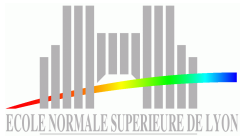
Data Management API within the GridRPC. Grid Final Draf (GFD)

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



Tuesday 22 March 2011

Data Management in the GridRPC

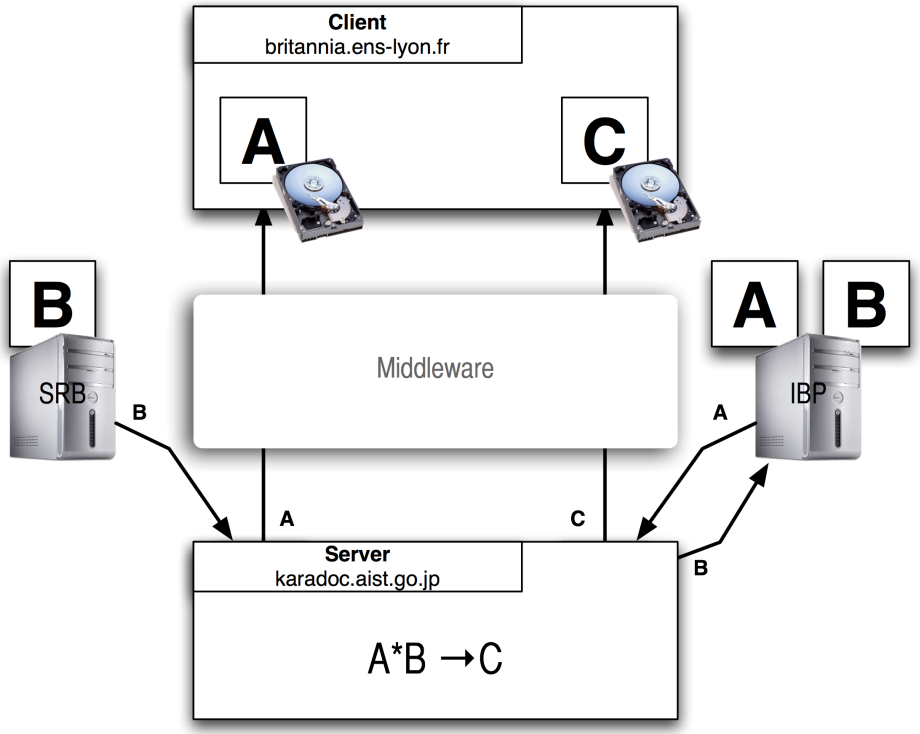
Aims of the Data Management API

- To avoid useless transfers of data
- Generic API unrelated to the data, its location, access protocol, etc.
  $\rightarrow$ Transparent access to the data from the user point of view
- Homogeneous use of different data transfer protocols
- To improve interoperability between different implementations
- The API should be compliant with SAGA API requirements

Constraints

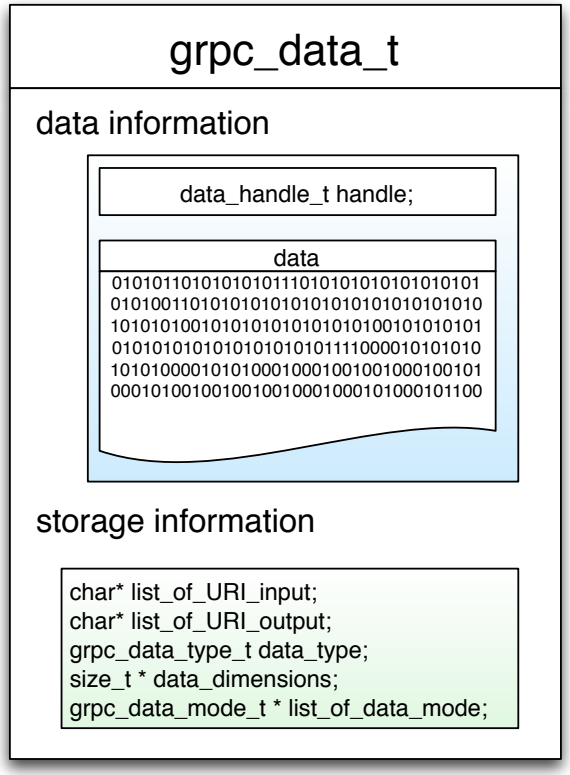
- Must be an optional improvement of GridRPC applications
- Must be in accordance with the GridRPC API
- Should be extensible to existent and future data transfer protocols

Data Management in the GridRPC: Example



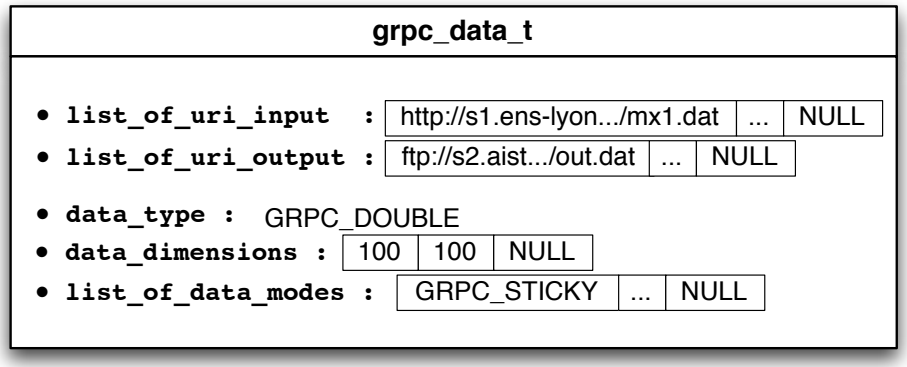
GridRPC Data Type

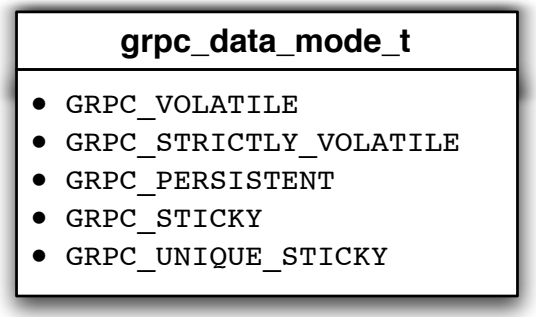
The *grpc_data_t* type contains the data or a handle on it.

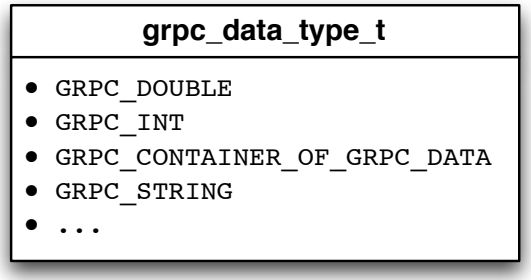


GridRPC Data Example

In this example, the grpc_data_t was initialized to use a matrix 100×100 of doubles located on an http server. The matrix is stored on an external ftp server with the STICKY persistence.







protocol://[user:password@]**hostname**[:port][/data_path]**/data**

Data Management Functions – 1/7

The **grpc_data_init()** Function

```
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_storage_mode);
```

This function initializes the GridRPC data with a specific data.
This data may be available locally or on a remote storage server.
Both identifications can be used.
GridRPC data referencing input parameters must be initialized with identified data before being used in a `grpc_call()`.
GridRPC data referencing output parameters can be initialized to NULL for an empty list.

Data Management Functions – 2/7

The **grpc_data_transfer()** Function

```
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);
```

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

Data Management Functions – 3/7

The **grpc_data_wait()** Function

```
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 transfered, which means that for a given data, all transfers involved for the input *or* output part are finished.

Data Management Functions – 4/7

The **grpc_data_unbind()** Function

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

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

Data Management Functions – 5/7

The **grpc_data_free()** Function

```
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.

Data Management Functions – 6/7

The **grpc_data_getinfo()** Function

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

The kind of information that the function gets is defined by the **info_tag** parameter. A server name can be given to get some data information dependent on the location of where is the data (like `GRPC_STICKY`). **info** is a `NULL`-terminated list containing the different available information corresponding to the request.

Data Management Functions – 7/7

The **grpc_data_load()** and **grpc_data_save()** Functions

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

These functions are used to load/save the data descriptions. Even if the GridRPC data contains the data in addition to metadata management information (data handle, size, type, etc.), only data information have to be saved in the location. 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.

Mappings of memory location to given names

Mapping functions

```
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 );
```

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

A new data type in *grpc_data_t* and new access functions

A new label for the *grpc_data_type_t*

`GRPC_BOOL`, `GRPC_INT`, `GRPC_DOUBLE`, `GRPC_COMPLEX`, `GRPC_STRING`, `GRPC_FILE` and `GRPC_CONTAINER_OF_GRPC_DATA`

Access Functions to Elements in a Container of *grpc_data_t*

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

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

- **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
- **data** is the data that the user wants to add in or get from the container

→ Getting the data does not remove the data from **container**

→ Container management is free of implementation

Conclusion & Future Works

In Brief

- Simple API for data management with only 12 functions
- Allowing a simple and powerful data management from the API
- Taking into account many use cases (all?)
  - Next OGF we plan to show how to use and implement these functions in a couple of examples.
  - send us your case !

Roadmap

- GridRPC data management interoperability
  - New document
  - Interoperability testing for the GridRPC data API specification
- Implementation into GridRPC compliant midlleware