a `NULL`-terminated list containing the different available information corresponding to the request.

Remarks:

- The exact syntax of outputs of this function has to be defined in an interoperable document.
- For values of `info_tag` 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`).
- `URI` 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` may be "GRPC_IN_PLACE", "GRPC_TRANSFERING" or "GRPC_ERROR_TRANSFER" for example (see the interoperable document to know standard outputs).

Note that in case of `info_tag` is set to `GRPC_DATA_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. Furthermore, the user has to free the memory allocated by the function to `info_tag`.

Error code identifiers and meanings are described in Table 10.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_NOT_INITIALIZED	`grpc_initialize()` was not called in advance
GRPC_NOT_IMPLEMENTED	The function is not implemented
GRPC_INVALID_DATA_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is invalid
GRPC_UNDEFINED	The information is undefined
GRPC_OTHER_ERROR	Internal error detected

Table 10: Error codes identifiers and meanings for the getinfo function.

4.2.8 Functions to load and save data

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, 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 information (dimension, type, etc.), only data information have to be saved to the `URI_output`. The format used by these functions is let to the developer's choice. The way the information are shared by different middleware is out of scope of this document and should be discussed in an interoperability recommendation document.

Error code identifiers and meanings are described in Table 11.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_NOT_INITIALIZED	`grpc_initialize()` was not called in advance
GRPC_NOT_IMPLEMENTED	The function is not implemented
GRPC_INVALID_DATA_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is invalid
GRPC_INVALID_URI	One of the URI is not valid
GRPC_OTHER_ERROR	Internal error detected

Table 11: Error codes identifiers and meanings for the load and save functions.

5 Contributors

Yves Caniou (Corresponding Author)
University of Lyon / CNRS / ENS Lyon / INRIA / UCBL
46 Allée d'Italie
69364 Lyon Cedex 7
France
Email: Yves.Caniou@ens-lyon.fr

Eddy Caron (Corresponding Author)
University of Lyon / CNRS / ENS Lyon / INRIA / UCBL / SysFera
46 Allée d'Italie
69364 Lyon Cedex 7
France
Email: Eddy.Caron@ens-lyon.fr

Frederic Desprez
University of Lyon / CNRS / ENS Lyon / INRIA / UCBL / SysFera
46 Allée d'Italie
69364 Lyon Cedex 7
France
Email: Frederic.Desprez@inria.fr

Gaël Le Mahec
University of Picardie Jules Verne
33, rue Saint Leu
80039 Amiens Cedex 01
France
Email: gael.le.mahec@u-picardie.fr

Hidemoto Nakada (Corresponding Author)
National Institute of Advanced Industrial Science and Technology
Room 1103
1-18-13 Sotokanda
ZIP 1010021
Chiyoda-ku, Tokyo
Japan
Email: hide-nakada@aist.go.jp

Yusuke Tanimura
Information Technology Research Institute, AIST
1-1-1 Umezono, Tsukuba Central 2
Tsukuba City 305-8568
Japan
Email: yusuke.tanimura@aist.go.jp

6 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.

7 Disclaimer

This document and the information contained herein is provided on an "As Is" basis and the OGF 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.

8 Full Copyright Notice

9 References

Appendix

A Examples of use of the API

In this section, we give examples of the data management API usage to illustrate its interest. We do not consider these examples as an exhaustive list but they can help to understand how to use the API, as well as the way to build a data management API in GridRPC middleware.

A.1 Basic example

In this example (see Figure 1), 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. Here, we perform $C = A * B$ on server *karadoc*.

```
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},
         NULL,
         GRPC_DOUBLE, (size_t []){3, 3, 0},
        (grpc_data_mode_t []) NULL);
grpc_data_init(&dhB,
         (const char * []){"DFS://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 * []){"FTP://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.1.1 Input data

Here, we illustrate the way to send 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. As `A` is in memory, a call to `grpc_data_memory_mapping_set()` has to be performed, hence allowing the use of the mapped string to reference `A` in the URI. `B` is a local file, and the protocol is set to DFS (Distributed File System) to express the will of the user to benefit from a distributed file system on the remote server if one is available. This can help to distribute the data, avoiding copies, if the computing resource is a cluster and the service will use some of them.

Note that the memory and DFS protocols are not network protocols: involved transfers will be performed using the GridRPC communication layer, and provided protocols are intended to give information to help the management of the data. Moreover, note that in a real code, `A` must be allocated and defined before any use.

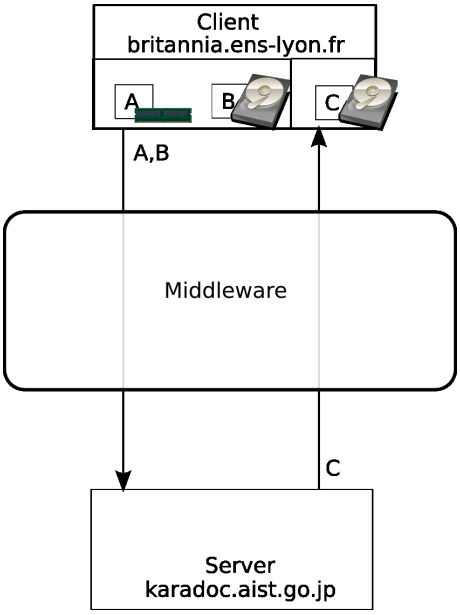

Figure 1: Simple RPC call with input and output data.

A.1.2 Output data

In this example, no data persistency is needed: the client issues a call with `A` and `B` as input data and `C` as output data, and output data `C` is sent back to the client at the end of the computation. But only the transfer is initiated, and a call to `grpc_data_wait()` would be mandatory on the client side to be sure that the data is in place before using it.

A.1.3 Note

In the first call to `grpc_data_init()`, we used a `NULL` value for the list of `grpc_data_mode_t`, which means the default value of the underlying data middleware (either `GRPC_VOLATILE` or `GRPC_STRICTLY_VOLATILE`). Thus, it may be the same than using a list containing `GRPC_VOLATILE` like done in the second and third call.

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

```
grpc_function_handle_init(handle1,"karadoc.aist.go.jp","*");
grpc_data_init(&dhA,
               (const char * []){"IBP://kaamelott.cs.utk.edu/1212#A.dat/ReadKey/READ", NULL},
               (const char * []){"FTP://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 * []){"FTP://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);
```

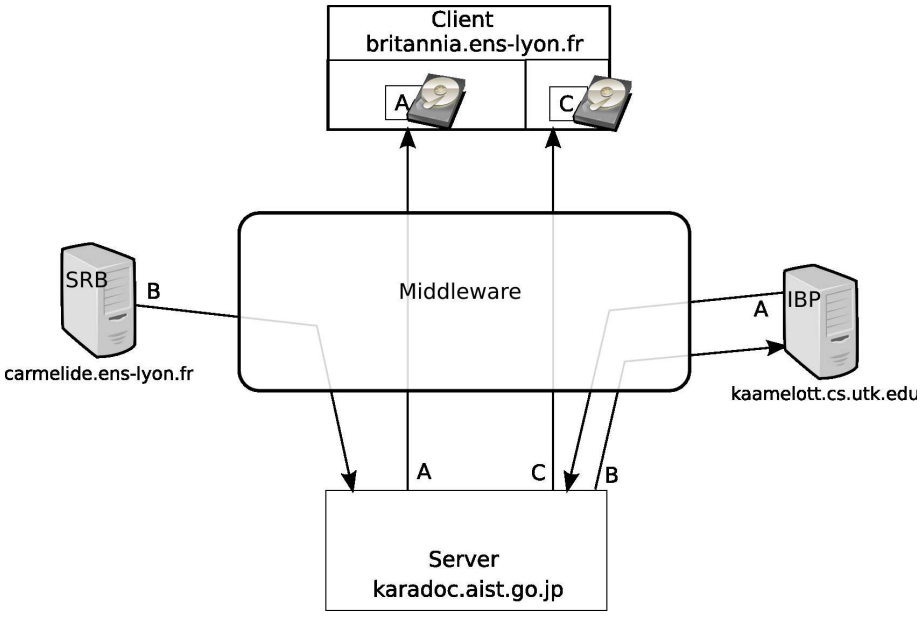

Figure 2: Simple RPC call with input and output data using external storage resources.

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 IBP repository and `B` is available on SRB repository. With the input and output parameters from the `grpc_data_init()` function we can copy the data from a repository to another one:

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

Note that in Figure 2, we arbitrarily show that the data `A` is transfered from the server `karadoc.aist.go.jp` to the client. But it could have been from kaamelott.cs.utk.edu depending on the implementation. The same comment arises for `B`: it could have been transfered from the server `carmelide.ens-lyon.fr`. Moreover, as `A` and `B` are input data, transfers can take place anytime before, during or after the computation. A call to `grpc_data_wait()` is needed to be sure that data `A` is completely stored on the client before using it locally.

A.2.2 Output data

The output data `C` is sent back to the client after the end of the computation. Only the transfer is initiated, and a call to `grpc_data_wait()` is needed to be sure that the transfer of data `C` is completed onto the client before using it locally.

A.3 Example with persistence

We show here how to re-use data on a specific server without resending them, *i.e.,* some data need to be stored inside the platform. In this example (see Figure 3), the client wants to compute $C = C \times A^{n+1}$ using the service `"*"` on server *karadoc*. Temporary data need to be kept on the same server to avoid useless transfers. The `GRPC_STICKY` mode provides this behavior.

```
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/A", NULL},
               GRPC_DOUBLE, (size_t []){3, 3, 0},
               (grpc_data_mode_t []){GRPC_STICKY, GRPC_END_LIST});

grpc_data_init(&dhC,
               (const char * []){"FTP://britannia.ens-lyon.fr/home/user/C.in", NULL},
               (const char * []){"memory://karadoc.aist.go.jp/C", 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(&dhA,
                     (const char * []){"memory://karadoc.aist.go.jp/A", NULL}, NULL,
                     GRPC_DOUBLE, (size_t []){3, 3, 0},
                     (grpc_data_mode_t []) NULL);
      grpc_data_init(&dhC,
                     (const char * []){"memory://karadoc.aist.go.jp/C", NULL}, NULL,
                     GRPC_DOUBLE, (size_t []){3, 3, 0},
                     (grpc_data_mode_t []) NULL);
   }
   if( i==n )
      grpc_data_init(&dhC,
                     (const char * []){"memory://karadoc.aist.go.jp/C", NULL},
                     (const char * []){"FTP://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, dhC, &dhC);
}
grpc_data_free(&dhA);
grpc_data_free(&dhC);
```

A.3.1 Input data

Data `A` will be used and will remain on server *karadoc*: we use the `GRPC_STICKY` parameter to keep the data on server *karadoc*. Data `C` is an input/output data. The first call to `grpc_data_init()` for this data requires only an input location and the `GRPC_STICKY` mode. In this example, we assume that the middleware deals with the readers-writers problem using a reader/writers lock for the access to the data. Then, the second call can only start when `A` is transfered from *britannia*.
If the middleware cannot deal with such an access, users should use `grpc_wait()` after each `grpc_call()`.

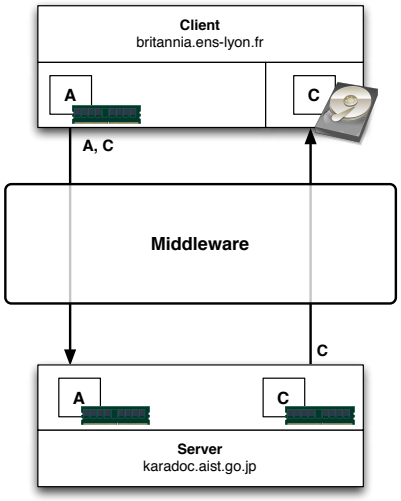

Figure 3: GridRPC calls with data management using persistence through the `GRPC_STICKY` mode.

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 `grpc_call()`. We again assume that the middleware allows only one process/thread to access `C` for writing at a time.

A.4 Example with data migration

In this example (see Figure 4), we show how to use the GridRPC data management functions to "manually" use, re-use and migrate data inside the platform. In this example we consider that the persistent 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 $C = A \times C$.

```
grpc_function_handle_init(handle1,"karadoc.aist.go.jp","*");
grpc_function_handle_init(handle2,"karadoc.aist.go.jp","+");
grpc_function_handle_init(handle3,"perceval.rush.aero.org","*");
grpc_data_init(&dhA,
               (const char * []){"memory://britannia.ens-lyon.fr/A", NULL},
               (const char * []){"memory://perceval.rush.aero.org/A", NULL},
               GRPC_DOUBLE, (size_t []){3, 3, 0},
               (grpc_data_mode_t []){GRPC_STICKY, GRPC_END_LIST});

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

grpc_data_init(&dhC,
                NULL,
                (const char * []){"memory://karadoc.aist.go.jp/C", NULL},
                GRPC_DOUBLE, (size_t []){3, 3, 0},
                (grpc_data_mode_t []){GRPC_STICKY, GRPC_END_LIST});

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

grpc_data_init(&dhB,
               (const char * []){"memory://karadoc.aist.go.jp/B", NULL},
               NULL,
               GRPC_DOUBLE, (size_t []){3, 3, 0},
               (grpc_data_mode_t []) NULL);

grpc_data_init(&dhC,
                (const char * []){"memory://karadoc.aist.go.jp/C", NULL},
                (const char * []){"memory://perceval.rush.aero.org/C", NULL},
                GRPC_DOUBLE, (size_t []){3, 3, 0},
                (grpc_data_mode_t []){GRPC_STICKY, GRPC_END_LIST});

grpc_call(handle2, dhB, dhC, &dhC);

grpc_data_init(&dhA,
               (const char * []){"memory://perceval.rush.aero.org/A", NULL},
               NULL,
               GRPC_DOUBLE, (size_t []){3, 3, 0},
               (grpc_data_mode_t []) NULL);

grpc_data_init(&dhC,
                (const char * []){"memory://perceval.rush.aero.org/C", NULL},
                (const char * []){"FTP://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(handle3, dhA, dhC, &dhC);
```

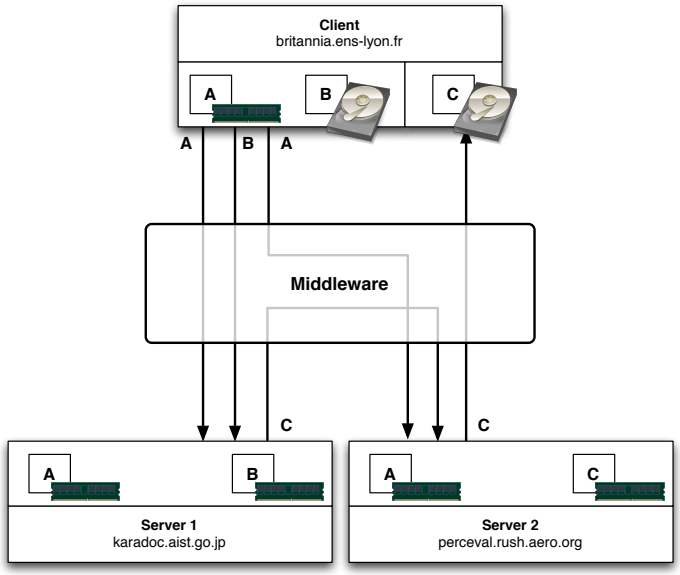

Figure 4: Three RPC calls with data migration.

A.4.1 Input data

Data `A` will be used on *karadoc* and *perceval*, we use a GRPC_STICKY persistence to reuse it with the third `grpc_call()`. The data `B` is only used on *karadoc*. The first `grpc_call()` transfers the data on `karadoc` to compute $C = A \times B$ and on `perceval` for the computation of $C = A \times C$. The second `grpc_call()` computes $C = C + B$ and transfers the data `C` from *karadoc* to *perceval* for the final computation.

A.4.2 Output data

Output data `C` is created on server *karadoc*. `C` moves (or is duplicated) from server *karadoc* to server *perceval*. Then, `C` is sent back to the client.

A.5 Example with containers

In this example, we want to use a GridRPC service which performs $B = \sum_{a_i \in \mathcal{A}} a_i$. Since the service does not know how much elements $\mathcal{A}$ contains before the execution, it is impossible to use a traditionnal type to store the input data. Thus, the containers introduced in the GridRPC API allow a GridRPC platform to propose such a service with a dynamic number of parameters.

```
grpc_data_memory_mapping_set("A", contA);
// "contA" is pointer to a data container type.
// The container type depends on the implementation.

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

grpc_data_init(&dhA,
               (const char * []){"memory://britannia.ens-lyon.fr/A", NULL},
               NULL, GRPC_CONTAINER_OF_GRPC_DATA, (size_t []){1, 0},
               (grpc_data_mode_t []) NULL);
for (i=0; i<10; ++i) {
  char name[34];

  ssprintf(name,"memory://britannia.ens-lyon.fr/a%d", i);
  data_memory_mapping_set(name, &values[i]);

  grpc_data_init(&dha[i], (const char * []){name, NULL},
                 NULL, GRPC_DOUBLE, (size_t []) {1, 0},
                 (grpc_data_mode_t []) NULL);
  grpc_data_container_set(&dhA, i, dha[i]);
}

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

gprc_call(handle1, dhA, &dhB);
```

A.5.1 Input data

We define `A` as a data container. Each element `a` of the set `A` is mapped in memory and added to the data container. The container is then used as input parameter of the `sum` service, exactly as a data of any other type.

A.5.2 Output data

The output is simply computed on *karadoc* and sent back to *britannia*. Here, the user has chosen to store the output data on memory. He asks the middleware to create a new data memory mapping using the name `B` on *britannia*.

A.6 Example containing the use of an underlying data middleware

In this example we show how an advance data middleware can be used through the API. The main differences are that the URIs used to initialized the data are not resource locators but resource identifiers and the data middleware manages data migration, data replication and data persistency.

We want to compute $B = A^n$ on the *karadoc* server with the matrix multiplication service available here. After the computation, we use `grpc_data_transfer()` to transfer the result on a ftp server running on the client (*britannia.ens-lyon.fr*).

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

grpc_data_init(&dhA,
               (const char * []) {"dagda://id-01234567-89ab-cdef-0123456789ab", NULL},
               NULL, GRPC_DOUBLE, (size_t []) {100, 100, 0},
               (grpc_data_mode_t []) {GRPC_PERSISTENT, GRPC_END_LIST});

grpc_data_init(&dhB,
               (const char * []) {"ftp://karadoc.aist.go.jp/pub/mat100_identity.dat", NULL},
               (const char * []) {"dagda://id-98765432-10fe-dcba-9876543210fe", NULL},
               GRPC_DOUBLE, (size_t []) {100, 100, 0},
               (grpc_data_mode_t []) {GRPC_PERSISTENT, GRPC_END_LIST});

grpc_call(handle1, dhA, &dhB);

grpc_data_init(&dhB,
               (const char * []) {"dagda://id-98765432-10fe-dcba-9876543210fe", NULL},
               (const char * []) {"dagda://id-98765432-10fe-dcba-9876543210fe", NULL},
               GRPC_DOUBLE, (size_t []) {100, 100, 0},
               (grpc_data_mode_t []) {GRPC_PERSISTENT, GRPC_END_LIST});

for (i=1; i<n; ++i) {
  grpc_call(handle1, dhA, &dhB);
}

grpc_data_transfer(&dhB, NULL,
                   (const char * []) {"ftp://britannia.ens-lyon.fr/pub/results/B.out", NULL},
                   NULL);
```

A.6.1 Input data

The data is designated by its unique identifier in the data middleware (here the data middleware is DAGDA, the DIET data manager). Then the data is located "somewhere" on the grid. Because we choose, `GRPC_PERSISTENT` as data persistency, the data will stay on the server after the first computation (DAGDA manages this kind of data persistency). Matrix B is initiliazed with the identity matrix and then stores the intermediate results.

A.6.2 Output data

The output data is designated to be a DAGDA data with `GRPC_PERSISTENT` persistency. Then, after each computation, data B stays on *karadoc* for the next loop step. After the n^{th} computation, we get back the result using the extra destination field of the `grpc_data_transfer` function.

We can see that using an underlying data middleware greatly simplifies the application.

B Table of functions

Category	Function Name	Section
lifecycle	`grpc_data_init`	4.2.1
	`grpc_data_unbind`	4.2.5
	`grpc_data_free`	4.2.6
mapping	`grpc_data_memory_mapping_set`	4.2.2
	`grpc_data_memory_mapping_get`	4.2.2
container	`grpc_data_container_set`	4.2.2
	`grpc_data_container_get`	4.2.2
read/write	`grpc_data_transfer`	4.2.3
load/save	`grpc_data_load`	4.2.8
	`grpc_data_save`	4.2.8
wait	`grpc_data_wait`	4.2.4
reflection	`grpc_data_getinfo`	4.2.7

Table 12: Functions defined in this document.

C Table of types

Category	Type Name	Possible values	Section
data structure	grpc_data_t	*structured type*	4.1
	grpc_data_type_t	GRPC_BOOL	4.1
		GRPC_INT	
		GRPC_DOUBLE	
		GRPC_COMPLEX	
		GRPC_STRING	
		GRPC_FILE	
		GRPC_CONTAINER_OF_GRPC_DATA	
data management	grpc_completion_mode_t	GRPC_WAIT_ALL	4.1.2
		GRPC_WAIT_ANY	
	grpc_data_mode_t	GRPC_VOLATILE	4.1
		GRPC_STRICTLY_VOLATILE	
		GRPC_PERSISTENT	
		GRPC_STICKY	
		GRPC_UNIQUE_STICKY	
		GRPC_END_LIST	
	grpc_data_info_type_t	GRPC_DATA_HANDLE	4.1.2
		GRPC_INPUT_URI	
		GRPC_OUTPUT_URI	
		GRPC_MANAGEMENT_MODE	
		GRPC_DIMENSION	
		GRPC_TYPE	
		GRPC_LOCATIONS_LIST	
		GRPC_STATUS	
error	grpc_error_t	GRPC_NO_ERROR	
		GRPC_INVALID_TYPE	
		GRPC_INVALID_MODE	
		GRPC_INVALID_DATA	
		GRPC_INVALID_DATA_HANDLE	
		GRPC_INVALID_RANK	
		GRPC_INVALID_URI	
		GRPC_UNDEFINED	
		GRPC_NOT_INITIALIZED	
		GRPC_NOT_IMPLEMENTED	
		GRPC_TRANSFER_ERROR	
		GRPC_OTHER_ERROR	

Table 13: Types defined in this document.