The ALICE Memory Snooper

3/31/1999

Ibrahima Ba and Barry Smith

Argonne National Laboratory

http://www.mcs.anl.gov/ams

Revised 6/2010

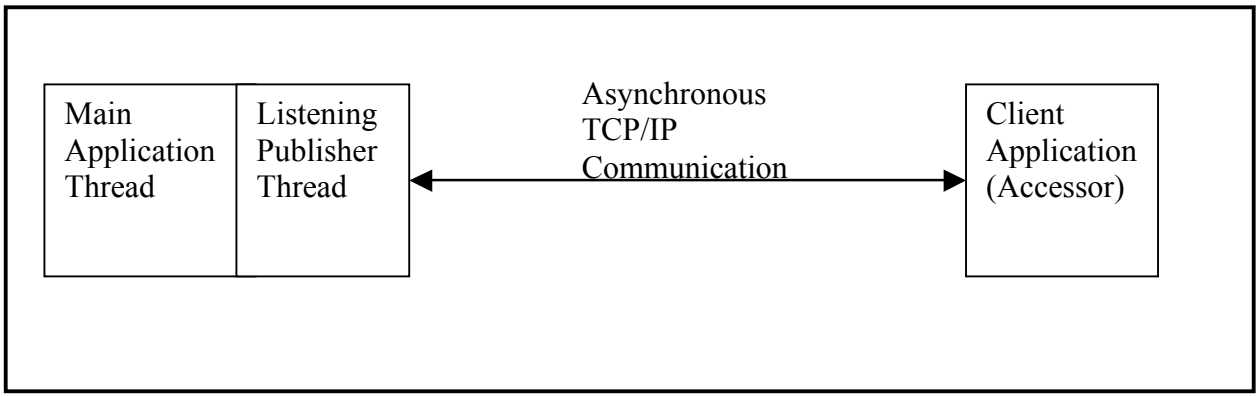
Barry Smith and Lisandro Dalcin

Introduction

The ALICE MEMORY "SNOOPER" (AMS) is an application programming interface (API) to help in writing computational steering, monitoring and debugging tools. The motivation for the AMS is to let users connect to the "running application" and access or modify variables (memory). Current monitoring systems require a lot of custom programming, rely on third party communication libraries, and are difficult to port to other platforms. We tried to resolve some of these problems by bundling with the API a portable communication library using TCP/IP, providing the API in the C language, and building general purpose C, JAVA GUI, MATLAB, and VTK monitoring clients. The AMS is distributed free of charge, and the source code is publicly available. It is usable from C, C++, and Fortran.

ALICE MEMORY SNOOPER API

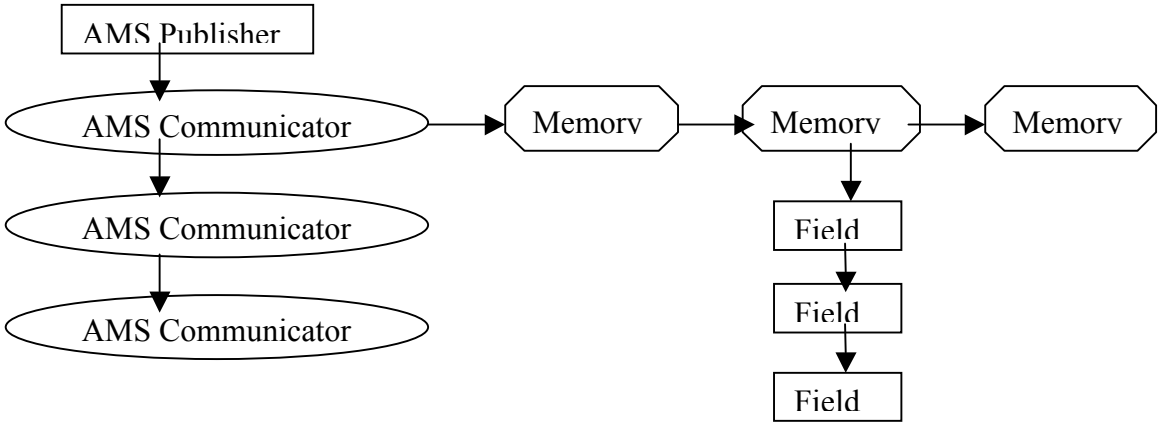
With the AMS API, a programmer only needs to make minor changes to the running application to publish (make accessible to other processes) its memory. In particular, the running application does not have to be interrupted by the client (Accessor); the Accessor "snoops" on the application. The design of the API relies on the use of threads for creating tasks to process requests from clients, and uses TCP/IP to communicate between the client program (Accessor) and the main application program (Publisher).



AMS Design

An important goal of the AMS is to provide a high-level API that manages most of the low-level, tedious-to-program details (communications and threads), and yet is flexible enough to let programmers build on top of it custom computational steering, debugging, and monitoring systems.

The AMS implements an object-oriented design in C. C was chosen for maximum portability and to maximize the number of languages in which the clients and servers could be written. Data structures within the API are created, manipulated, and destroyed through low-level API's. There are four types of objects used in the AMS. The hierarchical relationship among these objects is shown in the next figure.



The following table shows how the levels of the AMS object hierarchy fit into the structure of a typical running application:

Each	*Has its own*
Running application	Publisher
Component or library	AMS Communicator

Object	Memory
Array or variable	Field

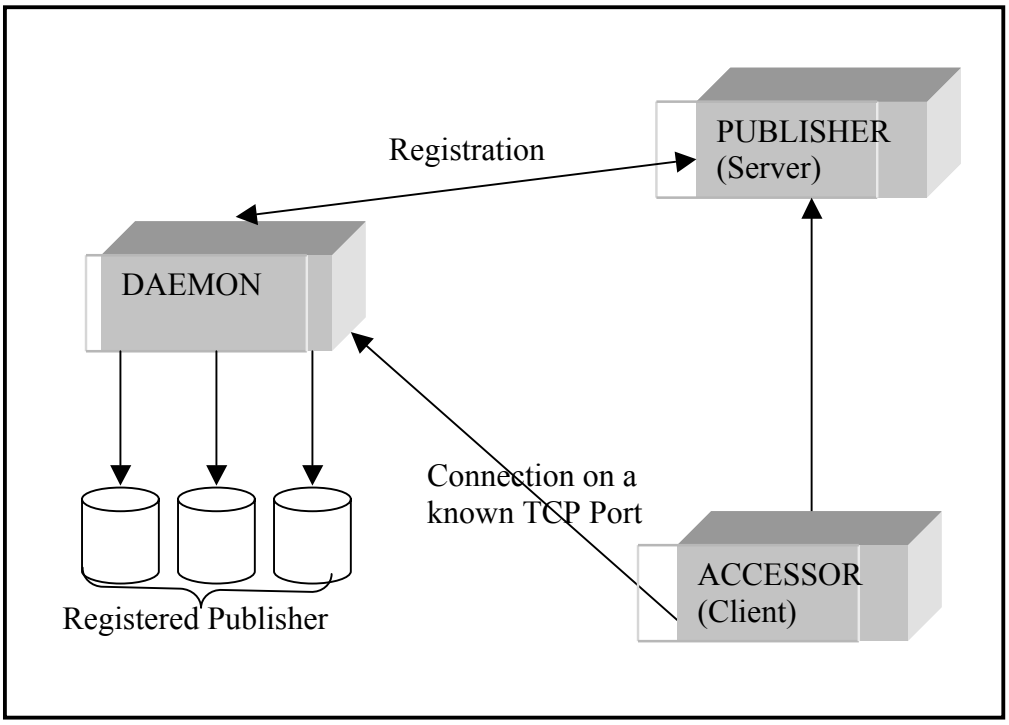
The purpose of the Publisher is to create worker threads that will respond to client requests and ensure the proper synchronization among the different threads in the main application. The Publisher API manages the creation, publishing, and destruction of the memory objects. The AMS Publisher design is based on the use of threads that handle requests from clients. Threads are concurrent tasks within a process.

The Accessor API is the client side for the AMS. The Accessor is used to access the server's memory in a consistent and transparent way. The API handles the communication connection, coding and decoding of requests, data transfer, byte ordering, and the creation of memory objects on the client side that mirror objects created on the server. The API also manages all the consistency and integrity of these objects (i.e., access control, type validation).

The Publisher Object

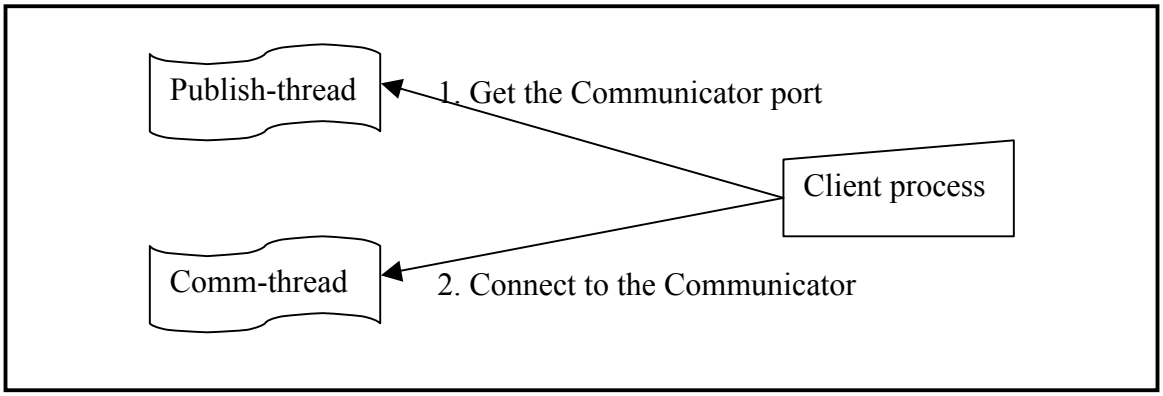
At the top of the hierarchy is the Publisher object. This data structure uniquely identifies the application being monitored. The Publisher's identity is given by the hostname and port number to connect to. The Publisher is started by the first call of *AMS_Comm_publish*. This call creates a server thread that listens on a particular TCP port (8967 by default) for incoming requests. The user could change this port by the setting the environment variable *AMS_ PORT*. The server thread is the first point of connection for a client program. The thread manages and keeps track of all current connections and pending requests. It is also used to properly shutdown the server.

In the future, we intend to develop a daemon program for example using LDAP that will handle some of the server thread work on a large network (LAN/WAN) based system. Client programs could connect to the daemon on a known port or through a broadcast call on a LAN. The daemon would then list all current Publishers and addresses. The daemon also could authenticate clients and verify their access attributes. This would manage potential conflicts among different clients trying to monitor the same program. The following figure shows the role of the daemon program would play in the AMS system.



The AMS Communicator Object

The AMS Communicator object is the data structure that logically encapsulates all "related" memory objects within a program. An AMS Communicator, identified by a unique name and port number, is also a server thread that manages a collection of memory objects. The idea behind an AMS Communicator is that applications could consist of several components and libraries. These components or libraries may, each separately, be linked with the AMS API and have separate AMS Communicator objects that could be published and monitored.
As mentioned above, the AMS Communicator object is created by a call to AMS_Comm_publish. This call creates a data structure associated with the AMS Communicator object and starts a server-thread that will listen to requests addressed to this particular AMS Communicator. This thread (AMS Communicator thread) finds an available port on its host system, informs the Publisher of this port number, and then waits for incoming requests on that port. A client issuing a connection with the server accesses the Publisher first to get the AMS Communicator's port number. The following figure illustrates the connection scheme.



The AMS Communicator's data structure keeps information on currently connected clients. When a client process sends a request to the AMS Communicator thread, a new worker thread is spawned to handle the request. The AMS Communicator will not shut itself down unless all the worker threads have processed pending requests. If a request arrives after the AMS Communicator has started shutting down, the worker thread sends back a shutdown warning to the client program.

Besides keeping track of clients, the AMS Communicator maintains a linked list of Memory objects. The following section discusses the role of these objects in the AMS design.

The Memory Object

A Memory object maintains a collection of related memory fields, which represent variables. Any instance of an application being monitored contains at least three threads that would have to compete for access to the memory: the main application thread, the AMS Communicator thread, and the worker thread. Before reading or modifying an individual memory field, the application must first "take access" to (lock) the Memory object that holds that field. Afterwards, the thread must "grant access" to (unlock) it so other threads could use it. This synchronization procedure is the only significant change required in the user application for enabling the use of the AMS.

Because access is granted to entire Memory objects, not to individual memory fields, Memory objects are the basic units of data synchronization. When one variable in a Memory object is changed, the AMS assumes that any of them might have changed. In an object-oriented program, you might want to group the member variables of an object into one Memory object. To ensure that related variables in the same Memory object are automatically updated properly, the Publisher allows the user to register "setter methods" to be used when changing a memory field (with the API call *AMS_Memory_set_field_func*). When the AMS Communicator receives a request from

the Accessor to modify a variable, it would lock the memory, and then call the user-defined setter method in order to update the field properly.

Each memory object maintains a current step number, which is useful in synchronizing the updates made by different processors in a parallel environment. Unless you explicitly "take read access" only, the AMS increments the Memory's current step number for that processor each time you take access to a Memory object. In a parallel environment, when an Accessor requests a Memory object, the processors will agree on a single step number at which they will each send data to (or take data from) the client.

Field object

Individual variables or arrays are stored in Field objects, each of which belongs to a particular Memory object. Field objects are at the bottom of the AMS object hierarchy, and are created by a call to *AMS_Memory_add_field*. Besides keeping a pointer to variable data, the Field object contains information on the user memory type (integer, float, double, or string), length, access (read/write) and the shared type (distributed, common, or reduced). *Distributed* arrays are those that are stored across different processors. If a Field is declared *distributed*, the AMS will contact each processor to get its part of the data. Fields that are declared *common* are accessed on only one processor.

Field objects should be viewed not only as a pointer to data, but also as any entity that can be manipulated from the client side. In the future, we plan to support user functions as type of fields. We are not trying to implement a remote method invocation protocol, but simply to provide the user with basic object types. If you are interested in more complex systems, you should consider standards in distributed computing such as CORBA, JAVA RMI, or DCOM/COM+.

Field objects have almost the same representation on both the client and the server, except that on the server we do not store any user data in the field structure, but instead, we keep a pointer to the data itself. This allows all threads within the application, provided they are synchronized, to access that same memory. The Field object stores the number of elements in its array. Using the type of the field, we determine the actual byte size of the field. This model allows us to store and retrieve a more complex data structure, such as list of strings. The following graph summarizes the underlying structure of the Field object. Below is an explanation of the different fields.

Pointer to User Data
Data type
Data length
Reduction type
Shared type

The Shared type could be defined as AMS_COMMON, AMS_REDUCED, or AMS_DISTRIBUTED. The AMS_DISTRIBUTED type is used only in a parallel environment.

Header file and AMS Data types

All the AMS high-level objects described above are defined in the header file *ams.h*. Note that the user does not have direct access to the Field object. The AMS API hides the complexity of these high-level objects and presents them as integers to the user. The *ams.h* header file also contains a list of error codes used in the API. Each error code explanation can be obtained by a call to *AMS_Explain_error*. By default, error messages are printed to stderr; you can redirect them to a file with the call *AMS_Set_option(AMS_LOG_FILE, "myfilename")*.

Following is a list of the most common types used in the AMS API:

AMS Type	**Object it represents or meaning**
AMS_Memory	Memory object
AMS_Memory_type	Memory access type (read, write)
AMS_Data_type	Field data type (*e.g.* integer, float, double, boolean, string)
AMS_Shared_type	Field distribution type (AMS_COMMON)
AMS_Reduction_type	How a field is distributed across different processors (AMS_MIN, AMS_MAX, or AMS_SUM) (only useful for reduction type data)
AMS_Comm	AMS Communicator object
AMS_Comm_type	MPI processors or Clusters of workstations (use NODE_TYPE, or MPI_TYPE)

Now that we have described the different data objects in the AMS, let us look at their implementation and how the AMS API manipulates these objects. There are two sets of APIs: The Publisher API, linked with the user's main application, and the Accessor API, used by the user to communicate/interact with the main application.

The Publisher API

The Publisher API is the AMS component to be used in the server (main application) side. The API consists of a set of calls designed to create threads (tasks) that will listen to incoming requests from clients. Other API calls are used to create memory objects and to ensure proper synchronization among concurrent threads. In this section, we will enumerate the steps needed to create a Publisher. The example used in this section can be found in the *testpub.c* file.

Creating a Publisher and an AMS Communicator

A Publisher is created with the first call that creates an AMS Communicator. To create an AMS Communicator, the following API call is used:

```
….
AMS_Comm comm;                  /* AMS Communicator */
char *msg;                      /* Pointer to error messages */
int err;

err = AMS_Comm_publish("simple", &comm, NODE_TYPE, NULL , NULL);
AMS_Check_error(err, &msg);
….
```

This sequence creates an AMS Communicator called "*simple*". The call returns the id of the AMS Communicator through the variable comm. The NODE_TYPE indicates that we are not using MPI processors. The fourth parameter (a list of hostnames) is set to NULL to use just the local host. The last parameter (a list of port numbers corresponding to the hosts in the list) is set to NULL to let the system determine the port numbers.
The call to *AMS_Check_error* macro will display a message if an error has occurred. (It will give you a pointer to the error message in *msg*). If the call is successful, a Publisher is started and an AMS Communicator is created and started. At this moment, we have three threads running in the main application. One thread representing the user application, another thread representing the Publisher. This thread is listening on a known TCP port for a new connecting client. A third thread is waiting (on a TCP port) for particular requests to the AMS Communicator. The next call to *AMS_Comm_publish* will create only an AMS Communicator; the Publisher is created only in the first API call.

Creating a Memory object

After an AMS Communicator is created, a Memory object can now be created and attached to it.

```
….
AMS_Memory memory;
err = AMS_Memory_create(comm, "simple_memory", &memory);
AMS_Check_error(err, &msg);
….
```

This call creates an AMS Memory object called "*simple_memory*" that is identified by the id returned in the memory variable. This call will fail if the AMS Communicator identified by *comm* is not valid (has not been successfully published) or the Memory name is already in use.

Adding Field Object(s) to an AMS Memory

Memory Field's objects are created by the user and attached to an existing Memory object. The Field object is not in itself manipulated by the user. Through the API, the user provides the properties of the Field. The Memory has to be destroyed in order to delete all the fields in that Memory. To add a Field, the following calling sequence is used:

```
….
int int_elem = 1;                    /* An integer field to be published */
err = AMS_Memory_add_field(memory, "int_elem", &int_elem, 1, AMS_INT,
                           AMS_WRITE, AMS_COMMON, AMS_REDUCT_UNDEF);
AMS_Check_error(err, &msg);
….
```

This call adds a Field object called "*int_elem*" to the Memory. The call will fail if the Memory object is invalid (has not been successfully created), or the Field name already exists in this Memory, or if any of the other parameters is invalid. Note that we pass the address of the integer *int_elem* to the API call. This address is referenced whenever we need access to this field value for reading or writing (updating). So the Field's data are not stored, only a pointer to the data is kept. The Field's length is also passed as a parameter. In our example, *int_elem* has a length of one (1). This length does not represent the number of bytes (which is sizeof(int)), but instead the number of integers in the field. The other parameters describe the data type (AMS_INT for integer), the access type (AMS_WRITE), the shared type (AMS_COMMON), and the reduction type (AMS_REDUCT_UNDEF). The reduction type is always undefined when the shared type is common. The API supports other data types (AMS_FLOAT, AMS_DOUBLE, and AMS_STRING). The string type could be used for publishing non-numerical fields such as a program stack (function names), object names (to monitor for instance the creation and destruction of objects within a program which is useful for debugging memory leaks), and general logging information that traditionally is dumped to a file. Two access types are provided: AMS_READ and AMS_WRITE. To allow a client to modify the Field's data, you must designate AMS_WRITE access.

If successful, this API call will create a Field object and attach it to the Memory. Other Fields can be created and added to the same memory. The following sample code adds an array of float to the same Memory:

```
….
float float_arr[20];     /* A field of arrays to be published */

err = AMS_Memory_add_field(memory, "float_arr", float_arr, 20,
```

```
AMS_FLOAT, AMS_READ, AMS_COMMON, AMS_REDUCT_UNDEF);

AMS_Check_error(err, &msg);

….
```

Setting Field Dimensions for Multi-dimensional arrays:

When publishing a multi-dimensional array, the user must specify the array's dimension so that a client could retrieve the information and reconstruct the array. The AMS provides the following API call to set the dimension information:

```
….
float float_arr[20][10];          /* A 2-d array to be published */
int dim = 2;                      /* Number of dimension of the array */
int start_ind[2], end_ind[2];   /* Array of starting and ending indices */

start_ind[0] = 0;         /* Starting index in the first dimension */
end_ind[0] = 19;          /* Ending index in the first dimension */
start_ind[1] = 0;         /* Starting index in the second dimension */
end_ind[1] = 9;           /* Ending index in the second dimension */

err = AMS_Memory_set_field_block(memory, "float_arr", dim, start_ind, end_ind);
AMS_Check_error(err, &msg);
….
```

Now that we created the last object (Field) in the chain, we need to signal the application to Publish the Memory so clients could access it.

Publishing the Memory:

The following call makes a Memory and its attached fields available to connecting clients:

```
….
err = AMS_Memory_publish(memory);
AMS_Check_error(err, &msg);
….
```

If successful, clients program can connect to the AMS Communicator, and browse the list of Memory within that Communicator. Each Memory will have a list of Field attached to it.
Once the Memory is published, the user has to deal with synchronization issues. This is the most important part of the AMS API. The following section describes the synchronization calls:

Granting (Taking) Access to (from) Other Threads:

Three API calls are used for synchronization among threads. These calls are placed before and after any use of a variable that is published in a Field. Depending on how fast and important are the changes in the published Field, the user could either perform a global lock at a high level (outside a loop for instance), or a local lock. The following segment of code uses what we call local locks:

```
….
for(i = 0; i < 20; i++) {
        /* Take access */
        err = AMS_Memory_take_write_access(memory);
        AMS_Check_error(err, &msg);

        /* Perform some operations on the data */
        if (float_arr[i] || i%2)
                float_arr[i] = 0;
        else
                float_arr[i] = int_elem;

        /* Grant Access */
        err = AMS_Memory_grant_write_access(memory);
        AMS_Check_error(err, &msg);
}
….
```

In the segment above, we used *AMS_Memory_take_write_access* because the *float_arr* field is being modified by the application. Otherwise, *AMS_Memory_take_read_access* would have been more efficient since client threads seeking read access would not block if the server (main application) has only read access. In other words, multiple threads may gain simultaneous read accesses while only one thread can gain write access. While the write access is taken, requests by other threads are blocked. The AMS API uses the same synchronization scheme to maintain consistency among objects in the Accessor and the Publisher. One should weigh the advantage of local vs. global locking strategies. Remember that synchronization resources are shared among different threads and locking a Memory for a long time may delay communications and requests processing from the clients and the server.

Note: Once a memory object is published, no particular thread owns a lock to it. Therefore, there is no need initially to call *AMS_Memory_grant_write_access* to allow other threads to access the memory. The main thread must take access every time it needs to read or modify the memory once the memory is published, and grant access afterwards, until the memory is unpublished. You cannot grant access to a memory if you did not taken access to it first. In addition, to release its ownership, the thread must call *AMS_Memory_grant_write_access* once for each time that

AMS_Memory_take_write_access was called, and *AMS_Memory_grant_read_access* for each time *AMS_Memory_take_read_access* was called.

Waiting Access from Other Threads:

Two API calls are used for conditional waiting among threads. These API are very important if the thread is waiting for certain event to happen to proceed. The waits are not active (low CPU usage). The following segment of code uses what we call local locks. The following segment will block the thread until the memory is read, or a timeout occurs.

```
/* Wait read access */
err = AMS_Memory_wait_read_access(memory, timeout);
AMS_Check_error(err, &msg);
```

The next segment will block the calling thread until the memory is written by another thread.

```
/* Wait written access */
err = AMS_Memory_wait_written_access(memory, timeout);
AMS_Check_error(err, &msg);
```

Destroying AMS Objects:

The user is responsible for destroying all objects previously created. It is important that these objects be destroyed so that the associated resources (memory, compute threads, etc…) be released. There are two AMS objects to be destroyed: the Communicator object and the Memory object. The following calls are used:

```
….
err = AMS_Memory_destroy(memory);        /* Destroy the Memory Object */
err = AMS_Comm_destroy(comm);   /* Destroy the Communicator */
….
```

Destroying the Memory object will release the resources associated with it and with all the fields attached to the Memory. It will also notify current connected clients that the Memory is being unpublished. If clients are not notified Memory locks might not be released. This could result in deadlocks and clients waiting for response to their requests. We recommend that all objects be destroyed before the program ends or aborts. Destroying the AMS Communicator objects is important. It shuts down the communication thread, and destroys all attached memories and fields.

The Accessor API

The Accessor represents the client component of the API. Once a main application is running the published information (memory, fields) can be accessed. The Accessor API provides a set of general-purpose, high-level calls that allow the user to access the information in the main application. In this section, we will discuss in detail each API call and give a context in which it is used. For more information on actual implementation, please refer to the monitor program (*mont*) provided with the examples source code. That program interprets simple user commands from the prompt or a file and executes the corresponding API calls. A Java GUI interface and a MATLAB interface are available on some architectures to demonstrate how one could build complex user interface clients on top of the API. We expect most users to build custom clients for their different needs (interacting, steering, monitoring, debugging, etc.).

Communication between the server (main application) and the clients is stateless (asynchronous) in that the connections are limited to specific requests. This model is a transactional-based model. Once a request is completed, the server does not remember the client. This method has a big advantage in that the server does not have to worry about idle client programs that do not release the much need communication resources. The drawback is that the user has to connect and disconnect for every request. However, we tried to bundle the most common and related requests in the same API calls. In the future, we plan to publish the specification of the communication protocol (encoding and decoding of requests) between the client and the server so that motivated users could develop their own clients API. Our implementation of the protocol is adapted from the tftp (trivial file transfer protocol) program by Richard Stevens[1].

The following sub-sections detail each Accessor API call. The order is important in that it reflects what a typical client program will have to do.

Connection to the Publisher:

The first step is a connection to the Publisher. A port number and a host name are inputted; the call returns the list of published AMS Communicators and how to access them (hostnames and port numbers):

```
….
int       port = -1, err, i;
char host[255], **comm_list, buff[255], *p, *msg;

/* Get the Communicators' list */
err = AMS_Connect(host, port, &comm_list);
AMS_Check_error(err, &msg);
….
```

[1] http://www.kohala.com/~rstevens/unp.html

If successful, the *comm_list* parameter will contain the list of AMS Communicators separated by the pipe "|" sign. If the Publisher has only one AMS Communicator then, the user need not to parse the *comm_list* parameter. This parameter can be used to attach the AMS Communicator without further processing. However, the following segment code shows how one may extract the AMS Communicator name from the list:

```
….
while (comm_list[i] && i < MAX_COMM) {
        strcpy(buff, comm_list[i]);
        p = strtok(buff, "|");
        printf("\t %s ", p);
        p = strtok(NULL, "|");
        printf("(host = %s) ", p);
        p = strtok(NULL, "|");
        printf("(port = %s) \n", p);
        i++;
}
….
```

MAX_COMM is a constant defined in *ams.h*. It defines the maximum number of AMS Communicators the main application could publish. Note that an AMS Communicator can have multiple hosts in a parallel environment.
Given the information for each AMS Communicator, the user can now attach to an particular AMS Communicator.

Attaching an AMS Communicator:

To attach an AMS Communicator, one need only the AMS Communicator's name. The request is actually sent to the Publisher. Attaching to a Communicator is creating a mirror of the server's object on the client side. This mirror object contains the list of hosts belonging to the same AMS Communicator and their port numbers. The following call attaches a Communicator:

```
….
AMS_Comm alice;

/* Attach to a Communicator */
err = AMS_Comm_attach(com_name, &alice);
AMS_Check_error(err, &msg);
….
```

The call will fail if the *com_name* is not published in the server side. The variable *alice* will contain a valid id for the AMS Communicator. The API will build a structure almost identical to the one on the server. If the AMS Communicator were already attached, this

call will de-attach it first, and re-attach it to make sure that the current AMS Communicator's state is consistent with the one the server.

Getting the Memory list:

Once attached to a Communicator, the user needs to get the list of published memories. The following call provides such a list:

```
….
char **mem_list, *msg;

/* Get the memory list */
err = AMS_Comm_get_memory_list(alice, &mem_list);
AMS_Check_error(err, &msg);

/* Print the list */
while (mem_list[i])
        printf("%s\n", mem_list[i++]);

….
```

The user should copy the memory list into a local buffer. Otherwise, subsequent calls to AMS_Comm_get_memory_list will override the current list. The *mem_list* variable contains names of all published memories. The user then needs to attach a particular memory for more details (its fields). This API call will fail if the AMS Communicator *alice* has not been attached.

Attaching a Memory:

A Memory object is attached with the following call:

```
….
char mem_name[255];                 /* Memory name */
unsigned int step;                  /* Memory step number */
AMS_Memory memory;

/* Copy the Memory name you want to attach to it */
/* This example uses "my_memory" as a name */
strcpy(mem_name, "my_memory");

/* Attach the memory structure */
err = AMS_Memory_attach(alice, mem_name, &memory, &step);
AMS_Check_error(err, &msg);

….
```

The call attaches the Memory identified by the string in *mem_name*. The first input parameter (*alice*) is the AMS Communicator that contains this memory. The third parameter is an output id of the Memory object, and the last parameter indicates the Memory step number. This identification indicates the Memory version number. Once, a Memory is attached, a mirror copy of the server's Memory object is created on the client side. You can attach multiple memories at the same time by separating them with "|". Now we have attached a Memory, we are ready to look at its Fields. The next API calls list the Fields' name, access and modify their content.

Getting the list of Fields attached to a Memory:

As we described in the API Design section, the Fields are the low-level objects in the AMS API hierarchy. These objects provide copies and properties of the physical memory in which we are interested. The next call allows the user to retrieve a list of names of Fields attached to a particular Memory, not the objects themselves.

```
….
char **fld_list;

err = AMS_Memory_get_field_list(memory, &fld_list);
AMS_Check_error(err, &msg);
….
```

Again, one may need to save a copy of the list so that subsequent calls to the API will not override the list. This call will fail if the Memory has not been attached or the given id, *memory*, is invalid. Given the list of fields, the user can now request information on a particular field. This is performed by the next API call.

Getting the Field's Properties

Users do not have direct access to the Field object, instead, through the following call, they retrieve the different attributes of a Field.

```
….
AMS_Memory                memory;       /* Memory id */
AMS_Memory_type   mtype;          /* Memory type */
AMS_Data_type             dtype;        /* Data type */
AMS_Shared_type           stype;        /* Shared type */
AMS_Reduction_type rtype;         /* Reduction type */

int     len;                            /* Data length */
void    *addr;                          /* Pointer to the data */
char    *fld_name;                      /* Input parameter, field name */

/* Get Field info */
err = AMS_Memory_get_field_info(memory, fld_name, &addr, &len, &dtype, &mtype,
```

```
                                        &stype, &rtype);
AMS_Check_error(err, &msg);
….
```

If successful, this call returns all the information regarding the Field. It also returns a pointer, *addr*, to a copy of the data. Using the variable *dtype*, and *len*, the caller can print the data in *addr*, as follows:

```
….
char  **tmpstr;

/* Data type and data format */
switch(dtype) {
case AMS_INT:
        printf("\t Data type: Integer \n");
        printf("\t Data value: \n");
        for (i=0; i<len;i++)
                printf("\n\t   [%d] = %d",i, *((int *)(addr) + i ));
        break;

case AMS_DOUBLE:
        printf("\t Data type: Double \n");
        printf("\t Data value: \n");
        for (i=0; i<len;i++)
                printf("\n\t   [%d] = %8.2f",i, *((double *)(addr) + i ));
        break;

case AMS_FLOAT:
        printf("\t Data type: Float \n");
        printf("\t Data value: \n");
        for (i=0; i<len;i++)
                printf("\n\t   [%d] = %8.2f ",i, *((float *)(addr) + i ));
        break;

case AMS_STRING:
        printf("\t Data type: String \n");
        printf("\t Data value: \n");
        tmpstr = (char **)addr;
        for (i=0; i<len;i++) {
                if (tmpstr[i])
                        printf("\n\t   [%d] = %s ",i, tmpstr[i]);
                else
                        printf("\n\t   [%d] = null ",i);
        }
        break;
….
default:
```

```
        printf("\t Data type: Undefined \n");
        break;
}
….
```

The constant *AMS_INT*, *AMS_FLOAT*, *AMS_DOUBLE*, *AMS_BOOLEAN,* and *AMS_STRING* are defined in the header file *ams.h.* Note that *len* represents the number of elements and not the data length (number of bytes).

Getting the Field's Size for Multi-dimensional arrays

The AMS has support for multi-dimensional arrays. The dimensions of an array can be retrieved by a call to AMS_Memory_get_field_block:

```
…..
int dim, *start, *end;

err = AMS_Memory_get_field_block(mem, fld_name, &dim, &start, &end);
AMS_Check_error(err, &msg);

….
```

dim will indicate the number of dimension of the array, and *start*, *end* will hold the staring index and ending index for each dimension. For instance, a two-dimensional array, A[20, 10] will have the function return dim=2, start[0]=0, end[0]=19, start[1]=0 and end[1]=9.

Receiving an update from the server:

Often the client needs to receive updates of its local Memory copy from the Publisher. The following API call updates the client Memory:

```
….
AMS_Memory memory;
int  err, changed, step;
char *msg;

/* Receive an update */
err = AMS_Memory_update_recv_end(memory, &changed, &step);
AMS_Check_error(err, &msg);
….
```

If successful, the client will receive the latest copy of all the Fields attached to this Memory. This call will fail if the Memory has been unpublished (by the server) or detached (removed) by the client. The variable *changed* will be set to 1 if it downloaded

a new version from the server, and 0 if the client's copy of the memory was still the most recent and thus no download was needed. The variable *step* indicates what the server's current step of calculations. This numbers indicates how many times the server locked and unlocked this Memory. Since this variable is a type of integer, its value is given modulo MAX_INT or 2^32 - 1.

Setting the Field's Properties:

If the Field's memory type is AMS_WRITE, the user could modify the Field's data by the following API call:

```
….
AMS_Memory memory;          /* Input: Valid Memory id */
int            len;              /* Input: New Data length */
void           *addr;            /* Input: Pointer to the new data */
char           *fld_name;        /* Input: Field name */

err = AMS_Memory_set_field_info(memory, fld_name, addr, len);
AMS_Check_error(err, &msg);
….
```

This call updates the local copy of the Field identified by *fld_name* by the new data in *addr*. However, the remote copy is not yet updated. This allows the user to change others fields' data before sending an update. The actual update is performed by the next API call:

Updating the Publisher Memory:

Once the modification of the client copy of a Memory is complete, the user can now post an update request so that the server (Publisher of the Memory) gets the latest copy of the Memory.

```
….
AMS_Memory     memory;     /* Input: Valid Memory id */
int            err;        /* Output: Error code */
char           *msg;       /* Output: Error message */

err = AMS_Memory_update_send_begin(memory);
AMS_Check_error(err, &msg);
….
```

All the Fields that are attached to this particular Memory are sent to the server to update its copies. Upon return, the function indicates that the server has updated successfully (if there are no error messages) its Memory.

In the design phase, the AMS API provided support for Memory updates in the background. That is, the *AMS_Memory_update_send_begin* would return as soon as the data was sent to the server (not the completion of the update itself). With that scenario, another call, *AMS_Memory_update_send_end* would be necessary to let the client that the update has finished. So far, we have not decided to implement the latter API call. This may change in the future to at least give developers the choice of handling the update with one call or two.

Locking a Memory:

The Accessor API provides a way to lock a Memory object on the server, thus stopping the publisher main's threat. This is useful when the user would like to pause the application so that he/she can do some processing on the client side. This is done at the client or the server level. To detach a Memory, the next API call is used:

```
….
AMS_Memory        memory;
int               err, timeout;
char              *msg;

/* Detach the Memory */
err = AMS_Memory_lock(memory, timeout);
AMS_Check_error(err, &msg);
….
```

The application main thread will block until timeout milliseconds, or until the client calls *AMS_Memory_unlock()*. If *timeout* is 0, the main thread will block until the client unblocks it.

Locking a Memory:

The Accessor API provides a way to unlock a Memory object on the server, thus signaling the publisher main's threat to proceed. This is done at the client or the server level. To unlock a Memory, the next API call is used:

```
….
AMS_Memory        memory;
int               err;
char              *msg;

/* Detach the Memory */
err = AMS_Memory_unlock(memory);
AMS_Check_error(err, &msg);
….
```

Detaching a Memory:

The Accessor API provides a way to delete a Memory object and release the associated resources. Deleting a Memory does not involve any connection with the server. This is done at the client level. To detach a Memory, the next API call is used:

```
….
AMS_Memory        memory;
int               err;
char              *msg;

/* Detach the Memory */
err = AMS_Memory_deattach(memory);
AMS_Check_error(err, &msg);
….
```

The Memory is first disconnected from the local copy of the AMS Communicator and then deleted. This will also delete all the Fields connected to it.

Detaching an AMS Communicator:

When a client attaches an AMS Communicator, it is indicating that further requests regarding Memories and Fields are directed to the current attached AMS Communicator. Detaching a Communicator not only release the corresponding resources (memory), but also resets the current AMS Communicator's port. The server (Publisher) is notified by a client that a Communicator is being detached. This will be used in the future to manage concurrent access by different client to the same AMS Communicator. For instance, only one client would have the right to modify a certain Memory while others have read access. When this particular client detaches its AMS Communicator, the server could then give write access to another client.
To detach a client, the following API call is used:

```
…
AMS_Comm alice;
int  err;
char *msg;

/* Detach the Communicator */
err = AMS_Comm_deattach(alice);
AMS_Check_error(err, &msg);
….
```

API for both the Accessor and the Publisher

The following API's are used to both from the Accessor and the Publisher. They generally serve to control the behavior of the AMS libraries by changing some default behavior.

API Common to the Publisher and Accessor

AMS_Explain_error
AMS_Print
AMS_Memory_lock
AMS_Memory_unlock
AMS_Set_abort_func
AMS_Set_exit_func
AMS_Set_output_file

For more information, please go to the AMS Web page http://www.mcs.anl.gov/ams

The Monitor Program (mont)

The Monitor program (mont, or mont.exe in Windows) is our first example on using the Accessor API. The role of the Monitor is to give the user a quick start on testing a Publisher program. The Monitors implements basic commands such as connecting to the server, attaching a Communicator, or Memory, and displaying/modifying the content of a Memory. The program can also take a list of commands from a file and execute them.

Starting the Monitor:

The AMS installation program puts the Monitor program in src/examples directory. To run the program type the command:
mont

This command gives you back the Monitor prompt:
(mont)

From now on, we assume that the Publisher program *simple,* which is located in the same directory as *mont,* has been started on host called *host.mcs.anl.gov* for example. The following subsections describe a complete session of the Monitor.

Getting help:

You can get help from the prompt by typing “help” or “?”

```
(mont) help
?        - to get this help
ac       - to attach communicator. Syntax: ac <comm_name>
connect - to connect to server. Syntax: connect <hostname> [port#]
exit     - exit the interpreter
help     - to get this help
mf       - to modify a field value. Syntax: mf <fld_name> new_value
pf       - to print the content of a field. Syntax: pf <fld_name>
           The field must be in current focus memory
pm       - to print memory fields. Syntax: pm [mem_name]
           If memory name is omitted, the current focus memory is printed
pml      - to print the list of published memory. Syntax: pml
quit     - to exit the interpreter
set      - to set an option
sfc      - to set focus on a communicator. Syntax: sfc <comm_name>
           This will make this communicator the default for memory accesses
sfm      - to set focus on a memory. Syntax: sfm <mem_name>
           This will make this memory the default for field accesses
status   - to get current status of the monitor variables
verbose - not implemented yet
```

(mont)

Connecting to the server (Publisher):

To connect to the Publisher using the default port (8967) use the command "connect":

(mont) connect host.mcs.anl.gov
Connected. The following communicators are published:
simple (host = host.mcs.anl.gov) (port = 58875)
(mont)

The command returns the list of AMS Communicators that are published. In our example, one AMS Communicator (*simple*) is published. Note that the port number, 58875, is returned to indicate to the client how to reach this particular AMS Communicator.

Attaching a Communicator:

Once we have a list of Published AMS Communicators, we are ready to attach to one of them. Attaching to a Communicator means that future requests regarding Memory objects are sent to this AMS Communicator. To attach to AMS Communicator use the command "ac":

(mont)ac simple
Communicator simple has been attached
Published memory(ies):
simple_memory
(mont)

The command attaches the AMS Communicator "*simple*". It also returns the list of published Memory objects within this AMS Communicator. In our example, one Memory, *"simple_memory"*, is published.

Printing the content of a Memory:

Going further down in the hierarchy, we can print the content of the Memory object by using the command "pm":

(mont) pm simple_memory
Memory simple_memory contains the following fields
int_elem
float_arr
(mont)

Our Memory object contains two fields: *int_elem*, and *float_arr*. These represent the names of the fields as they were published. Note, that the command *pm* requires an

argument memory name, unless the *sfm "mem_name"* command has previously been type. The set focus memory (sfm) is used to set the default name to use for commands that require a memory name.

Printing the content of a Field:

Once the user got access to the Memory, the command "pf" prints the content of a Field that is attached to the current Memory. The following example prints the Field *int_elem*:

(mont)pf int_elem
Field int_elem descriptions:
Memory type: Read/Write
Shared type: COMMON
Data type: Integer
Data value:

[0] = 1
(mont)

The Field properties are printed, and its value is displayed. In this example, *int_elem* is an integer. It has one element with a value of one ([0] = 1). The access type of this Field is Read/Write, which means that the Field could be modified. The "*pf"* command first gets the latest copy from the server and then prints it. The next command displays the content of the second Field (*float_arr)* in our example, which is an array of floats that is read-only:

(mont)pf float_arr
Field float_arr descriptions:
Memory type: Read
Shared type: COMMON
Data type: Float
Data value:

[0] = 0.00
[1] = 0.00
[2] = 1.00
[3] = 0.00
[4] = 1.00
[5] = 0.00
[6] = 1.00
[7] = 0.00
[8] = 1.00
[9] = 0.00
[10] = 1.00
[11] = 0.00
[12] = 1.00

```
        [13] =    0.00
        [14] =    1.00
        [15] =    0.00
        [16] =    0.00
        [17] =    0.00
        [18] =    1.00
        [19] =    0.00
(mont)
```

Using the Monitor might not always be practical for large arrays. This is in general valid for many interpreters. We hope to overcome this in the future by adding more commands that will let users control how arrays and other objects are to be displayed or manipulated. Motivated users can always modify the Monitor program for customization.

Modifying a Field's content:

Read/Write Field objects can be modified from the Monitor prompt by using the command "mf". As soon as the command returns, the server (Publisher) copy of the Field has already been updated. We will show this by printing the Field's content just after its modification:

```
(mont)mf int_elem -3
(mont)pf int_elem
Field int_elem descriptions:
        Memory type: Read/Write
        Shared type: COMMON
        Data type: Integer
        Data value:

          [0] = -3
(mont)
```

In the last commands, we modified the Field int_elem value by –3 and then printed its new value.

Status command:

Often, the user wants to keep track of the status of the connection, which AMS Communicator is attached, or which Memory is active. The Monitor program provides the *status* command:

```
(mont)status
Connected
Active Communicator: simple
Active Memory: simple_memory
```

(mont)

We believe the Monitor program is a first step toward giving the user a tool to start with in developing and testing a Publisher program. This tool is limited but it has the advantage of running on all platforms (unlike the JAVA GUI, JAccessor). In future releases, we will add more features and enhances some of commands to make the Monitor an even more useful tool. We also welcome comments, requests for additional features and bug reports.

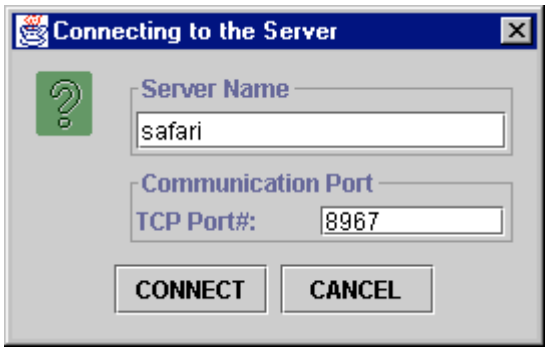
For those who have access to Windows platform or Solaris as client desktops, we recommend the use of the Alice Memory Browser, our JAVA GUI client, as a monitoring tool. The JAccessor GUI should run on other flavors of UNIX provided they have the jdk1.1.5 or later installed. The next section describes in details the how to use the JAccessor.

The ALICE Memory Browser Client Program

One easy way to access the user application is to use our Java client, the ALICE Memory Browser (AMB). The AMB uses Java Native Interface to access the AMS Accessor library and connect to the Publisher. The AMB is available on Windows, Solaris, and IRIX architectures.

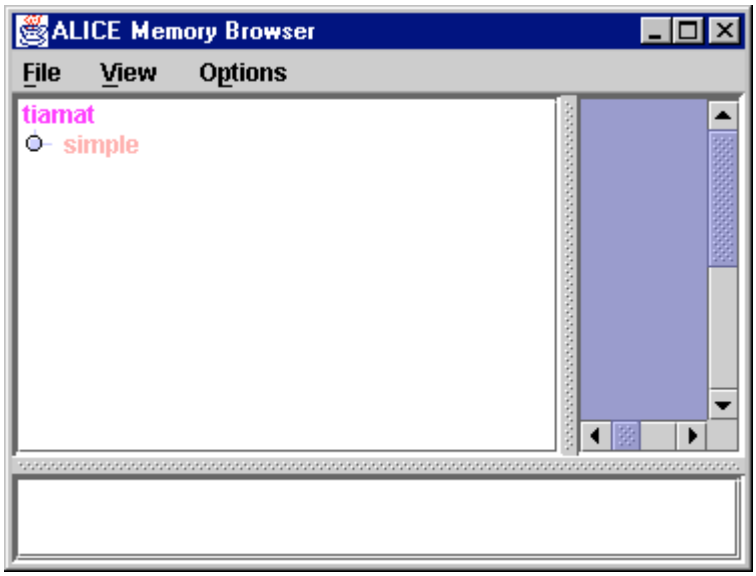
Starting the AMB

The AMB can be started using the script jams, or *jacc for jdk1.1.x*, located in the java/client directory in the AMS package. The following dialog box appears when starting the AMB:



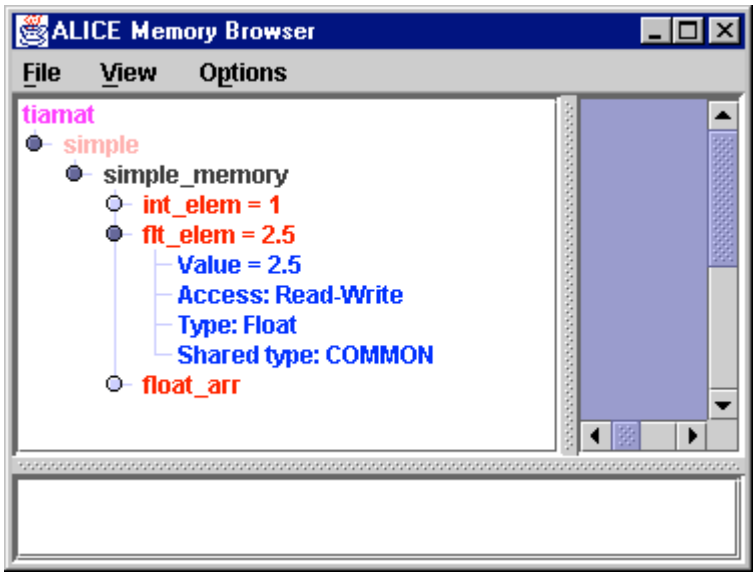
The AMB will prompt the user for the name of the server where the application is running on, and the TCP port number to use for communication. By default, the local hostname and the port 8967 are used. You can change such behavior by setting the environments variables AMS_SERVER, and AMS_PORT to the host name and port number you wish to use every time.

Upon connection, the AMB will display a list of AMS Communicators published by the user's application. The next figure shows that the AMS Communicator *simple* is published in the host *tiamat*:



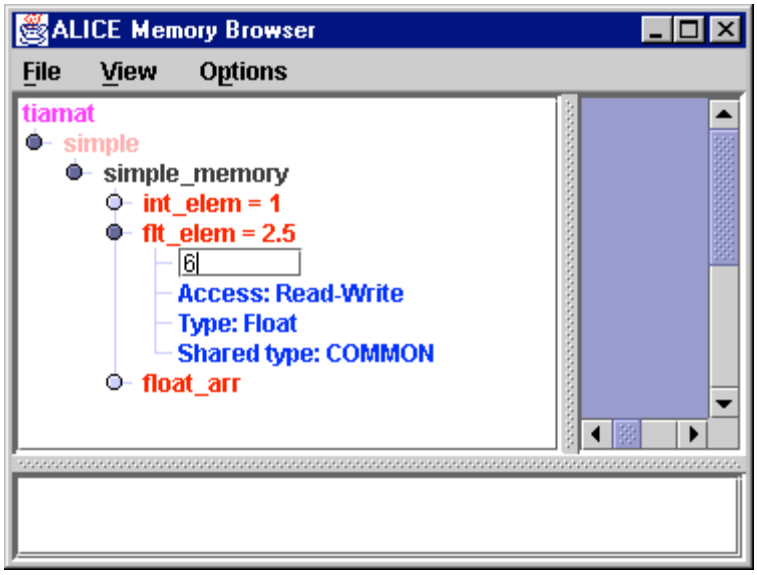
Using the ALICE Memory Browser

Published AMS objects are displayed within a tree structure. By double clicking on the tree nodes, the user can expand the different objects such as Communicators, Memory, and Fields. The next figures show the different AMS objects in the *simple* Communicator:

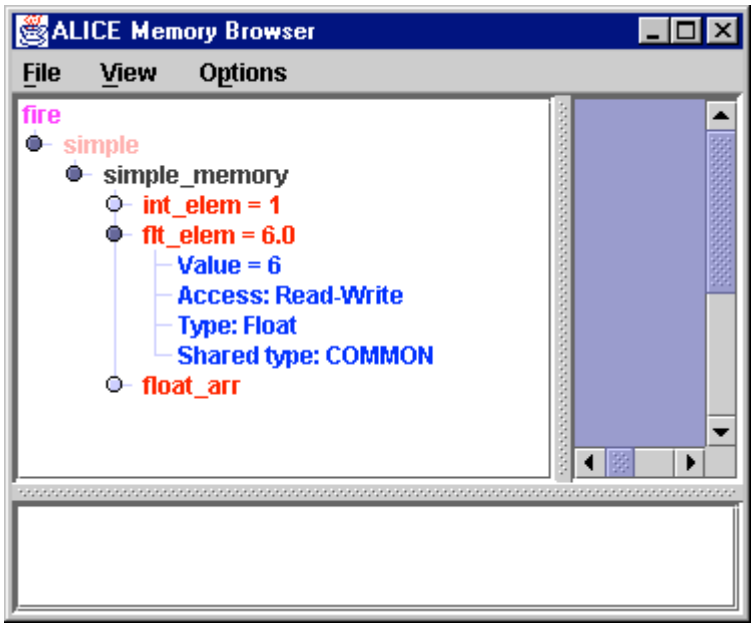


The above figure shows that AMS Memory *simple_memory* contains three fields: *int_elem, flt_elem,* and *float_arr*. If an AMS Field is atomic (has only one element), the field's value is displayed. The user can still expand the field to view its properties (access type, field type, etc…). In the above example, the field flt_elem is Read-Write which

means that we could modify its value. Double clicking on its value will allow the user to change its value as in the next figure:

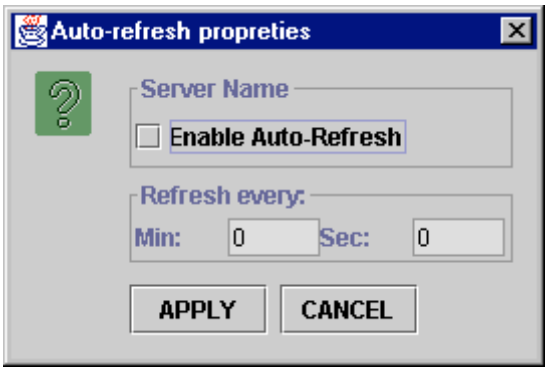


After entering a new value, and hitting the *Enter* key the AMB will change, update the server's copy for this field and display the new value as in the following figure:

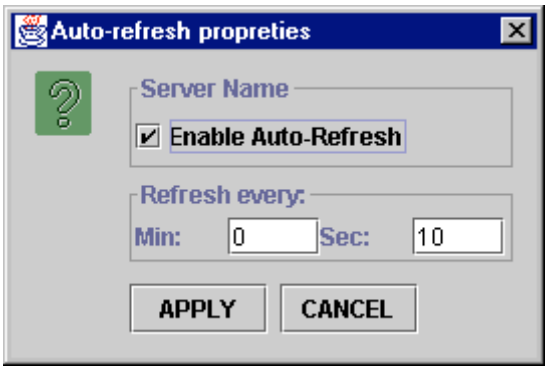


Monitoring an application with ALICE Memory Browser

In addition to the GUI interface to AMS, the ALICE Memory Browser provides other features such as "refreshing" the data periodically. This is important if the user is interested in monitoring the application while it is running. For example, to get the data from the application every 10 seconds, use the menu view and then auto-refresh (or ALT-F5 key). The following dialogue box appears:



You need first to enable the "Auto-Refresh" check box before and input the number of seconds desired as follow:

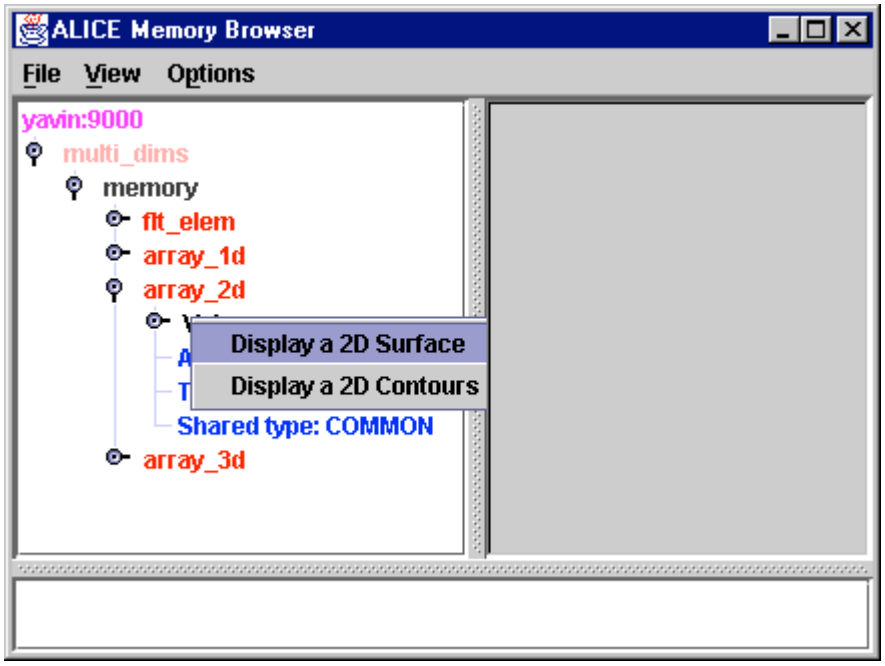


Upon clicking on the "APPLY" button, the AMB will refresh all the data that are currently displayed every 10 seconds for example.

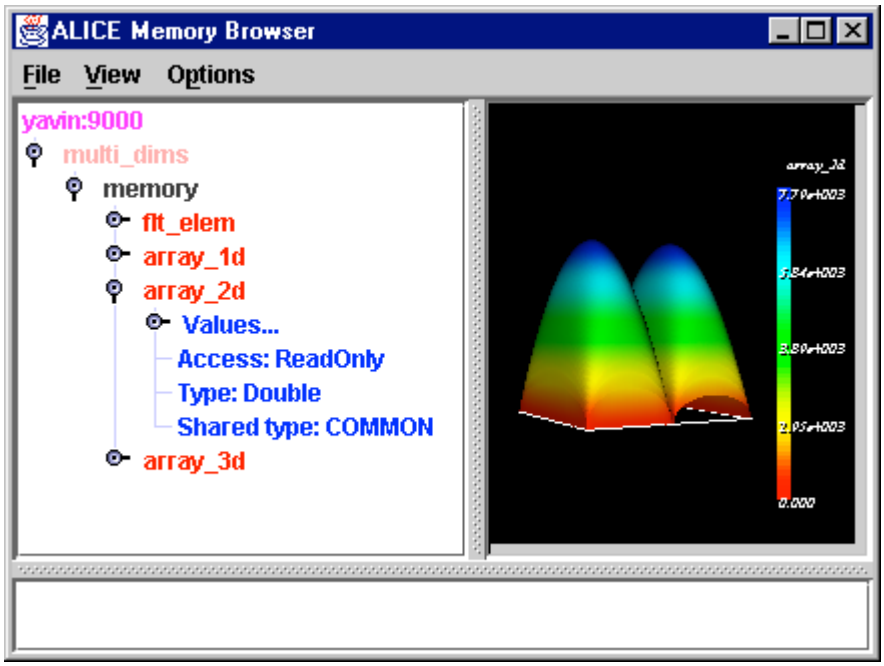
If the user wants only to refresh the data once, they can hit the F5 key or go to the menu view, and then refresh. Complex JAVA Application clients can be written using the AMSBean class (in AMSBean.jar). Its API is straightforward and documented on the AMS Web page http://www.mcs.anl.gov/ams (Java API).

Visualizing Data with the ALICE Memory Browser

In addition to the GUI interface to AMS, the ALICE Memory Browser provides the ability to visualize application data "on the fly". However, to use this feature you need to have VTK on your system. The AMS's Windows version provides support for VTK automatically if you select so at installation. We highly recommend that you do so if you are interested in visualization with the AMB. For example, in the following graph, the user has connected to an application that is publishing arrays of data. By clicking right mouse on the *values* of a field, a popup menu appears if the data can be visualized (1D, 2D, 3D, numerical type).



In the above example, we have a 2D array for which we can either plot contour lines, or a surface. Plotting a surface gives the following graph



Other plots can be obtained for 1D, 2D, and 3D data. Custom visualization applications can be written easily to visualize more complex data (unstructured grids).

The Matlab Client Program

The ALICE Memory Snooper comes with a Matlab interface to the Accessor API. Each function of the API is translated into Matlab using C-Mex files. This allows Matlab users to create custom AMS client to connect to their application. This is especially helpful if the user wants to do run-time visualization directly from the server to the client. Note that the Matlab client does not allow yet the user to change the server's values. This could be achieved separately by using the ALICE Memory Browser or the MONT program.

Using the Matlab Accessor API

Matlab files are located in the directory *matlab* under the AMS directory. In Windows environment, the *ams.dll* is used. The user can recompile it using *nmake*. In Unix, a makefile is available for building the Mex files. By default, these files are not built for UNIX. For both architectures, the user might need to edit the makefile to specify the path to Matlab Mex compiler.

The following is an example on how to use the Matlab Accessor API. We first start matlab from the AMS directory *matlab*, and get the standard matlab prompt. From the Matlab prompt », you can get help by typing *ams_help*:

» ams_help

ans =

Getting a variable
---
commlist = ams_connect(host,port)
comm = ams_comm_attach(commlist(i,:))
memlist = ams_comm_get_memory_list(comm)
[memory,step] = ams_memory_attach(comm,memlist(i,:))
fieldlist = ams_memory_get_field_list(memory)
data = ams_memory_get_field_info(memory,fieldlist(i,:))

Getting a variable repeatedly
---
ams_memory_update_rec_begin(memory)
[changed,step] = ams_memory_update_recv_end(memory)
data = ams_memory_get_field_info(memory,fieldlist(i,:))

Selecting a variable from menu
---
data = ams_view_select(host,port)

For example, to connect to a program running on host *tiamat*, type the following command:

» ams_connect('tiamat',-1)
ans =
simple|tiamat.mcs.anl.gov|35753
»

The –1 parameter in ams_connect indicates the use of the default port on the server. The result of the previous command is a string containing the AMS Communicator name, hostname, and port number. To connect to the AMS Communicator, type the following command:

» comm = ams_comm_attach('simple')
comm =
0
»

This will attach the AMS Communicator *simple* and return an id *comm*. To get the memory list, use the command:

» memlist = ams_comm_get_memory_list(comm)
memlist =

```
simple_memory
»
```

The result indicates one memory called *simple_memory.* The use can then attach to this memory and get the list of fields using:

```
» [memory,step] = ams_memory_attach(comm,memlist)
memory =
     1
step =
    52972865
»
```

This function returns two outputs, a memory id, and a step number. The step number is a representation (modulo an unsigned int) the number of times this memory has been accessed for writing. In our *simple* example, the server is executing a loop and changing fields in the memory every time. That is why the number is too big. To get the fields list within the memory use the command:

```
» fieldlist = ams_memory_get_field_list(memory)
fieldlist =
int_elem
flt_elem
float_arr
»
```

The command shows that there are three fields in this memory: *int_elem, flt_elem,* and *float_arr.* To access data within a specific field use the command:

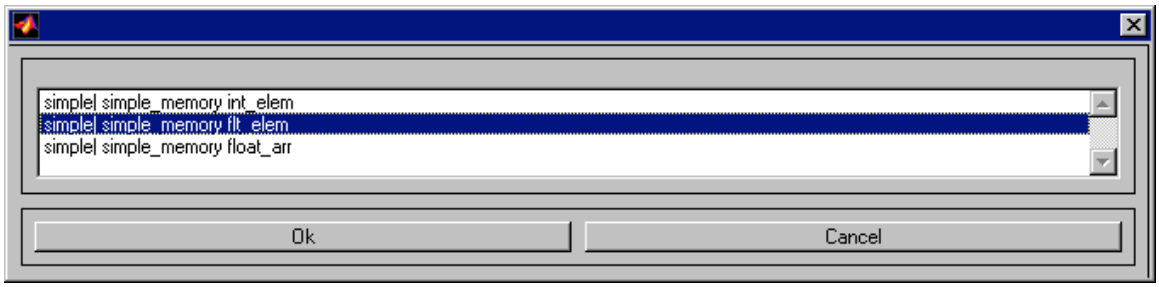
```
» data = ams_memory_get_field_info(memory, 'flt_elem')
data =
    2.5000
»
```

Accessing the data with one API call

The API call *ams_view_select(host, port)* should be used to list all the fields that have been published, select a field, and display its value. For example, the user can type the command:

```
» data = ams_view_select('tiamat',-1)
```

The following Matlab window will display the list of published fields:



The use can then select a field to view and Matlab will return the data in the field:

data =
2.5000
»

Using Matlab interpreter interact with the data

Two Matlab classes are provided to allow easy interaction and scripting of AMS published data. These are ams_comm_class and ams_memory_class located under the matlab directory within AMS.

For example, to connect to an application running on *yavin* using the port *9000* type of following command:

» comm = ams_connect('yavin', 9000)

comm =

multi_dims|yavin.mcs.anl.gov|4483

Once you have a handle to the AMS Communicator, we can use it to create an ams_*comm*_class:

» com_class = ams_comm_class(comm)

memories =

memory

The results indicate the *com_class* has on memory, called *memory*. Now the user can simply reference that memory to obtain the list of fields:

» flds = com_class.memory

fields =

flt_elem

array_1d
array_2d
array_3d

In this particular example, there are 4 fields. If you want to view the content of a field, we just reference that field as follow:

» x = flds.flt_elem

x =

2.5000

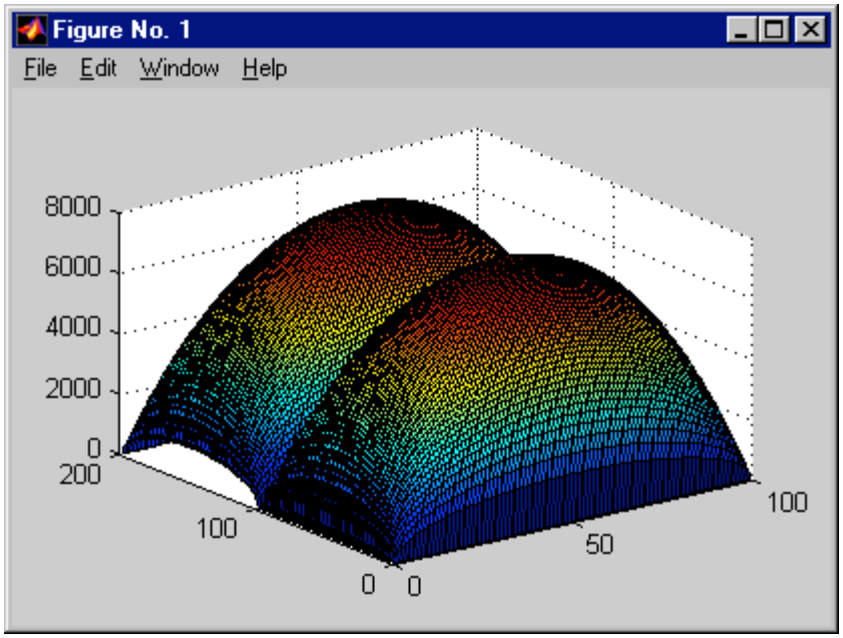
Similarly, we can modify the content of the field by:

» flds.flt_elem = 3;

This will automatically modify the value of that field on the running application (Powerful, isn't it?).

In the same manner, one can visualize the content of arrays. For example, to plot the content of *array_2d*, we use:

» surf(flds.array_2d);
»



The AMS Matlab interface will automatically detect the array dimensions and provide.

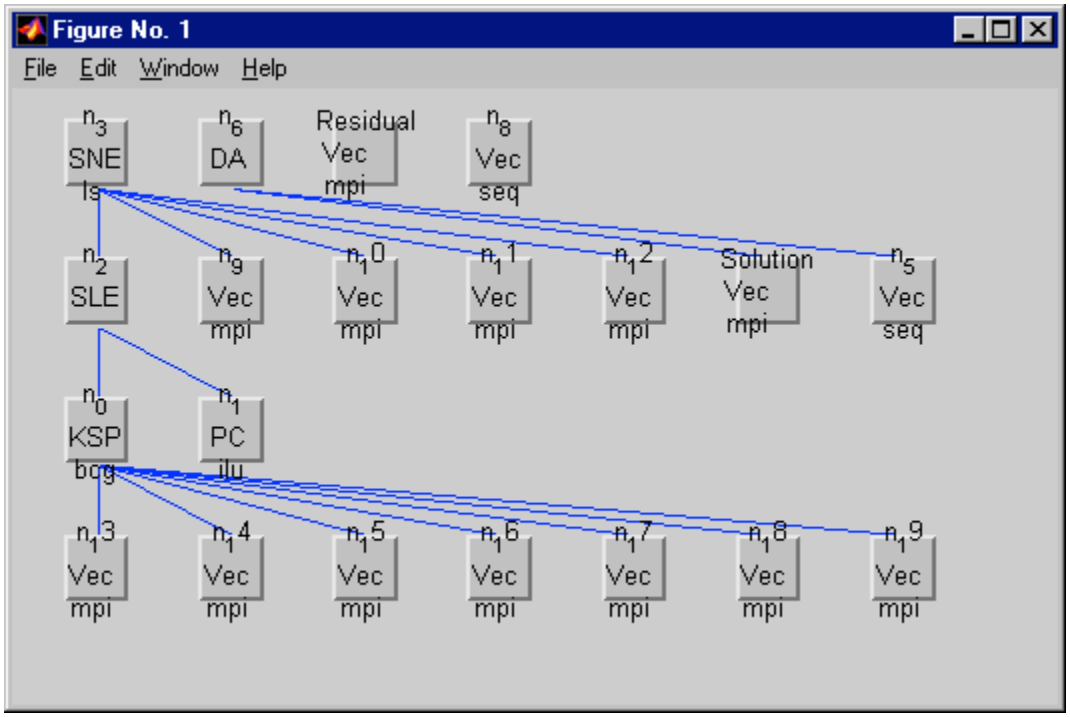
All the above commands can written in a script to create some sort of animation or custom Matlab script to process the data.

Using Matlab interface to build a custom client Accessor

The following is an example on how the user can build on the AMS Matlab API a more complex monitoring system. The application running on the server is a PETSc[2] Euler code, which we will assume that it is running on the host *tiamat*. It publishes various types of objects (solutions, vectors, iterations, residual, etc…). The Matlab client is located in the matlab/petsc directory. From the Matlab prompt, the user can type the following command:

» petscview('tiamat',-1)
»

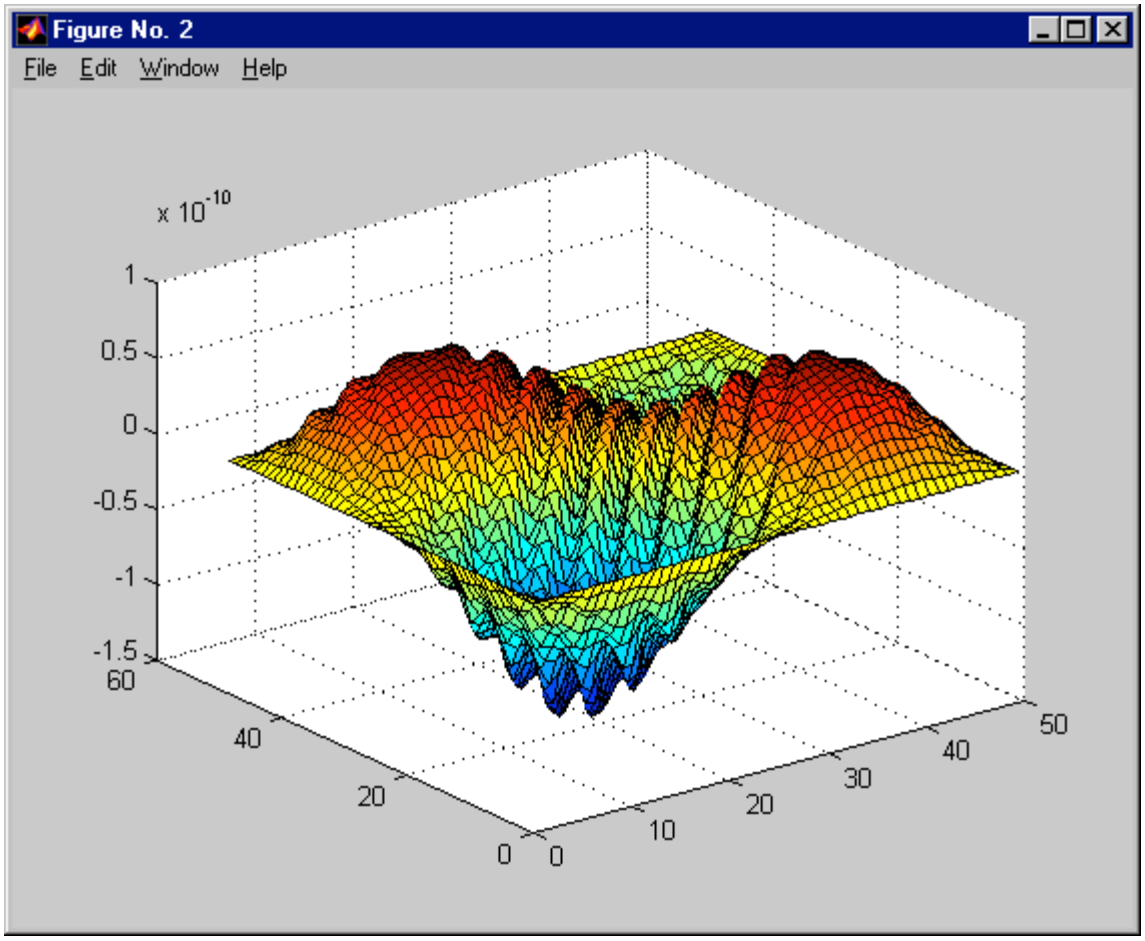
The following window will appear



This graph is built from information published by the server. The relationship among the objects is also constructed by the AMS API from the information that is published by the server. While the application is running on the server, the user can click on any button to

[2] http://www.mcs.anl.gov/petsc

get the data or display a solution. For example, clicking on "$n_1$7 Vec mpi" the following Matlab graph is displayed:



The AMS API will recognize weather an object is for example 2d, or 3d array and display it accordingly. If also the object has only one element, its value is displayed in the Matlab command window.