Note 186: The Tasking Environment of AIPS++

B.E. Glendenning
bglenden@nrao.edu

December 22, 1995

Contents

1 Requirements

The tasking environment of AIPS++ is used to construct the overall runtime system that astronomers see. It is also must provide various language-independent

high-level services which are used both by applications programmers and astronomers who wish to perform some ad-hoc calculations on their data.

These requirements have been culled from the original AIPS++ specification, and from our experience building various prototype applications. In abbreviated form, these requirements are:

User Interface Both a GUI interface and a powerful command line interpreter(CLI) should be provided. It should be possible to replace the user interface.

User-callable toolkit A set of useful tools (computation, plotting, ... should be available to users (both from the CLI and from other languages — FORTRAN, C, C++).

Access to data The user should be able to directly manipulate AIPS++ data structures, *e.g.*, through the CLI. The user should be able to manage data through normal operating-system utilities.

Tasks — convenient encapsulations of algorithms — should be available. It should be possible to execute tasks directly from the operating system.

- Task parameters should have context-dependent default values.
- Parameters should be local by default, although it should be possible to copy parameters from one task to another.
- The user must be able to save, copy, and edit parameters.
- It should be possible to change some parameters for some running tasks.
- A mechanism to support advising the user of unusual parameters should be provided.
- An "executable" processing log should be maintained for all running tasks.

Help Context-sensitive help should be available online at all levels of the system.

Packages The user should be able to customize what packages are to be used.

Nearly real-time The system should be usable in a nearly real-time environment, i.e. at the telescope.

Network capable The system should be capable of distributing computation, displays, *etc.*, over the network.

Data "catalogs" Flexible mechanisms to organize data under the control of the user must be provided.

2 Overview

This document defines an environment which orchestrates the creation and intercommunication of various software entities. These software entities include *application objects*, which are packagings of (presumably) useful functionality with a standard interface that are directly controlled by end users. Besides the application objects, the environment provides access to an extensible set of other *distributed objects*. These objects might be used by an application programmer, or they might be used directly by an end user to perform some ad-hoc calculation (by manipulating the objects through a command-line interpreter (CLI)).

All distributed object operations are managed by a *control hub*. The control hub is responsible for activating and deactivating the distributed objects, and for mediating communications amongst its objects. It also provides run-time access to a simple database, used for information like absolute paths to data directories. The control hub itself appears as a special object in the system.

The rest of this paper fills in the details:

1. Describe how object interfaces are defined, and their execution semantics.
2. Describe the design of the control hub.
3. Propose an initial set of standard object services to implement a canonical user environment.
4. Proposes a small set of further objects that are of short-term utility for programmers.

It is worthwhile noting that the computer industry is in the process of creating various distributed object systems as well, although it is not yet clear when these systems will be widely available in our Unix marketplace, or even what system (CORBA/SOM vs. OLE/COM) will be the eventual winner. It is certain, however, that such a system will be widely available within a small number of years. When this has happened, it is intended that the system described in the paper should (gradually) be replaced with that system.

Note that the classes in this document should be considered to be illustrative, not complete.

3 Distributed Objects

3.1 Object Interface

The clients of a distributed object may only manipulate the object through its interface. To express the interface, we use a subset of C++ and the AIPS++ class library. It must be emphasized that the objects can be manipulated from

languages other than C++, and can be implemented in languages other than C++.

An alternative strategy would be to describe the interfaces other than C++, such as the language neutral "IDL." Using C++ however has a couple of advantages:

1. Many people already know it, so it makes for one less language to learn.
2. AIPS++ has developed documentation tools which can parse C++ header files.

The following rules are to be followed when defining a distributed object. While these rules may appear restrictive to the C++ programmer, they are chosen to reduce difficulties in providing a bindings to other languages.[1], or to alleviate confusion for end users (case should not be considered).

- An interface is described by a pure-abstract base class. (All member functions are virtual, with no defined implementation).
- No global functions or overloaded symbols ("operator+"), are to be used.
- Only single inheritance is supported.
- All member function names (including those which are inherited) must be unique and case-insensitive.[2]
- Parameters to functions must be one of the following types:

  Scalar Char, uChar, Short, uShort, Int, uInt, Float, Double, Complex, DComplex, and String. Scalars may be passed by value or by reference.

  Index An index is an integral type which is used to specify a position in a data structure. Its value is adjusted by the run-time system to be zero or one-relative, depending on what the native language requires An Index should be pased by value or reference, an IPosition by reference or const reference. (The canonical representation of an Index is 1-relative).

  Record The Record class or a *known structure* may be passed by reference or const reference, and returned by value. "A known strcuture" is a pure aggregation of scalar, array, and record types, for example:

```
struct real_grid {
    Array<Float> data;
    Vector<Float> origin, increments;
};
```

[1] Such as FORTRAN and Glish

[2] Underscores should be used for multi-word variable names.

Only predefined structures may be used, they may not be created at will. Note that a particular language binding might just use a general Record type rather than providing all the structure types.

ObjectID An ObjectID is a unique identifier which is used by the run-time system to uniquely identify a particular object. The ObjectID is opaque, however it may be mapped int, and retrieved out of, a String. An ObjectID is either valid or invalid. ObjectID's may be compared for equality.

ObjectType An object type is an opaque type which is interconvertible with strings. The only operations which are available are:

1. Exact comparison of type.
2. Determining if one type is a direct or indirect parent of the other.

The ObjectType of any valid ObjectID may be obtained through the language binding.

Array Multi-dimensional arrays of scalar type (including Index) are allowed. Only Arrays of dimension one may be used for types Record, ObjectID, and ObjectType. Array, Vector, Matrix and Cube of the above scalar types. Vector, Matrix, or Cube may be substituted for Array to indicate the required dimensionality. Arrays must be passed by reference or const reference, and returned by value.

Object Reference An `ObjectRef<Class>` may be passed as an argument to a function by reference, const reference, or by value. Passing the object by value indicates that the object might make use of the object at a later time. Passing it by reference indicates that only temporary use of the Object is being made. An ObjectReference is either valid, or it is null.

The run-time system is responsible for canonicalizing the data types (so, for example, integers are converted between little-endian and big-endian systems, and for Index values are modified to be appropriate to a given language binding).

- The return type of a function can either be "void", or it can return by value any of the types which are allowed as function parameters.

3.2 Object References

After an object reference is obtained by a client process, it's member functions ("services") may be invoked by the client process.

When an object reference is no longer needed it should be *released*[3]. An object reference can be queried to determine whether or not it is valid (attached to an object server).

[3]In C++, this is done implicitly by the destructor

Most language bindings will provide two methods of invoking the objects member functions.

Stub Interface This is the interface which is most "natural" for any particular language binding. For example, in C++ it corresponds to invoking the member functions of the class.

Dynamic Invocation Interface [4] With this interface an argument list is marshalled, a function is named, and a request for service is sent to the object server. While this interface is generally less convenient it has two advantages over the usual stub interface:

1. It allows the services of any object to be invokes, not only those which have been "compiled into" the client.
2. Typically the stub interface will be synchronous, the DII will usually have an asynchronous mode of operation.

The Dynamic Invocation Interface might often be made available through an "any" object.

An execution of a service from an object may return normally, or it might result in an exception. An exception might be thrown by the object if it is being used improperly, or it might be thrown by the run-time system to signal a failure (for example, an object server crash). The exception mechanism is part of the language binding. To facilitate this binding, the possible exceptions are limited to the following:

ConnectionError A communications error, probably either a network outage or a crash of a process.

HubError An error in the internal workings of the control hub (a hub crash would result in a ConnectionError).

ArgumentError An invalid argument (type or value) was given to an object implementation.

ObjectError All other errors from the object implementation.

Each exception has three strings associaged with it:

1. An informative error message.
2. A string which describes the originator of the exception.
3. A (possibly blank) relative URL (described below in section xxxx) which contains additional help information.

[4] This is the CORBA term.

3.3 Object Base Class

All object classes must be derived from a base class (`distributed_object`). This is done to ensure that all objects have a common set of services, primarily to allow the objects to function well in a GUI environment.

```
class distributed_object {
public:
    Record help() const;                        // 1
    Record interface() const;                   // 2
    Bool save_state(String file_name);          // 3
    Bool restore_state(String file_name);       // 4
};
```

The member functions fulfill the following purposes:

1. Distributed objects may be directly manipulated by end users, so it is important that a uniform method of obtaining help is available. The exact mapping to the Record is still undetermined, but generally speaking it will provide both per-class and per-method documentation.
2. This member is provided so that a GUI control for any distributed object can be provided. Again, the exact mapping is not decided, but there will be an entry per member function (inclduing arguments and return types).
3. This member is provided to implement a simple form of persistence and so that the user can save the state of his environment. AIPS++ core classes are expected to save their state into AIPS++ table objects.
4. This is used by the object to restore its state.

4 Control Hub

The control hub implements the functionality required to support distributed objects and the AIPS++ environment. The actual functionality available to the user is provided by standard service objects, which are described later.

The control-hub itself appears to be an object.

4.1 Environment

The environment is a simple hierarchical collection of "keyword=value" strings, similar in spirit to the X-window system resource files. It is implemented on top of the "AIPS++ Resource Database."[5]

These resources are maintained in text-only initialization files, and they are used to maintain information related to bootstrapping (such as paths to data directories) and user preferences.

[5]Described in `http://info.cv.nrao.edu:85/info/System,aipsrc`.

4.2 Object references

An attached object reference may be associated either with a *local* object or a *distributed object*. Local objects exist in the address space of the object reference and member functions are invoked directly. Member functions of distributed objects are invoked via the control hub. Local objects are preferred when the communication overhead is important.[6] It generally does not make sense to create objects which can only be invoked locally – in this case it is more convenient to create native-language library entities.

4.2.1 Type and Object Servers

Objects exist in some server object. The most common use will be to create a transient instance of some particular ObjectType ("FFT server"). It is also possible to directly request a particular object instance, which might or might not be active. This is the method by which persistence is implemented in AIPS++.

An process can thus be regarded as a repository of objects. It can either create the objects, activate them from persistent state (*i.e.*, disk files), or both.

Note that a given binary executable can result in more than one object server being active at one time, since the same executable may have more than one process associated with it.

An object server itself appears to be an object.

4.2.2 Object Creation and Method Invocation

All objects, except the control hub pseudo object, reside in a process, including the object server objects from which all other objects are ultimately obtained. The object server for each process is special in that it is implemented by the run-time-system, not the object programmer.

Requests to create or restore an object are passed through the object server into callback functions registered with it. A list of active and persistent objects is maintained by the object server.

If the creator and user of an ObjectID are in the same process, then the ObjectID can resolve into a local reference to the object with some language bindings (*e.g.*, in C++ a pointer to the actual object will be used behind the scenes). Method invocations for local objects use the native language invocation mechanisms.

Distributed object invocations proceed as follows:

1. The method is invoked on the object reference.

[6] Distributed objects may allow for faster computation though – for example if the object is running on a faster computer, or if there are multiple distributed objects running in parallel on a multi-processor computer.

2. The function arguments, method name, and ObjectID are bundled up into a record, and passed to the dynamic invocation interface. The caller blocks until the DII returns. (If asynchronous behaviour is desired, the DII should be used directly).

3. The method and its arguments are passed to the control hub.

4. If the object server is single-threaded and already processing a request, the new request is queued until the server is available. Otherwise the request is sent on to the object server.

5. The object server server calls a dispatch method in the implementation object. The dispatch method demultiplexes the function arguments, and calls the desired object method.

6. The dispatch mechanism bundles up the return result and modified parameters, or thrown exception, and sends them back to the calling object.

7. The function return is handled much like the function invocation.

8. In the event of an error in the system, an exception is returned from the point "closest" to the error:

   (a) From the control hub if the object server crashes.

   (b) From the calling processes run-time-system if the control hub crashes.

4.3 Packages

A package is the organizing unit for collections of programs and associated documentation and system data files. Packages must be "registered" with the control hub, since all objects must ultimately be found and manipulated through the control hub. An initial package list is obtained from the environment.

5 Standard Environment

The control hub and related structures are implementation details — they impose no structure for the user to operate in. This section describes the standard environment that the user operates in.

The system described here has the following elements:

1. An object chooser for browsing data objects

2. Various type-specific browsers (*e.g.,* aipsview, table browser) which can be launched from the object browser.

3. A task-launcher for objects and tasks which act on data.

4. A CLI for ad-hoc calculations, high-level programming, and command-line control of objects (the CLI must be bound to the object system). Note that the CLI is a separate process from the control hub, and it need not be the same language.

5. Log and progress monitors.

6. A context-sensitive[7] help and online-documentation system.

7. Various objects to allow CLI-users and other programmers access to data and computations that it would be inconvenient to reproduce in the native language.

8. A workspace in which the user may conveniently group data and task objects.

9. A parameter database which acts both to save user defaults for parameters, and to act as a clipboard for selections (e.g. an Image BLC,TRC).

In brief, I would describe this model as data-centric and moderately coupled.

Note that GUI related issues are outside the scope of this paper, although they will be important for the adoption of this system.

5.1 AIPS++ Browser

The Object Browser is used to find data and tasks (although to the user they appear in different GUI's). Thereafter it is used to activate operations on the data, and control for the tasks.

```
class aips_browser {
public:
    Bool add_package(String package_name);
    Bool remove_package(String package_name);
    Vector<package_description> active_packages() const;

    void current_data_directory(String &alias, String &path) const;
    Bool change_data_directory_to_path(String path);
    Bool change_data_directory_to_alias(String alias);
    void rescan_directory();

    void show_all();
    void set_name_filter(String show_it_match_this_pattern);
    void set_type_filter(Vector<ObjectType> hide_these_types);
```

[7] That is, the help system can be directed by other software entities to the correct entry in the documentation system.

```
    Vector<ObjectID> select_objects();
    ObjectID launcb_task_control();

    set_viewer(ObjectType type, String server_executable);

    // e.g. so you can follow one set of directories with a different browser
    ObjectID clone() const;
};
```

5.2 Tasks

Computational applications which only require modest levels of interaction are called tasks for historical reasons. The model here has the following features.

1. Tasks parameters are saved in a database, and can be context sensitive ("in VLB mode, you have to set these parameters as well"). Task parameters are inherently local, although defaults may be copied from database if desired.

2. Tasks which require it can have (or require) interactive updates to some parameters.

3. Tasks may be rerun with from an initial set of parameters. A task writer may allow a task in which parameters are updated during execution to be rerun as well.

4. Tasks produce log messages, progress events, and can provide context sensitive help.

5. Tasks can be started from the Host OS shell.

5.2.1 Parameters

Parameters are named values required by a task to execute. The values may be of type scalar, array, enumeration, parameter set (hence they are hierarchical), ObjectID, or decision. A decision type is an enumeration that is used to select further parameters. Parameters, as well as their allowed and default values, are mapped into a Record.

Besides a name, each individual parameter might have a parameter type. These types are common to all tasks, and are analogous to the known structures, however they need not be of record type (e.g., it could be an "alias" to a String"). These types are used by the parameter database, and are useful for documentation (a "file name" instead of just "string"). Each parameter type has a help pointer.

A task has one parameter set which is used to initiate execution. Ongoing execution of the task is controlled by parameters as well. Those parameters

will usually be different (more restricted) than the initial parameters. The parameters are fundamentally obtained from the task at run-time, although they can be cached externally for efficiency and browsing purposes.

Parameters may be stored in a parameter database "by task" - with multiple named versions. Defaults for particular parameter types may also be stored.

For an arbitrary object, the "type" of the task parameter will be its name. This implies that parameter names should be chosen with some care; in particular they should be the same across classes when their purpose is the same.

5.2.2 Task Activation and Control

Task activation follows a particularly simple protocol:

```
class task {
public:
    Bool go(ObjectRef<task_controller> controller, Bool block);
    Bool shutdown();
};
```

The task controller, which is queried by the task, is a more interesting class:

```
class task_controller {
public:
    // Common services and information that may be of use to the task
    Int                          desired_log_level() const;
    ObjectRef<log_sink>          log_displayer();
    ObjectRef<help_viewer>       help_displayer();
    ObjectRef<progress_monitor>  progress_displayer();
    // These are object-ID's rather than references to prevent unnecessary
    // linking
    ObjectID image_displayer_id();
    ObjectID plot_displayer_id();
    ObjectID table_displayer_id();

    Int demand_parameters(const Record &paraform, Record &set_parameters);
    void set_update_paraform(const Record &paraform);
    Bool have_update() const;
    Record get_update() const;

    Bool have_replay() const;
    void replay(Vector<Record> &stored_parameters,
                Vector<Int> &sequence_points) const;

    // Use for hints, like memory use
    Record environment() const;
```

```
};
```

Note that this protocol allows the ability for a task to "replay" itself, even if task parameters are modified while the task is running. This is implemented by the task emitting a "sequence number" whenever its parameters are changed (these numbers need not be consecutive – they will often be associated with an algorithm parameter, *e.g.*, iteration number). When a replay is desired, the sequnce numbers are provided along with the sequence of stored parameters.

5.2.3 Standalone Tasks

It is often useful to be able to run "load and go" tasks from the host operating systems command line, and not requring a running control hub, etc. This is accomplished by the run-tim-system providing a local task controller which fills in parameters from command line arguments, and returns local objects which print to standard output for help, logging, and progress monitoring. The other object services, like image display, are null.

5.3 Service Objects

These are objects which are provided for the user to manipulate directly, often from the command-line interpreter. They fall into the following categories:

Data exploration These objects will be typically be attached to a GUI. They are not only used to view the data, but to interactively get selected parts of it (*e.g.*, a table column, or a subset of points to use for continuum subtraction), and possibly to change values as well. Initially we will require objects for tables, 1D graphics, and 2D+ graphics.

Computation and algorithms The computations and algorithms which are available to the C++ programmer should also be made available to the CLI user through computational objects. Normally the same executable would serve both the application and its unique computations.

Persistent data The user must be able to manipulate persistent data. At a minimum, this means he must be able to read and write tables, since by edict all AIPS++ data, with the exception of user initialization tables, is accessible through the table interface. An additional interface for the Image will probably be required. Note that system databases (e.g. leap seconds) are merely table objects that have a known name in a standard directory.

It must be emphasized that these services are being provided directly for the end user. They should generally posess fairly simple interfaces. I would suggest that the interfaces should be approved by potential users.

6 Some Development Plan Items

This plan lists the high-level work items which need to be assigned to implement the design described here. Assignment of tasks and target dates will be recorded elsewhere. The items are listed in approximately the order I believe implementation should proceed.

1. Control hub implementation in Glish.
2. Implement C++ binding and C++ RTS along with some sample object servers.
3. `.aipsrc` parsing in Glish/C++. [C++ is done]
4. Package manipulation and registration (includes command line argument conventions to executables).
5. Define Glish language binding (the Glish binding may drive enhancements in the Glish language). After, implement distributed object RTS for Glish
6. Determine method of documenting both pure Glish functions, and especially inter-language classes (a FORTRAN programmer won't want to see a C++ class).
7. Non-GUI aips browser implementation.
8. Non-GUI task controller (including parameter/Record conventions).
9. Logging, Help, Plot, Image display, Table editor interfaces and non-GUI implementations.
10. Implement parameter database (requires tables to be able to store Record/keyword sets in a cell).
11. Sample tasks.
12. GUI interfaces for standard user services.
13. C++, Glish, and FORTRAN "compilers". My feeling is that we can do two languages "by hand", but when we want to go to a third we will need automation.