MED Calculator Component

1. Introduction

This example is the following of the HELLO component. It's purpose is to show how to quickly create a new Salome module, to introduce the use of MED objects within Salome/Corba context, to demonstrate the mechanism of exceptions, and to introduce the use of Salome supervisor.

2. The MED libraries

Let's go back a little bit on the different MED libraries (a web site giving general information and documentation about MED project is accessible from the Salome2 web site, links section).

The first level (called “Med File”) is a C and Fortran API that implements mesh and field persistence. The definition of the mesh format (the definition of the implemented entities, the connectivity references ...) is done in [3]. The use of the API is documented by [4] and [5]. The produced files are based on hdf format, and have the “.med” extension.

The second level library is called “Med Memory”, which is a C++ API that allows creating mesh and field objects in memory. Mesh creation can be done from scratch using set functions, or by loading a file with a driver (see html documentation and [6]). Fields are also created using drivers, or by initialization with a user-defined analytic function. Supported file format are .med, .sauv (persistent format of the CASTEM code), vtk (only in write mode), and porflow (fluid flow and heat transfer code). Med Memory was developed to allow the exchange of mesh and field objects in memory between solvers. It can also be used as an internal data structure for new solvers. A last purpose was to gather in the same place existing algorithms and services around meshes and fields :

- computation of sub-elements nodal connectivities (faces, edges),
- geometric computation like volumes, barycenters,
- arithmetic operations on fields, and different kind of norms (integral or discrete, 1 or 2 morms)
- logical operations on field's supports such as union, intersection,
- entity location functionalities, and interpolation toolkit
- building of boundary support of mesh (nodes, elements or faces).

There are two libraries on the third level: a python API generated by SWIG, and a CORBA API :

- The python API wraps the complete Med Memory API, and allows a python user to manipulate Med Memory C++ objects within python.

- The CORBA API was written to facilitate distributed computation inside Salome. The API defines interfaces for the main MED objects (FIELD, SUPPORT, MESH). These interfaces are defined in the file MED.idl, they propose methods that allow distant users to access data. They are implemented by CORBA servants that encapsulate C++ Med Memory objects. This last library is the one we are using here to implement the CALCULATOR component on the server side.

Finally, on the client side, we will demonstrate the use of the MEDClient classes. These classes are proxy classes designed to facilitate and optimize interaction of distant MED CORBA objects with local solvers. The Corba API is completely hidden to clients.

3. Creation of a new Salome module - Compilation

The first step when developing a new Salome module is to create a directories tree with makefiles that allow you to compile a Salome component. This directories tree must follow Salome2 rules, which are described in [1]. Create a complete tree from scratch is complicated, and error prone. The easiest way consist to find an existing module that "looks like the one you need", and copy it. A shell script was written to facilitate this step : `renameSalomeModule`. This utility replace, after copying a module, any occurrence of the old name by the new one, thus avoiding to forget one of them. In the following example, we create a `CALCULATOR_SRC` module by copying `HELLO_SRC` module, then `renameSalomeModule` replace any occurrence of HELLO by CALCULATOR in `CALCULATOR_SRC` module:

```
cp -r HELLO_SRC CALCULATOR_SRC
renameSalomeModule  HELLO CALCULATOR CALCULATOR_SRC
```

The remaining charge for the developer is to define the module interface (by writing a CORBA IDL file), and to implement it. But before, you may want to check that your duplicated module still compiles :

```
CALCULATOR_SRC/build_configure
mkdir CALCULATOR_BUILD
mkdir CALCULATOR_INSTALL
cd CALCULATOR_BUILD
../CALCULATOR_SRC/configure -prefix=installDir
make && make install
```

For more information about makefiles and compilation in Salome environment, please refer to [7].

4. Modification of interface (IDL)

The chosen methods demonstrate the use of MED fields (`FIELDDOUBLE` interface) as in/out parameters and return value.

```
#ifndef __CALCULATOR_GEN__
#define __CALCULATOR_GEN__

#include "SALOME_Component.idl"
#include "SALOME_Exception.idl"
#include "MED.idl"

module CALCULATOR_ORB
{
  /*! \brief Interface of the %CALCULATOR component
   */
  interface CALCULATOR_Gen : Engines::Component
  {
     /*!
         Calculate the maximum relative difference of field
with the previous one.
         At first call, store passed field and return 1.
      */
     double convergenceCriteria(
         in SALOME_MED::FIELDDOUBLE field);
     /*!
         Apply to each (scalar) field component the linear
function x -> ax+b.
         Release field1 after use.
      */
     SALOME_MED::FIELDDOUBLE applyLin(
         in SALOME_MED::FIELDDOUBLE field1,
         in double a1,
         in double a2);

     /*!
         Addition of fields.
         Return exception if fields are not compatible.
         Release field1 and field2 after use.
     */
     SALOME_MED::FIELDDOUBLE add(
         in SALOME_MED::FIELDDOUBLE field1,
         in SALOME_MED::FIELDDOUBLE field2)
         raises (SALOME::SALOME_Exception);

     /*!
         return euclidian norm of field
         Release field after use.
      */
     double norm2(in SALOME_MED::FIELDDOUBLE field);
    /*!
         return L2 norm of field
         Release field after use.
      */
     double normL2(in SALOME_MED::FIELDDOUBLE field);

     /*!
         return L1 norm of field
         Release field after use.
      */
     double normL1(in SALOME_MED::FIELDDOUBLE field);

     /*!
         return max norm of field
         Release field after use.
      */
     double normMax(in SALOME_MED::FIELDDOUBLE field);

     /*!
         This utility method print in standard output the
coordinates & field values
         Release field after use.
      */
     void printField(in SALOME_MED::FIELDDOUBLE field);

     /*!
         This method clones field in four examples.
         Release field after use.
      */
     void cloneField(
         in SALOME_MED::FIELDDOUBLE field,
         out SALOME_MED::FIELDDOUBLE clone1,
         out SALOME_MED::FIELDDOUBLE clone2,
         out SALOME_MED::FIELDDOUBLE clone3,
         out SALOME_MED::FIELDDOUBLE clone4 );
  };
};

#endif
```

The main points to note are:

- the protection against multiple inclusion (ifndef instruction),
- the inclusion of `SALOME_Component.idl` and `SALOME_Exception.idl` files, necessary for each Salome component (the CALCULATOR interface inherit from `Engines::Component` to benefit common services),
- the inclusion of MED.idl, because we are using the `FIELDDOUBLE` interface defined in `SALOME_MED` module.
- The use of "doxygen like" comments, to allow automatic generation of inline documentation.

5. Component implementation

After defining the interface of our component, we have to implement it by modifying the C++ implementation class (`CALCULATOR.hxx` and `CALCULATOR.cxx` in `src/CALCULATOR` directory) and adapt it to the new IDL. In our case, this means to replace the HELLO method "`char* makeBanner(const char* name)`" with new methods that extends the IDL-generated implementation base class (as explained in the HELLO documentation, when compiling the IDL, CORBA generates an abstract base class, that the developer of the component has to derive and write code for the abstract methods). For the CALCULATOR component, the IDL-generated base class is called `POA_CALCULATOR_ORB::CALCULATOR_Gen` and is defined in generated header `CALCULATOR_Gen.hh`.

The IDL attributes are mapped to C++ methods. This operation is normalized by CORBA. Here, we give the mapping for the types involved in our example:

IDL Type	**C++ type**
double	CORBA::DOUBLE
in FIELDDOUBLE	FIELDDOUBLE_ptr
out FIELDDOUBLE	FIELDDOUBLE_out
FIELDDOUBLE	FIELDDOUBLE_ptr

`FIELDDOUBLE_ptr` and `FIELDDOUBLE_out` are C++ classes generated by the IDL compiler to map the MED CORBA interface `FIELDDOUBLE`. We will see below how to create such classes. But before, let's have a look on the new header of the user-defined derived class `CALCULATOR.hxx` :

```
#ifndef _CALCULATOR_HXX_
#define _CALCULATOR_HXX_

#include <SALOMEconfig.h>
```

```
#include CORBA_SERVER_HEADER(CALCULATOR_Gen)
#include CORBA_CLIENT_HEADER(MED)
#include "SALOME_Component_i.hxx"

class CALCULATOR:
  public POA_CALCULATOR_ORB::CALCULATOR_Gen,
  public Engines_Component_i
{

public:
    CALCULATOR(CORBA::ORB_ptr orb,
            PortableServer::POA_ptr poa,
            PortableServer::ObjectId * contId,
            const char *instanceName,
            const char *interfaceName);
    virtual ~CALCULATOR();

    CORBA::Double convergenceCriteria(
         SALOME_MED::FIELDDOUBLE_ptr field);
    CORBA::Double normMax(
         SALOME_MED::FIELDDOUBLE_ptr field1);
    CORBA::Double normL2(
         SALOME_MED::FIELDDOUBLE_ptr field1);
    CORBA::Double norm2(SALOME_MED::FIELDDOUBLE_ptr field1);
    CORBA::Double normL1(
         SALOME_MED::FIELDDOUBLE_ptr field1);
    SALOME_MED::FIELDDOUBLE_ptr applyLin(
         SALOME_MED::FIELDDOUBLE_ptr field1,
         CORBA::Double a,CORBA::Double b);
    SALOME_MED::FIELDDOUBLE_ptr add(
         SALOME_MED::FIELDDOUBLE_ptr field1,
         SALOME_MED::FIELDDOUBLE_ptr field2)
        throw ( SALOME::SALOME_Exception );
    void printField(SALOME_MED::FIELDDOUBLE_ptr field);
    void cloneField(
         SALOME_MED::FIELDDOUBLE_ptr field,
         SALOME_MED::FIELDDOUBLE_out clone1,
         SALOME_MED::FIELDDOUBLE_out clone2,
         SALOME_MED::FIELDDOUBLE_out clone3,
         SALOME_MED::FIELDDOUBLE_out clone4);
};


extern "C"
    PortableServer::ObjectId * CALCULATOREngine_factory(
            CORBA::ORB_ptr orb,
            PortableServer::POA_ptr poa,
            PortableServer::ObjectId * contId,
            const char *instanceName,
            const char *interfaceName);


#endif
```

The main points to note are:

- the inclusion of `CORBA_SERVER_HEADER(CALCULATOR_Gen)` : this macro includes the header of the base class generated by CORBA
- the inclusion of `CORBA_CLIENT_HEADER(MED)` : this macro includes the header we needs to use CORBA MED interfaces (here, to use `FIELDDOUBLE` interface).

The implementation of the methods is very simple, thanks to the use of MEDClient library, which create an automatic link between CORBA and C++ objects. As a first example, let' sconsider the implementation of the `norm2` method. For being more concise, we do not explicit here the namespace `SALOME_MED::` .

```
CORBA::Double CALCULATOR::norm2(FIELDDOUBLE_ptr field1)
{
    beginService( "CALCULATOR::norm2");
    BEGIN_OF("CALCULATOR::Norm2(FIELDDOUBLE_ptr field1)");

    // Create a local field from corba field
    // apply method normMax on it. When exiting the function
    // f1 is deleted, and with it the remote corba field.
    FIELDClient<double> f1(field1);
    CORBA::Double norme = f1.norm2();
    END_OF("CALCULATOR::Norm2(FIELDDOUBLE_ptr field1)");
    endService( "CALCULATOR::norm2");
    return norme;
}
```

The `norm2` method receives as an input parameter a reference to a distant MED CORBA field (named `field1`). It plays the role of the client toward the distant field `field1`. As a client, we could directly call the methods of the `FIELDDOUBLE` CORBA API, for example call the `getValue()` method to retrieve the field values as an array. Doing this has some drawbacks. The transfer is not optimized because values are duplicated on server side. On the client side, we retrieve an array, but if we want to use existing solver or a function that takes an MedMemory C++ field, we need to rebuild a C++ field from the array, which is fastidious. That' swhy we are using here FIELDClient class : `FIELDClient<double>`. This is a proxy C++ template class (also available for int type), that inherit the interface of the MedMemory C++ `FIELD<double>` class. Therefore, it can be used anywhere in place where a `FIELD<double>` is expected. The characteristics of this class are :

- it holds the CORBA reference of the distant field – and release it when object get out of scope (done in the class destructor),
- on creation, only the general information are retrieved from distant field (like size, number of component), not the complete array,
- complete array is transfered only on demand,
- the transfer is optimized : duplication is avoided on server side, and transfer protocol may be switched at compile time (for example to MPI on a parallel machine), without any modification of client code,
- the memory is automatically managed : when deleted, the FIELDClient release the CORBA reference it holds.
- and as already said, it can be used anywhere in state of a FIELD<double>, thus facilitating re-use of existing C++ API.

In our example, we simply create a `FIELDClient`, and then call on it the norm2 method of the MedMemory C++ API :

```
FIELDClient<double> f1(field1);
CORBA::Double norme = f1.norm2();
```

A client class was also created for MESH, called `MESHClient`, with the same characteristics. For meshes, all the arrays (connectivities, coordinates) are transferred on demand, which is generally more interesting than for fields (where we usually need to retrieve values soon or later).

`BEGIN_OF` et `END_OF` macros are used to send traces to standard output when working on debug mode. `BeginService` and `endService` macros are used to send signals to the Supervisor to let him know the state of computation.

As a second example, let consider the applyLin method, which plays both the role of client and server:

```
FIELDDOUBLE_ptr CALCULATOR::applyLin(
          FIELDDOUBLE_ptr field1,
          CORBA::Double a,CORBA::Double b)
{
    beginService( "CALCULATOR::applyLin");
    BEGIN_OF("CALCULATOR::applyLin");
    // create a local field on the heap,
    // because it has to remain after exiting the function
    FIELD<double> * f1 = new FIELDClient<double>(field1);
    f1->applyLin(a,b);

    // create servant from f1, give it the property of c++
    // field (parameter true).  This imply that when the
    // client will release it's field, it will delete
    // NewField,and f1.
    FIELDDOUBLE_i * NewField = new FIELDDOUBLE_i(f1,true) ;
    // activate object
    FIELDDOUBLE_ptr myFieldIOR = NewField->_this() ;

    END_OF("CALCULATOR::applyLin");
    endService( "CALCULATOR::applyLin");
    return myFieldIOR;
```

The method is client for the parameter field `field1`, and server for the returned field `NewField`. The client part (treatment of `field1`) is similar to the first example : we create with `field1` a FIELDClient `f1` and apply on it C++ method applyLin. The difference is that creation is done on the heap, not on the stack (we will explain why later) :

```
FIELDDOUBLE_i * NewField = new FIELDDOUBLE_i(f1,true) ;
f1->applyLin(a,b);
```

For the server part, we create a CORBA field (class `FIELDDOUBLE_i`), activate it and return a reference on it :

```
FIELDDOUBLE_i * NewField = new FIELDDOUBLE_i(f1,true) ;
FIELDDOUBLE_ptr myFieldIOR = NewField->_this() ;
return myFieldIOR;
```

The parameters passed to the `FIELDDOUBLE_i` constructor are the C++ field f1 that is wrapped and used to give the services declared in IDL, and a boolean that indicates if ownership of wrapped field is transferred or not. If ownership is transferred, this means that when the CORBA field will be released by a client (for example by a `FIELDClient` created with a reference on it), it will delete the C++ field it holds. For example, the following code a hypothetic client could write would cause deletion of C++ field `f1` :

```
FIELDDOUBLE_ptr distant_f = CALCULATOR::applyLin(f,a,b);
FIELD<double>* local_f = new FIELDClient<double>(distant_f);
//  .. Use  local_f
delete  local_f; // causes release of distant_f and deletion
                 // of the C++ field it holds
```

This is why `f1` is created on the heap and is not deleted : we want it to survive the end of the method! It will be deleted when client will release it reference.

References

1 Guide pour le développement d' un module Salome 2 en Python (C. Caremoli).

2 Guide pour le développement d' un module Salome 2 en C++ (N. Crouzet).

3 Définition du modèle d' échange de données MED V2.2 (V. Lefebvre, E. Fayolle).

4 Guide de référence de la bibliothèque MED V2.2 (V. Lefebvre, E. Fayolle).

5 Guide d' utilisation de la bibliothèque MED V2.2 (V. Lefebvre, E. Fayolle).

6 User' s guide of Med Memory (P. Goldbronn, E. Fayolle, N. Bouhamou).

7 Using the Salome configuration znd building system environment (P. Goldbronn, M.Tajchman)