Examples Concerning Containers Within the GridRPC Data Management API

GridRPC working group
Caniou Yves - Yves.Caniou@ens-lyon.fr
Caron Eddy - Eddy.Caron@ens-lyon.fr
Le Mahec Gaël - lemahec@clermont.in2p3.fr

We find that the extension of the GridRPC API with new array functions has some limitations and some drawbacks. As previously said, we firstly consider that there are some overlaps with the OGF GridRPC API standard and that functionalities are/must be included within the Data Management API proposed in the session. Here are other limitations:

- New additional functions to implement.
- New additional functions to test, and more cases of divergence during the interoperability phase.
- Introduction of a new and mandatory way to perform a grpc_call(): If a user wants to perform a call using the possibility to group data inside an array, he must build the array using the new functions and then perform the call with a grpc_call_array() or grpc_call_array_async() function.
- Manage only arrays, not lists: arrays must be of fixed length. If the goal is to ease some kind of data management (get each parameter inside a 'for' for example – which can already be made using argv[i] in C standard –) then lists must be implemented rather than arrays.

To show how simple it is to manage containers of grpc_data_t elements with the proposed GridRPC Data Management API, we join 2 different examples of implementation and one example of utilisation with this document. Please note that **the API does not force any implementation**. We only present examples of implementations here to show how simple and convenient for the programmer to include such functionnalities and the usage example to show how the user can rely on the OGF GridRPC standard and Data Management API.

Of course, what is proposed here is not tested within a real GridRPC implementation and could contain some bugs. The C-style is used to allow a quick and easy test of a such real implementation.

1 Examples of Implementation

1.1 Implementation of the init Function

Before to use a grpc_data, it is necessary to initialize it. The *grpc_data_init()* function makes all the necessary treatments to manage parameters and fills the different fields of the `grpc_data_t` structure. The initialization of a container of `grpc_data_t` data is managed exactly the same way as other types.

```
grpc_error_t grpc_data_init(grpc_data_t * data, char * URI_in, char * URI_out,
                            grpc_data_type_t variable_type,
                            grpc_data_mode_t storage_mode)
{
  /* Initialize the grpc data values, eventually after some treatments */
  data->infos.mode = storage_mode;
  /* ... */

  switch (variable_type) {
    /*
       Specific treatment of each type of data.
   */
  case GRPC_SHORT:
    /* ... */
    break;
  case GRPC_INT:
    /* ... */
    break;
  case GRPC_DOUBLE:
    /* ... */
    break;
  case GRPC_CONTAINER_OF_GRPC_DATA:
    /* ...
       All the data in the list have to be initialized separately before to
       be copied into the array/linked list. Then, the initialization of a
       GRPC_CONTAINER_OF_GRPC_DATA type is the same as for other types. */
    break;
  default:
    /* Return an error code: type is not managed by the API implementation. */
    return GRPC_INVALID_TYPE;
  }
}
```

If the API implementation does not manage containers of GridRPC data, the init function returns GRPC_INVALID_TYPE as defined in the API proposal.

1.2 First Example of Implementation of Containers: With NULL-Terminated Array

This first example presents an implementation of the GridRPC data container as NULL-terminated array (*i.e.,* a pointers array of whom the end is a NULL pointer).

1.2.1 The grpc_data_t Structure

With this implementation, the data structure is the same than the one proposed in the API:

```
typedef struct grpc_data_t {
  void * data;
  grpc_data_handle_t handle;
  grpc_data_storage_info_t infos;
} grpc_data_t;
```

Then, the container of data is simply an array of *grpc_data_t**. The *data* pointer is a pointer on *void* to manage as well simple GRPC data as a container of GRPC data. Moreover, it guarantees the compatibility with implementations which do not manage the *GRPC_CONTAINER_OF_GRPC_DATA*.

1.2.2 Implementation of the *grpc_call* Function

The *grpc_call* function can be implemented as follows:

```
grpc_error_t grpc_call(grpc_function_handle_t * handle, ...)
{
  va_list args;
  va_start(args, handle);
  grpc_data_t * arg;

  while( (arg=va_arg(args, grpc_data_t *))!=NULL ) {
    switch( arg->infos.type ) {
    case GRPC_SHORT:
      /* ... */
      break;
    case GRPC_CONTAINER_OF_GRPC_DATA:
      grpc_data_t * tmp = arg->data;
      while( tmp ) {
/* Manage the data in the array. */
        ++tmp;
      }
      break;
      /* ... */
    }
  }
  /* ... */
}
```

The difference with other implementations is the way we iterate the elements of the array.

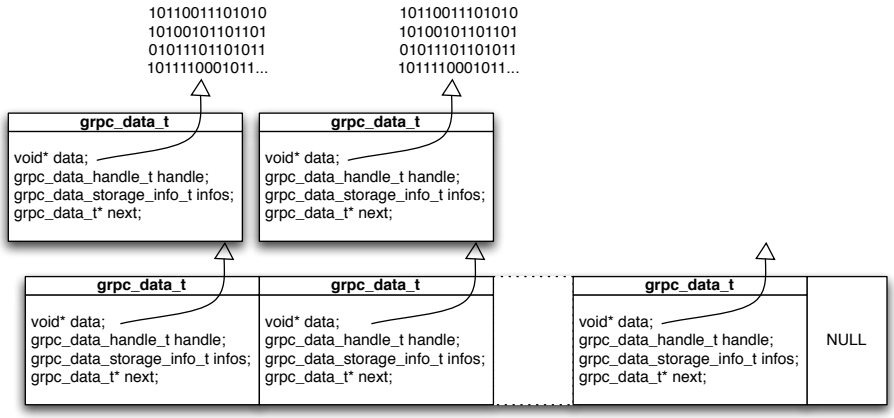
1.2.3 Implementation of Container Management Functions

To use this implementation, the API defines two container management functions: the *grpc_data_container_set(grpc_data_t * container, int index, grpc_data_t * data)* and the *grpc_data_container_get(grpc_data_t * container, int index, grpc_data_t * data)*. For a NULL-terminated array, the functions can be defined as follows:

```
grpc_error_t grpc_data_container_set(grpc_data_t * array, int index,
                                     grpc_data_t * data)
{
  grpc_data_t * it = array;
  int size = 0;

  /* Creation of the array. */
  if( array == NULL ) {
    array = malloc(sizeof(grpc_data_t *));
    array[0]=NULL;
  }

  /* Array size computation. */
  while( it!=NULL ) {
    ++it;
    ++size;
  }

  /* The array need to be extended. */
  if( size<index ) {
    size = index + 2;
    array = realloc(array, size*sizeof(grpc_data_t *));
    array[size-1] = NULL;
  }

  /* Insert the data. */
  array[index] = calloc(1, sizeof(grpc_data_t));
  array[index]->data = data;
  array[index]->infos = calloc(1, sizeof(grpc_data_storage_info_t));
  array[index]->infos.type = GRPC_CONTAINER_OF_GRPC_DATA;
  /* ... */
}

grpc_error_t grpc_data_container_get(grpc_data_t * container, int index,
                                     grpc_data_t * data)
{
  data = container[index]->data;
}
```

Note that here, `index` is the index in the array, but could be in another implementation some key to access the data with, for example, a hash function.

Using this implementation, a grpc_data_t structure can be viewed as follows:



1.3 Second example of implementation of containers: Linked-list implementation

This example presents a way to implement container using a linked-list. This implementation is one of the several possible implementations of such a list.

1.3.1 Data Structure

A possible implementation is to include an additional pointer in the structure itself.

```
typedef struct grpc_data {
  void * data;
  grpc_data_handle_t handle;
  grpc_data_storage_info_t infos;
  struct grpc_data * next;
} grpc_data_t;
```

As for the first implementation example, the *data* pointer points to a grpc_data_t structure when the data type is a *GRPC_CONTAINER_OF_GRPC_DATA*.

1.3.2 Implementation of the *grpc_call* Function

With this data structure, the *grpc_call* function can be defined as follows:

```
grpc_error_t grpc_call(grpc_function_handle_t * handle, ...)
{
  va_list args;
  va_start(args, handle);
  grpc_data_t* arg;

  while ((arg=va_arg(args, grpc_data_t *))!=NULL) {
    switch (arg->infos.type) {
    case GRPC_SHORT:
      /* ... */
      break;
    case GRPC_CONTAINER_OF_GRPC_DATA:
      grpc_data_t * tmp = arg->data;
```

```
      while (tmp) {
        /* Manage the data in the list. */
        tmp=tmp->next;
      }
      break;
      /* ... */
    }
  }
  /* ... */
}
```

The only difference with the previous example is the way we iterate the element of the container.

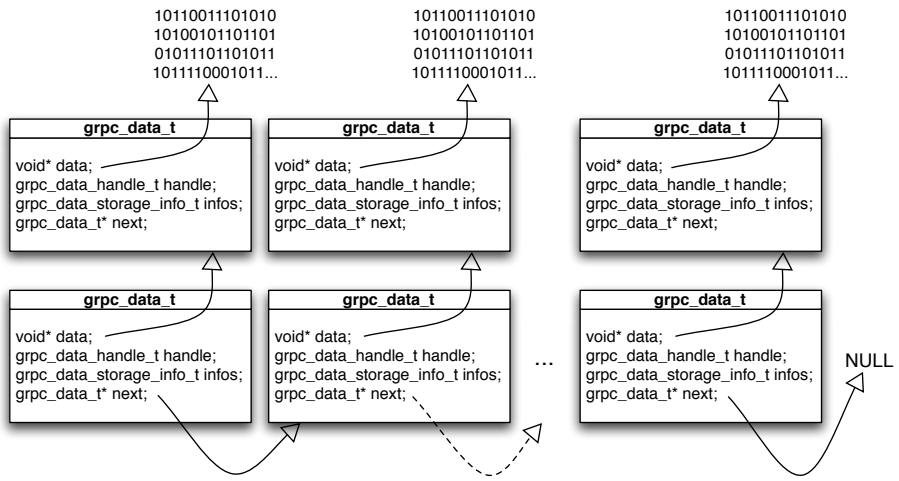
1.3.3 Implementation of Container Management Functions

To use this implementation, the API defines two container management functions: the *grpc_data_container_set(grpc_data_t * container, int index, grpc_data_t * data)* and the *grpc_data_container_get(grpc_data_t * container, int index, grpc_data_t * data)*. For a linked-list, the functions can be defined as follows:

```
grpc_error_t grpc_data_container_set(grpc_data_t * list, int index,
                                     grpc_data_t * elt)
{
  grpc_data_t * it, * prev, * idx;
  int size = 0;

  /* Creation of the list. */
  if( list == NULL ) {
    if (index == 0)
      list = elt;
    else
      list = calloc(1, sizeof(grpc_data_t *));
    list.next = NULL;
  }

  it = list;
  /* List size computation. */
  while( it!=NULL ) {
    prev = it;
    if( size == index )
      idx = it;
    it = it->next;
    ++size;
  }

  /* The list needs to be extended. */
  it = prev;
  if( size<index ) {
    for( int i=size; i<index; i++ ) {
```

```
      it->next = calloc(1, sizeof(grpc_data_t));
      it->next->infos = calloc(1, sizeof(grpc_data_storage_info_t));
      it->next->infos.type = GRPC_CONTAINER_OF_GRPC_DATA;
      /* ... */
      it = it->next;
    }
    it->next->data = elt;
  } else {
    /* We replace the data. */
    idx->data = elt;
  }
}

grpc_error_t * grpc_data_container_get(grpc_data_t * list, int index,
                                       grpc_data_t * data)
{
  grpc_data_t * it = list;
  int idx = 0;

  while( (it != NULL) && (idx < index) ) {
    it = it->next;
    ++idx;
  }
  return it->data;
}
```

Using this data list implementation, a grpc_data_t structure can be viewed as follows:



2 Usage of the GridRPC Data Management API for containers

Whatever the management of the container by the GridRPC Data Management Middleware, the following example is always correct:

```
grpc_data_t pool, d1, d2;
```

```
/* Data initializations */
grpc_data_init(&d1,
               "IBP://kaamelott.cs.utk.edu/1212#A.dat/ReadKey/READ",
               "NFS://britannia.ens-lyon.fr/home/user/A.dat",
               GRPC_DOUBLE, GRPC_VOLATILE);
grpc_data_init(&d2,
               "LOCAL_MEMORY://britannia.ens-lyon.fr/&A",
               "LOCAL_MEMORY://karadoc.aist.go.jp",
               GRPC_DOUBLE, GRPC_STICKY);
grpc_data_init(&pool,NULL,NULL, GRPC_CONTAINER_OF_GRPC_DATA, GRPC_VOLATILE);

/* ... */

grpc_data_container_add(pool, 1, d1);
grpc_data_container_add(pool, 2, d2);
```

3 Conclusion

These examples are possible GRPC data container implementations. Other implementations using hash maps, double linked-list, etc., could be implemented with few effort. Using advanced data container from the standard template library of the C++ or a Java Collection class should even simplify the development of the *grpc_data_container_set* and *grpc_data_container_get* functions.
This implementation allows to use complex combinations of data. For example it is even possible to define a container of container of grpc data and more...