

An Open Architecture and Framework for Astronomical Data Processing and Analysis

D. Tody^{1,2}, P. Grosbol³, B. Garilli⁴, W. Cotton¹, P. Linde⁵, D. Ponz⁶, R. Hook³, K. Banse³, K. Reinsch⁷

¹*National Radio Astronomy Observatory*, ²*U.S. National Virtual Observatory*,
³*European Southern Observatory*, ⁴*Italian National Institute for Astrophysics*,
⁵*Lund Observatory*, ⁶*European Space Agency*, ⁷*Institute for Astrophysics*
Gottingen

Abstract. An open system architecture for astronomical data processing and analysis is described. Most of the functionality is in the form of reusable components which can be written in any major language. A execution framework based on a component-container architecture connects all components of the system, providing distributed execution and scalability. Applications connect together components via the execution framework, and are normally written in a high level language such as Python or Java. Integration with the virtual observatory (VO) is provided for both access to remote data and services, and for publishing new data and services to the VO. An open architecture allows elements of the system to be used individually or integrated with external software.

1. Introduction

Most astronomical data analysis software in use today is found in systems which are 10-25 years old (AIPS, AIPS++, IDL, IRAF, MIDAS, Starlink, etc.), and which are often no longer well supported or under active development. The technology used in these systems is outdated and does not make effective use of the wealth of software developed outside astronomy over the past decade or more. These older systems provide only limited interoperability and software sharing between systems. Their architecture was not designed for modern astronomical data processing which is often characterized by large complex datasets and distributed access to remote data and computation.

The Virtual Observatory (VO) addresses part of this problem, but VO is mainly about middleware and how widely distributed components talk to each other and interoperate. VO is less concerned with how scientific computation is actually performed, assuming that most computational software will come from the astronomical community. Our focus here is on what is required to provide this computational software, which we would like to integrate well with the VO while providing a high level environment suitable for scientific software development by general users.

In the past year we have developed an architecture, described herein, for a new system to address these issues. The project described here is a joint effort of the OPTI-CON network in Europe and the NSF-funded NVO project in the US, with participation from groups carrying out related prototyping efforts (e.g., ESO, NRAO).

2. Principal Use Cases

Our primary use cases are the following:

- **Desktop data processing and analysis system.** This is an astronomer's workbench for data analysis, providing both an integrated set of tools, as well as a scripting environment (e.g., Python, Java) for the user to develop their own software. Integration with VO (see VO-Client below) provides access to remote data and computation. Equivalent access is provided for both local and remote data.
- **Pipeline processing system.** For PI observing programs on modern telescopes it is often desirable for the astronomer to be able to manually reprocess and tweak their data, even though a standard data reduction pipeline may already be provided. The same software should be used in both cases. Execution may take place either remotely, or locally on the user system, hence it is necessary to be able to export the pipeline software to the user. For this to work well the desktop system and the pipeline system should be based on the same software.
- **Service framework for VO.** Production of virtual data products for VO data services is very similar computationally to pipeline processing; both operations produce similar astronomical data products. Hence it is desirable to be able to re-use the same software for both cases. Similarly, analysis services for the VO can be produced by wrapping application scripts and publishing them to the VO via a web portal. The resultant VO-enabled applications can be used in Grid workflow systems such as Astrogrid.

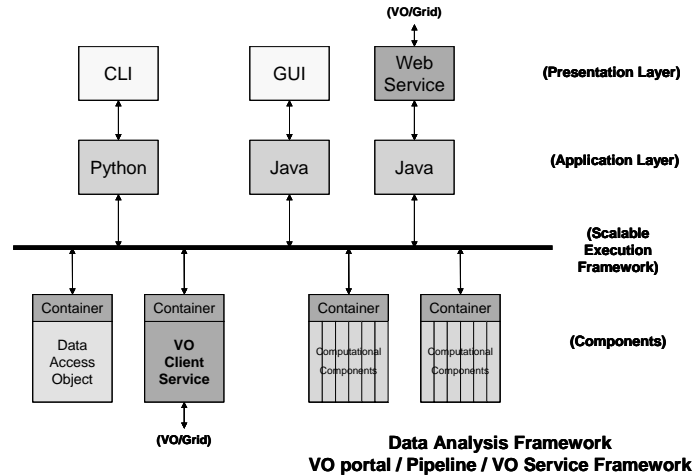
A common architecture for all these cases is desirable to enhance familiarity and allow re-use of software in multiple contexts.

3. Goals and Features

The system architecture we come up with should have the following characteristics:

- **Open architecture.** The major elements of the system should be replaceable modules with well defined interfaces, often usable stand-alone separately from the rest of the system. The technology used to implement a module should be largely hidden behind the interface. System integrators may mix and match components to build many kind of systems.
- **Multi-wavelength.** The system framework should be general enough to support all branches of astronomy (science software may however need to be tailored to a single branch of astronomy or instrument). System components or applications may come from many groups.
- **User programmable.** It should be possible for a user with a scientific rather than computer science background to develop applications with minimal training. The science software and applications should be isolated from the details of the system framework and underlying technologies used.
- **Scalable.** The system should be transparently scalable to run on anything from a laptop to a Beowulf cluster, or on the grid. A user should be able to transparently develop software on their desktop which can be later deployed to a remote cluster or supercomputer.

See the project TWiki page for more detailed information on the system requirements.



4. Architecture Overview

These goals, plus related requirements for multi-language support (C/C++, Python, Java), and support for legacy software (not entire systems but the useful bits that do actual data processing) lead us to a distributed component-framework architecture. In this approach, most astronomical software is cast into the form of re-usable components which can be deployed in various ways. The major system elements are as follows:

- **Presentation layer.** Presents the functionality of the system to whatever outside agent drives the system, be it a human user, a Grid workflow, a Web browser interface, or whatever.
- **Applications layer.** The applications layer is used to implement top level applications. The applications layer can be anything which can drive the execution framework to execute components, for example, Python, Java, a GUI, or a workflow engine of some sort.
- **Execution framework.** Provides the functionality needed to execute components, including capabilities such as component registration and management, distributed execution, scalability, messaging, logging, and so forth. A range of execution frameworks of varying degrees of capability are possible.
- **Container.** Components execute within a container which defines the life cycle and runtime environment seen by the component. The container is the interface between the execution framework and an individual component.
- **Components.** A component is a computational object, with one or more service methods, which can be plugged into the framework. Components are grouped into component packages of related components. Components provide most of the functionality of the system.
- **Task-Parameter Model.** A special case of a component is a task, with a single service method, using a parameter set to drive the functioning of the task. Output

from a task can also be returned as a parameter set, and passed on as input to another task.

5. VO Integration

The computational framework described here is intended primarily for high performance computing on a single computer (workstation or cluster). Integration with VO for wide-area (grid) computing occurs mainly in two areas:

- **Web service portal.** Any application in the system can be exposed to the VO as a web service via a web service portal. A conventional Web front-end (Tomcat/Axis, dotNet, etc.) interfaces to the Web and serves as the presentation layer to call computational applications. In general, special provision is needed to observe VO/Grid protocols, e.g., for authentication and file transfer.
- **VO-Client.** The VO-Client module implements the client side functionality required for a data analysis application to make use of the VO. This includes code to access the VO registry to find data and services, client-side code for the VO data access services for data discovery and retrieval, and an implementation of the VOStore/VOSpace mechanism for asynchronous staging of data. VO-Client is implemented as a daemon running within the data analysis system, with bindings to all supported languages. Object APIs will provide direct access to data objects (images, spectra, etc.), including both data objects retrieved from the VO, and local data.

6. Status

Specification of the requirements and high level architecture have been completed. Most of the major elements of the system have been prototyped, either directly or in related systems. Over the next year the system design will be further detailed and the first elements of the system will be implemented. See the project web pages^{1 2} for further details.

¹<http://archive.eso.org/opticon/twiki/bin/view/Main/WebHome>

²<http://www.us-vo.org>