Proposal and Prototype for an in-memory representation of ADF trees

/ELSA/NI-02004
Marc Poinot
poinot@onera.fr

These pages are a **proposal** for in-memory representation of ADF trees. The ADF trees are the underlaying representation of CGNS trees, which are more complex structures of data. The CGNS standard actually defines an high level representation of data, but not a low level one. The ADF library has a de-facto data format, which has good qualities of portability and compactness, but has poor results for, at least, data retrieval or garbage. We are describing an in-memory tree, that can be translated into various formats using specific tree parsers. The two main formats which appear to be useful in the CFD applications scope are the proprietary memory format and the XDR memory stream format.

1 Scope of the proposal

This document presents an extension built using the ADF library. This extension manages in-memory ADF trees. As a matter of fact, in-memory representation of ADF trees doesn't exist, and the proposal is more an interface and protocol description than a real memory representation of data.

1.1 The CGNS standard

We are focusing on ADF library/API, but there is no doubt the MLL API will be available with the in-memory tree system. It is matter of replacing ADF calls by ADFM calls (the in-memory ADF-like API). The point is to find a good production system, which insures a complete compatibility with existing MLL/ADF libs. We have to avoid to modify existing libs for people who doesn't want to deal with in-memory stuff. But we should take care of duplicate code between ADF and/or ADFM.

1.2 Sharing trees

So far, the ADF trees can be shared using the disk. This means the interoperable applications have to stand on the same platform, or this requires a system mecanism such as NFS (which does NOT insure portability of data in the files).
We want to share tree using the memory. We want to send a tree for pre or post processing, between interoperable applications running on distant and different hardware platforms. Many mecanisms are insuring data transfert and portability, but we have to deliver to these mecanism a contiguous memory zone representing the whole tree.
The ADFM library provides such an contiguous in-memory tree (more exactly, we are providing a way to get the tree in a contiguous memory zone). The ADFM API is the almost same as ADF API (the *almost* is discussed below).
We also want to avoid to duplicate very large amount of data. There is the memory space problem, but also the translation or transfert time to take into account.

1.3 Deliverables

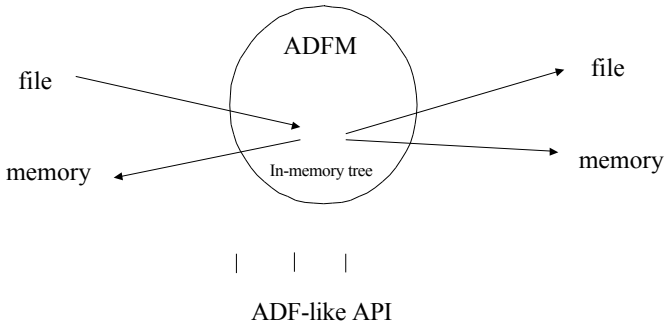
There are source files, examples and documentation. The source files are the ADF-like API (ADFM.h ADFM.c) and the tree management/translation with XDR format (ADFM_tree.h ADFM_tree.c). The examples are described below, there are source file and makefiles for examples using shared memory, RPC and MPI.
The whole set could be delivered as a sub-part of pyCGNS (not that relevant !), or as a separate file tree or an extension to the ADF existing file tree.

2 Architecture & Design

This part has to be detailled, non-maintenability is one of the main causes of software death.

2.1 Using the black box

The ADFM lib creates a tree in the application memory. At open and close time, the lib looks for the ADFM_IO_mode variable which defines the IO mode for opening and/ or closing the file. In the case of a XDR input/output, using the XDR memory stream, it is possible to declare a pointer where the user wants to start the memory zone. The user can then read/write an XDR coded tree. Such a tree can be sent to any platform without loss of portability. The data arrays are stored as *opaque* data, with the IEEE format. Then, on an IEEE machine, there is no translation overhead time.
The data arrays are pointed in the user's application memory zone. At close time, the arrays are copied on disk if the IO mode is FILE. In the case of XDR, the array is copied is the mode is XDR PLAIN, it is not copied if the mode is XDR SHADOW. This latter mode can be used to transfert tree structures, the data itself is copied only if the size does not exceed a value configured at compile time (could be changed to have a API defined value).



2.2 Inside the box

The in-memory tree is a classical node structure, used in C langage based applications. The structure has its own data and a list of sub-nodes, or sons. This classical structure also is used by the ADF library, and the mapping to the in-memory tree is quite easy to achieve.
The open/close calls on the database are triggering translations. One can read from a file at open time, store in contiguous memory zone at close time. Or open/close in memory. All calls to ADFM (e.g. create node, add data, get node label…) are reading or writing the in-memory representation of the node. The actual result is obtained on disk at close time, if the disk is declared as the output.
When the output is an XDR memory stream, the first step is to parse (depth first) the tree and count the size it has using the XDR translation. This size is re-used to allocate an actual bunch of memory before finally writing the tree.

3 Programming interface

This is the important point. We can implement systems and use many external libraries, if the interface/protocol of our system is bad or if the user does NOT agree with it… you loose!

3.1 ADF wrapper and extension

The ADFM API is the same as ADF API. The call

```
ADF_Create(double nodeid, char *name, double newnode, int error)
```

Is then

```
ADFM_Create(double nodeid, char *name, double newnode, int error)
```

Both API could be used at the same time, but the ids are not of the same meaning. The extensions to the API are functions to get/set the IO mode and the pointer to contiguous memory zone:

```
ADFM_Set_IO_Mode(ADFM_IO_MODE_ENUMERATE mode, int error)
ADFM_Set_IO_Ptr(void **ptr, int error)
```

Thus, a simple pre-processing system can swith calls from ADF to ADFM or back.

A open time, the if the IO mode is not ADFM_IO_FILE, the filename is not taken into account. The allowed modes are `ADFM_IO_xxx` with `xxx` in:

NONE	Nothing is used at open time or close time
XDR PLAIN	The XDR memory stream is used, arrays are read/write at open/close time
FILE	The ADF lib is used at open/close time to read/write a tree in memory
XDR SHADOW	Same as XDR above, but arrays are not copied is larger than a fixed size

3.2 API discussion

Now the system is alive (not very robust, but alive!), we have to discuss about changing some API issues.

3.2.1 MLL multi-ADF API

We could add both ADF and ADFM in the MLL. A mode/option setting could handle the call to ADF or ADFM without changing the MLL calls.

3.2.2 Links management

Links are skipped. This means that if we find a link in the path, we are ignoring it and we return an error. We want to avoid to store all the trees in memory. But this could be managed in another way. The ADF lib has a function which insures the retrieval of a real node or a link.

3.2.3 Access to memory zone information

The size of the memory zone has to be handled in a nice way. A good approach would be to have a mini-API for both IO mode, pointer to data and its size. So far, the size is hard coded in examples, and the pointer management often is made with memcpy.

4 Examples

Please note all these examples may not be usable using copy/paste, because some lines has been removed to avoid having too much code in this document. The usable examples actually are in the in-memory delivery, with makefiles. The RPC example also comes with the `rpcgen` build script.
The examples are using system mecanisms, such as shared memory, RPC, MPI, etc… and I certainly do not claim to be a good user of such mecanisms. Then, MPI or RPC experts may find some bad practice in the examples. Theses examples only are proof of usability.

4.1 Fit in-memory before actual store

The simpler way to use the in-memory representation is to set the IO mode to NONE and call the ADF-like functions for tree management. The tree lays in memory, and will be destroyed at close time. See array owner discussion for deletion of arrays in the tree. The user can change the IO mode before the call to the close function. If the FILE IO mode is set, the in-memory tree is saved as an ADF file, exactly as if it would have been by calling the ADF function. Then, the user can built his tree, modify it, change, add move data and actually store the data on disk when the tree has the expected structure.

```
int main(int argc,char **argv)
{

  double root_id,parent_id,child_id,tmp_id;
  int c[6] = {1,2,3,4,5,6};
  int c_dimension = 6;
  int error_flag, i, j;
  int error_state = 1;

  ADFM_Set_Error_State(error_state,&error_flag);
  ADFM_Set_IO_Mode(ADFM_IO_NONE,&error_flag);

  ADFM_Database_Open(file_name,"new"," ",&root_id,&error_flag);
  ADFM_Create(root_id,"n1",&tmp_id,&error_flag);
  ADFM_Create(root_id,"n2",&tmp_id,&error_flag);
  ADFM_Create(root_id,"n3",&tmp_id,&error_flag);
  ADFM_Get_Node_ID(root_id,"n1",&parent_id,&error_flag);
  /* some more calls for tree creation - deleted lines */
  ADFM_Create(child_id,"n13",&tmp_id,&error_flag);
  ADFM_Set_Label(tmp_id,"Label on Node n13",&error_flag);
  ADFM_Put_Dimension_Information(tmp_id,"i4",1,&c_dimension,&error_flag);
  ADFM_Write_All_Data(tmp_id,(char *)(c),&error_flag);

  ADFM_Set_IO_Mode(ADFM_IO_FILE,&error_flag);
  ADFM_Database_Close(root_id,&error_flag);
}
```

4.2 Shared memory

Two applications are sharing the same memory space, using the shared memory IPC (POSIX) mecanisms. The semaphores are used to control access to data. The writer is called the server application, the reader is the client. In this basic example, a simple tree is built in memory, dumped into the shared memory zone and then unlock for read access. Please, again, note this sample is a purged version of the example delivered with the package.
The server opens an ADF file it and stores it as an in-memory XDR representation. This representation is copied into the shared memory segment (one could give the segment adress at first, then the copy is not useful).

```
/* server side */
int main(int argc,char **argv)
{

  unsigned *map1, *map2;
  int fd;
  sem_t *semdes;
  double root_id;
  int error_flag, i, j;
  int error_state = 1;

  fd = shm_open( "/home/poinot/Wksp/shmadf", O_RDWR | O_CREAT, 0777 );
  ftruncate( fd, sizeof( *map2 ) * getpagesize() );
  map2 = mmap( 0, sizeof( *map2 )* getpagesize(),PROT_READ | PROT_WRITE,MAP_SHARED, fd, 0 );
  semdes=sem_open("/home/poinot/Wksp/shmadf.lock", O_CREAT, 0644, 0);
```

```
  ADFM_Set_IO_Mode(ADFM_IO_FILE,&error_flag);
  ADFM_Database_Open("Z.adf","old"," ",&root_id,&error_flag);
  ADFM_Set_IO_Mode(ADFM_IO_XDR_PLAIN,&error_flag);
  ADFM_Database_Close(root_id,&error_flag);
  ADFM_Get_IO_Ptr((void**)(&map1),&error_flag);

  memcpy(map2,map1,1024);
  sem_post(semdes);
  close( fd );
}
```

The client is waiting for the lock, then it reads the XDR tree and prints it.

```
/* client side */
int main(int argc,char **argv)
{
  unsigned *map1, *map2;
  int fd;
  sem_t *semdes;
  double root_id;
  int error_flag;

  fd=-1;
  while (fd == -1)
  {
       fd = shm_open( "/home/poinot/Wksp/shmadf", O_RDWR , 0777 );
  }
  map2 = mmap( 0, sizeof( *map2 )* getpagesize(),PROT_READ | PROT_WRITE,MAP_SHARED, fd, 0 );
  semdes=sem_open("/home/poinot/Wksp/shmadf.lock",0,644,0);
  if (!sem_wait(semdes))
  {
    ADFM_Set_IO_Ptr(map2,&error_flag);
    ADFM_Set_IO_Mode(ADFM_IO_XDR_PLAIN,&error_flag);
    ADFM_Database_Open("NO FILE","old"," ",&root_id,&error_flag);
  }
  sem_post(semdes);
  print_child_list(root_id,0); /* shows data tree */
  ADFM_Set_IO_Mode(ADFM_IO_NONE,&error_flag);
  ADFM_Database_Close(root_id,&error_flag);
  close( fd );
  shm_unlink( "/home/poinot/Wksp/shmadf" );
  return 1;
}
```

4.3 RPC

The RPC example is build using the rpcgen tool. This tool takes a program with functions and splits it with client/server parts depending on the user directives. It is very easy to make a client/server application, with data portability and without all network related issues (at least basic ones).

The example is the same as the shared memory example. The client reads an ADF tree, the server reads it through a TCP/IP connection on a remote machine.

The command line on server side is:

```
Server &
```

The client side command has the remote host name and the client-side ADF file to read:

```
Client hostname adf-file-name
```

The data is transfered as a anonymous memory zone, already encoded using XDR. Note that XDr actually is the encode/decode layer for RPC calls.

The protocol description file is the following:

```
/*
 * adftree.x: Remote message printing protocol
 */
struct adf_tree_s
{
 int    id;
 opaque data<>;
};
typedef adf_tree_s adf_tree;

program ADFTREEMANAGER {
    version ADFTREEMANAGER_1 {
        int SENDTREE(adf_tree) = 1;
    } = 1;
} = 99;
```

The client side is:

```
/* includes deleted here */
void main(argc, argv)
int argc;
char **argv;
{
    CLIENT *cl;
    int  *result,error_flag;
    char *server;
    char *filename;
    adf_tree *tree;
    double root_id;
    void *map1;

    if (argc < 3) {fprintf(stderr, "usage: %s host adf-file-name\n", argv[0]); exit(1);}
    server = argv[1];
    filename = argv[2];

    tree = (adf_tree*) malloc(sizeof(adf_tree));
    tree->data.data_len=1024;
    tree->data.data_val=(void*) malloc(1024);
    strcpy(tree->data.data_val,argv[2]);
    tree->id=1;

    ADFM_Set_IO_Mode(ADFM_IO_FILE,&error_flag);
    ADFM_Database_Open(filename,"old"," ",&root_id,&error_flag);
    ADFM_Set_IO_Mode(ADFM_IO_XDR_PLAIN,&error_flag);
    ADFM_Database_Close(root_id,&error_flag);
    ADFM_Get_IO_Ptr((void**)(&map1),&error_flag);

    memcpy(tree->data.data_val,map1,1024);

    cl = clnt_create(server,ADFTREEMANAGER,ADFTREEMANAGER_1,"tcp");
    if (cl == NULL) {clnt_pcreateerror(server); exit(1);}
    cl->cl_auth = authunix_create_default();
    result = sendtree_1(tree, cl);
    if (result == NULL) {
        clnt_perror(cl, server);
        exit(1);
    }

    if (*result == 0) {fprintf(stderr,"%s: %s tranfert failed\n",argv[0], server); exit(1); }
    printf("Tree sent to %s\n", server);
    exit(0);
}
```

The server side is mostly generated by rpcgen. The function declared in the protocol is in a separate file. A specific server has to be built for each server platform. This is also true for client side. The RPC/XDR system insure the data portability :

```
/* includes deleted here */
int *sendtree_1(tree, UNUSED)
/* UNUSED specified for prototype agreement */
adf_tree *tree;
struct svc_req *UNUSED;
{
    static int result;  /* must be static! */
    double root_id;
    int error_flag;

    ADFM_Set_IO_Ptr(tree->data.data_val,&error_flag);
    ADFM_Set_IO_Mode(ADFM_IO_XDR_PLAIN,&error_flag);
    ADFM_Database_Open("NO FILE","old"," ",&root_id,&error_flag);
    print_child_list(root_id,0);

    result = 1;
    return (&result);
}
```

4.4 MPI

The last example is a MPI multiprocessus program. It has the same design as previous examples, a processor is

```
#include "mpi.h"
#include <stdio.h>
#include "ADFM.h"

void print_child_list(double node_id,int dec);

int main( argc, argv )
int argc;
char **argv;
{
  int rank,size,treesize,treeid,error_flag;
  MPI_Datatype T;
  void *tree;
  void *map1;
  char filename[256]="../Z.adfm";
  double root_id;
  MPI_Status st;

  MPI_Init( &argc, &argv );

  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );

  printf( "Starting %d of %d\n",rank, size );

  treeid=234;
  treesize=1024; /* hard coded so far... */

  if (rank == 0)
  {
    tree = (void*) malloc(treesize);
    T=MPI_UNSIGNED_CHAR;

    ADFM_Set_IO_Mode(ADFM_IO_FILE,&error_flag);
    ADFM_Database_Open(filename,"old"," ",&root_id,&error_flag);
    ADFM_Set_IO_Mode(ADFM_IO_XDR_PLAIN,&error_flag);
    ADFM_Database_Close(root_id,&error_flag);
    ADFM_Get_IO_Ptr((void**)(&map1),&error_flag);
    memcpy(tree,map1,treesize);

    error_flag=MPI_Send(tree, treesize, T, 1, treeid, MPI_COMM_WORLD);
    free(tree);

  }
  else if (rank == 1)
  {
    tree = (void*) malloc(treesize);
    T=MPI_UNSIGNED_CHAR;

    MPI_Recv(tree, treesize, T, 0, treeid, MPI_COMM_WORLD,&st);

    ADFM_Set_IO_Ptr(tree,&error_flag);
    ADFM_Set_IO_Mode(ADFM_IO_XDR_PLAIN,&error_flag);
    ADFM_Database_Open("NO FILE","old"," ",&root_id,&error_flag);
    print_child_list(root_id,0);
    free(tree);
  }

  MPI_Finalize();
  return 0;
}
```

5 Conclusion

We are believing the in-memory tree representation is important. The first step we actually have achieved with ADFM requires now some more tests, checks, concensus and documentation.
The ADFM lib and API will be delivered into the *pyCGNS* v0.4 package, even if this is out of Python scope.

I expect to have the CGNS community comments, about the system itself, the API, the needs, requirements…