AIPS/AIPS++ INTEROPERABILITY

A. J. Kemball

August 29, 1997

File: /aips++/210.latex

Abstract: This note discusses the question of interoperability between AIPS and AIPS++ during the transition period between the two systems. The objectives of such an effort and the technical means by which it might be achieved are considered. A distributed object design, which has already been investigated as a prototype, is presented as a solution.

1 Introduction

Some measure of interoperability between AIPS and AIPS++ during the transition period between the two systems holds clear advantages for both NRAO and the user community. This note examines this question in more detail in order to identify which forms of interoperability are likely to be scientifically useful, and the technical means by which they could be achieved.

2 Global objectives

The term interoperability is used here to imply the availability of means to access AIPS data, algorithms or the command line interface (POPS) from within AIPS++ applications or the AIPS++ command line environment (Glish). A complete definition of interoperability in this context however requires a clear specification of the extent to which such access will be enabled. This needs to be determined by weighing the scientific advantages of such an effort against the cost of implementation. The benefits of interoperability during the transition include:

- **Simplified user environment:** The availability of some AIPS functionality from within the AIPS++ command line interface (CLI), such as the capacity to launch AIPS tasks acting on AIPS data, will provide a simplified interactive interface during the transition period, which has certain benefits for scientific productivity.
- **Improved bridging utilities:** Targeted access to the AIPS data representation and algorithms from within AIPS++ applications will significantly simplify the development of bridging utilities for use during the transition period. These may include data conversion tasks or simple calibration conversion utilities to translate certain forms of AIPS data to the equivalent AIPS++ representation. This will have certain restrictions due to the different data representations but should nonetheless be useful in allowing smoother transition points from one package to the other in the data reduction sequence.

- **Coordinated development:** Interoperability will improve the likelihood that development priorities can be coordinated between the two packages.
- **Improved programmability:** Access to certain types of AIPS methods and data from the AIPS++ CLI will improve the programmability available to the users and provide scientific incentives to use AIPS++. There are clear scientific benefits to being able to access and manipulate both AIPS and AIPS++ data in the AIPS++ CLI.
- **Code re-use and testing:** Interoperability has some internal uses within the AIPS++ project in allowing re-use of well-established AIPS code in inter-comparison tests between the two systems. This will speed up development and testing of new AIPS++ applications and provide some important information on reliability and accuracy to the user community.

The disadvantages or pitfalls to be considered in implementing interoperability are:

- **Cost:** The resources required to implement interoperability need to be considered carefully in deciding how far to take this process. The development of native AIPS++ applications is a high priority.
- **Design implications:** Interoperability should not impact the design or functioning of AIPS or AIPS++ when considered as separate stand-alone packages.

3 Interoperability options

As discussed in the previous section, a significant consideration in this matter is the exact degree of interoperability that is envisaged between the two systems and the manner by which this is achieved. Specific options that might be made available to an AIPS++ user or programmer include the following capabilities:

- **Enhancement of and access to the AIPS CLI:** This includes the ability to launch standard AIPS tasks acting on AIPS data from the AIPS++ CLI, and the ready implementation of an AIPS graphical user interface (GUI) in Glish/Tk.
- **Invoke AIPS components from Glish:** This would involve the provision of specialized AIPS components to serve selected AIPS data or algorithms to the AIPS++ command line environment. These would most readily be implemented as Glish distributed objects (DO). This option is most closely related to improved user programmability. The AIPS and AIPS++ data could also be passed to other command line environments including IDL or Mathematica.
- **Provide a C++ interface to the AIPS libraries:** In this approach, selected AIPS libraries could be provided with a C++ calling interface, as has been pursued with other external libraries. This would allow an AIPS++ programmer to re-use some AIPS algorithms with a direct binding.
- **Recognition of the AIPS data format:** AIPS++ applications could be made aware of the AIPS data format and be able to act on AIPS data files directly.

- **Documentation:** The ease with which users will be able to navigate the AIPS/AIPS++ transition will depend on the quality of documentation describing the available interoperability options. The transition points between the two systems are likely to change dramatically on short time scales during this period.

Several of these options have disadvantages and are not emphasized in the design as a result. These include the possibility of recognizing AIPS data from within AIPS++ applications and the provision of a direct C++ binding to the standard AIPS libraries. Both significantly influence the design of the two packages considered as separate stand-alone entities, would be expensive to implement and are impractical due to the mismatch in data and function representation. In particular, a direct C++ binding to the AIPS libraries is difficult due to the lack of a clear external interface to many of the routines and the limited encapsulation of global data. It would also require close synchronization with continuing evolution in the AIPS libraries and would therefore significantly influence the development of each system considered as a stand-alone entity, thus violating the design objectives given above. A direct binding may be possible in some cases however and is not excluded.

Likewise, the recognition of AIPS data from within AIPS++ applications is ruled out because of the significant overhead this would place on the development of native applications. This is fundamentally due to the mismatch in data representation and the implied requirement of replicating significant parts of the AIPS infrastructure within the AIPS++ libraries.

A design that provides access to AIPS data and algorithms, while avoiding the disadvantages listed here, is given in the next section.

4 Technical solutions

This section describes a design that has already been implemented as a prototype to meet the objectives discussed above. A technical solution to the question of AIPS/AIPS++ interoperability is best approached through the formalism of distributed objects, a key element of the global AIPS++ design. Distributed object methodology has many uses in developing client-server applications in heterogeneous network environments using object-oriented software design methods. In this model an object request broker (ORB) is available to locate and invoke methods on a collection of distributed software components or objects. This allows the integration and re-use of multiple software components of unknown implementation or network location to form coherent integrated software applications.

This methodology is also commonly used to allow the co-existence of legacy code with modern distributed object systems. In this case a dedicated object adaptor presents an object-based interface to the rest of the system for the legacy code components, defining an arbitrary high-level set of methods or accessors as required by the global design. Private data and other implementation specifics are left undisturbed in the legacy code component as far as possible.

The design adopted here for AIPS/AIPS++ interoperability is related closely to this approach, and represents a limited implementation of a true distributed object system. The criteria listed in the preceding sections can be met using a global design shown schematically in Fig. 1.

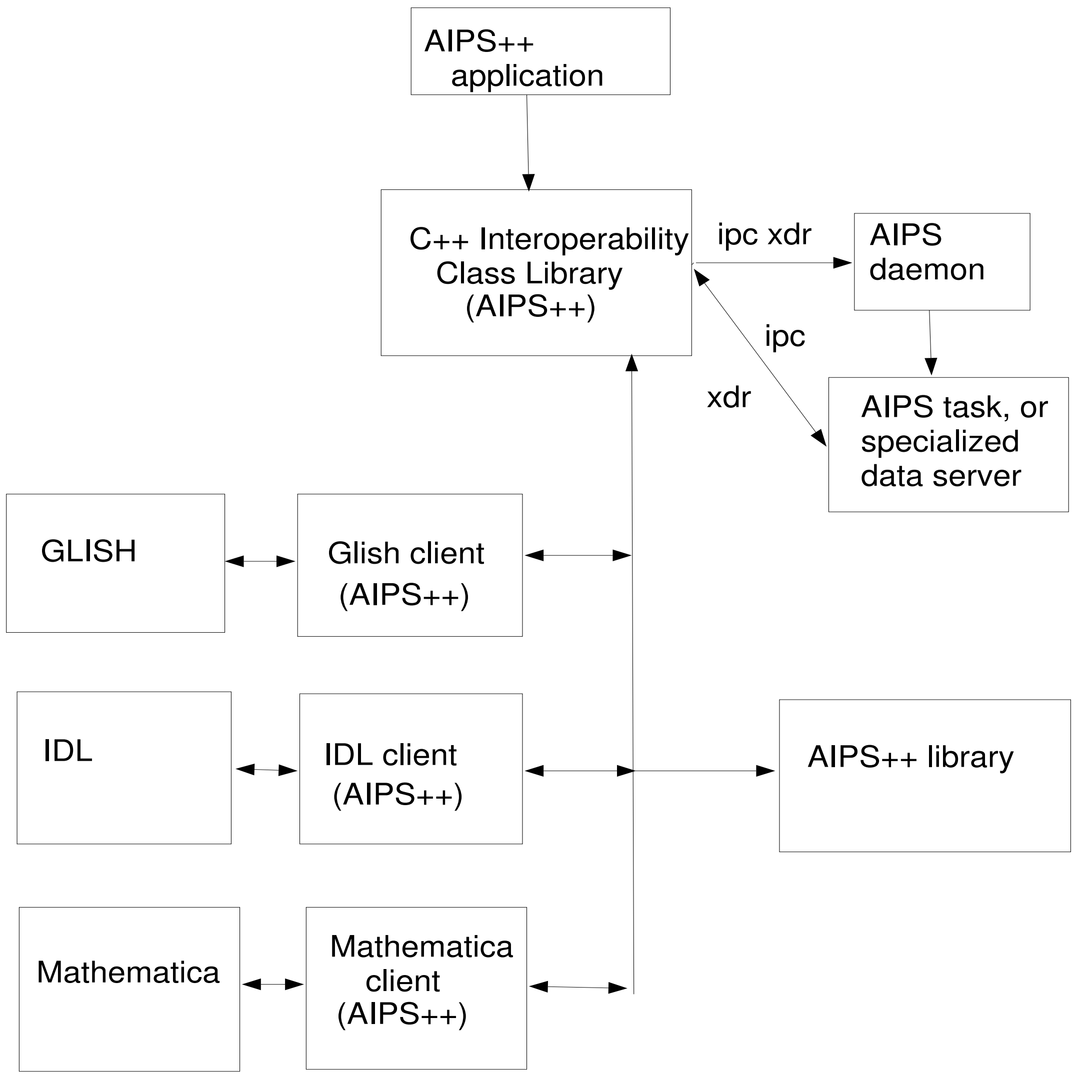

Figure 1:

Schematic diagram of the proposed design for AIPS/AIPS++ interoperability. Several aspects of this design have already been demonstrated in a working prototype.

In this design a central component is a set of AIPS++ classes defining an active object interface to the AIPS system. These classes communicate with an AIPS daemon that provides access to the command line interpreter, and with specialized AIPS servers using an event messaging system. The message system is based on standard internet protocol messaging (IPC), with XDR encoding for network independence and a simple request-reply model.

The AIPS daemon is a specialized version of the POPS interpreter AIPS.FOR modified to communicate with the C++ classes rather than accepting keyboard POPS input. The content of events recognized by this daemon, called DAIP.FOR, constitute standard ASCII strings containing POPS commands; in most other respects there is little difference from the standard interactive interface. The daemon can launch standard AIPS tasks if it receives a POPS **GO** command. This allows, for example, the implementation of a simple GUI interface for AIPS from Glish.

In addition, the AIPS daemon can be used to launch specialized AIPS servers for passing image, uv or table data from AIPS to the active C++ classes and vice versa. The AIPS servers are tasks in the standard AIPS format that implement event communication with the C++ classes using a subset of IPC and XDR routines in the AIPS library. The AIPS servers and daemon can be implemented as part of standard AIPS and do not impact the AIPS distribution as a stand-alone package. They can however use the full AIPS library in a direct and standard fashion and reflect high-level functionality to the rest of the AIPS++ system at an arbitrary level of abstraction. The format of supported events defines the dedicated object interface to the C++ classes. Thus, this is a simple distributed object system with a hard-wired object adaptor and messaging system. The AIPS XDR event format consists of the following general fields: i) message length; ii) event name; iii) event version number, and; iv) multiple fields of the form (name, type, dimension, value), where type can accommodate most of the basic C++ data types.

An ancillary feature is the availability of a slightly modified form of the standard AIPS start-up script which starts only the AIPS TV, tape and Tekronix servers and creates a checkpoint file of global defined logicals which need to to known later to the process launching DAIP.FOR. The changes to the startup script do not compromise the functioning of standard AIPS. This startup script can be implemented as a global function in the C++ interoperability classes and invoked as needed.

The active C++ classes can be used directly by AIPS++ Glish clients to reflect the AIPS events to the Glish CLI for direct manipulation, or to other CLI environments such as IDL or Mathematica. The C++ classes can also be used directly by standard AIPS++ applications to create test and bridging utilities during the transition period without duplicating significant parts of the AIPS infrastructure in AIPS++. The C++ interoperability classes shown in Fig. 1 consist of the basic messaging classes to support AIPS events, and higher level classes that provide access to AIPS data structures and algorithms.

It is important to stress here that the primary user access to the AIPS data and methods will be at the AIPS++ CLI level and that the corresponding Glish closure objects need to present the same programming interface for both AIPS and AIPS++ data. For example, access to AIPS image data will only be supported for a subset of the methods available for AIPS++ data but the method and function names will be identical apart from the constructor. This will allow the direct re-use of the Glish scripts. This objective is best achieved by closely basing the AIPS C++ classes on their AIPS++ counterparts and enforcing the same naming conventions. Direct inheritance from the AIPS++ base classes may be possible in some cases but the

AIPS C++ classes and Glish proxy objects will be kept completely separate and will have no influence on the general AIPS++ class library development.

As an example, access to AIPS image data may be obtained through an *aipsimage* class, which supports a subset of the features of the standard AIPS++ *image* class, either with the same method and function names or through direct inheritance. The *aipsimage* class uses the event messaging classes for IPC/XDR communication with a specialized AIPS image server to access the AIPS image data. The *aipsimage* class is made available to Glish through the standard DO formalism, as part of the image server. This ensures a common programming and user interface.

In closing it is noted that parts of this design could be implemented using industry standard distributed object implementations, such as CORBA or OLE. However, the AIPS C++ interoperability classes could easily be incorporated in such an implementation if this were to be adopted within the AIPS++ project as a whole, as might be considered at some later time. The same consideration applies to the object adaptor to AIPS, but in this case the interface issues discussed in Section 3 would need to be resolved.

Note that the design discussed in this document could be used for other data reduction packages such as MIRIAD, as appropriate.

5 Conclusions

The design for AIPS/AIPS++ interoperability presented here meets the global objectives discussed at the start of this document, offering significant scientific advantages to the users of AIPS and AIPS++ during the transition period. It also enhances programmability available to the users and simplifies development planning. It is believed to avoid the identified disadvantages. Key elements of this design have already been implemented and shown to work in a prototype. A significant component of this implementation could be distributed as part of the AIPS++ beta release and the 15OCT97 AIPS release.