

Future Astronomical Software Environments Design Concepts and Architecture

Introduction

Much astronomical data analysis software in use today is found in systems which are 10-25 years old. The technology used in these systems is outdated and does not make effective use of the wealth of software developed outside astronomy over the past decade or more.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 have a largely closed architecture with only limited interoperability and software sharing between systems. Their architecture was not designed for modern astronomical data processing which is characterized by very large data volumes and distributed access to remote data and computation (Grid computing).

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 and data analysis software will come from the astronomical community. Our main focus here is on this computational software which the astronomical community is expected to provide, which we would like to integrate well with the VO. The software required ranges from a desktop data analysis environment which can serve as a portal to the VO, to a computationally intensive server-side element such as a data access service or pipeline. A common architecture for all such software is desirable to allow software developed by the astronomer in the familiar local desktop environment to be dynamically deployed to a remote server, to enhance scalability and make it feasible to move the computation to the data.

The system concept we explore here is at the core a processing engine designed for scientific computation. Such computation can be complex and is often compute intensive, and may involve the execution of many components. The system we describe is tightly integrated and scalable, targeting primarily desktop and cluster processing within a local administrative domain. Grid computing is supported at a higher level by using the processing engine to build services which are linked together via the VO/Grid infrastructure.

To help develop the necessary software architecture it may help to look more carefully first at the problem domain. What types of users are we writing this software for? Who will use it? What types of problems do we expect the software to be used to solve?

User Perspectives

Various types of people would use the software we describe here (see also [UserScenarios](#)):

- **Astronomer.** The astronomer just wants to get their research done with a minimum of fuss. They want a ready-to-use system which is easy to use and which either

includes all the needed functionality or integrates well with standard tools. A range of functionality is needed, ranging from data reduction to data analysis, including tools for data visualization, data browsing, plotting, etc. Data management is important as modern data sets can be complex and large. Operations may combine both user data and data from public archives, and data and computation may be either local or remote.

- **Developer.** In this context, by developer we refer primarily to someone who develops science (computational or algorithmic) software, e.g., for data processing or data analysis. Such software is often complex, may be highly structured, and is focused entirely on complex processing or analysis of data. For reasons of longevity and maximal re-use, science software wants to be as system and technology independent as possible. The science software developer wants to focus on science data processing functionality, with as little concern as possible for the details of the system framework or runtime environment within which the software will execute. It should be possible for a science software developer to work effectively without having to learn much about complex execution frameworks or system infrastructure.
- **System Integrator.** The system integrator is anyone who builds an end-to-end, integrated system to give to users, or for use as a facility system at some site. The system integrator needs to decide what will be in their system and what the resultant system will look like to a user. While the system integrator wants to be in charge, they do not want to have to develop all the software from scratch. The system integrator wants re-usable software which can be flexibly integrated to build a wide variety of systems. The system integrator needs to control software versions, needs software available in source form to be able to diagnose and fix problems, and needs to be able to integrate and redistribute any software used. Examples of the types of systems a system integrator might produce include a desktop system for end-user data analysis, a data reduction pipeline for an instrument or survey, or an archive front-end for interfacing an archive to the VO.

An analogous case of system integration is the **Linux** operating system. Multiple variants of Linux can be produced from the same underlying framework (the Linux kernel) and "components" (various open source packages and libraries). Aside from being higher level, our case differs in that the system integrations may address problem domains which differ to a greater extent than different Linux variants do. Examples of these different problem domains, or use-cases, are given in the next section.

Design Reference Use-Cases

We would like to have a common software architecture for at least the following application areas:

- **Desktop Data Processing and Analysis.** The user at their workstation interactively processes and interacts with science data to perform their personal research. The data involved may be user data, workgroup data, public data from some archive, or any combination of the above. The same facilities are used regardless of whether data or computation is local or remote. It must be possible to exploit local resources (e.g., a workstation or departmental cluster) for processing, but access to remote resources is

desired as well. The toolset available is controlled locally by the user. The user may write their own software for custom data processing or analysis. Most commonly this will be at the level of a script or workflow executing within the provided desktop environment, but capabilities for user development of computational components should be provided as well.

- **VO Data Access and Analysis.** In this case we have a service implemented as a front-end to some archive, with high bandwidth access to the stored data and with access to adequate computational resources local to the data. The archive in question may be a specialized archive for some specific data collection, e.g., an observatory or survey archive, or a large scale data warehouse on the Grid. We would like to implement the standard VO data access services as well as provide the ability to execute user processing scripts with high-bandwidth access to the data. Even the standard VO data access services may require significant computation (e.g., OTF generation of images or spectra, simulated observations, or computation of virtual data conformant to a standard data model from archival data), so this use-case may require significant scientific computation beyond what a mere Web service interface would provide. If we add the capability to execute user-defined computation on the server (part of the Desktop use-case outlined above) then a common architecture for both server and desktop is also required.
- **Pipeline Processing.** Generation of virtual data products in response to a VO data access request is very similar to conventional pipeline processing. In both cases a sequence of mostly automated processing steps are performed to generate the desired data product from more fundamental data. The main differences are in how data processing is driven and in the type of processing performed. In the case of data access, processing is driven by a client trying to access a virtual data product via a Web service; in the case of an automated pipeline, processing is driven by the dataflow from an instrument or survey. While the processing involved is very similar, the system integration required may be quite different - a facility pipeline for an instrument may look quite different at the top level than a VO service, and may involve completely different software at the top level. But at the level of the actual science data processing required the two systems may be very similar. (Not everyone agrees that pipeline processing should be a primary use-case but we include it here nonetheless as it is a requirement for some of us to export pipelines to users and not just run them in a one-off fashion in the back room).

All of these use-cases involve complex, data and computationally intensive scientific data processing, where a number of processing steps are performed, each of which may require application of a complex algorithm or process. While the user interface may vary greatly, in most cases the processing required is much the same regardless of the context, or how much data is involved, or whether the data is local or remote. Hence issues such as scalability or location transparency should be factored out, dealt with as part of the execution framework which controls how the computation is actually carried out. In all cases the processing required is similar, hence a common solution is indicated, allowing us to share software for all of these use-cases. Aside from software sharing, a common solution is desirable as we need to allow observers to manually process observational data from the

desktop, and we would like to provide the flexibility to dynamically deploy software components in any of these three contexts.

Core Architecture

If we analyze the high level requirements presented above we find that a different approach is required for science software than for system software. Science software is computational in nature, often complex in the processing required, but with a highly abstracted interface which largely isolates it from the details of the external environment in which it functions. System software, e.g., for the execution framework, for data management, or for user and other external interfaces, is where we address systems problems like scalability and distributed execution, and is where we most need to capitalize on modern technology and the wealth of software available from outside astronomy. Below a certain level all scientific data processing is similar in nature, with adaptation to the different use-cases occurring mostly in the highest level software.

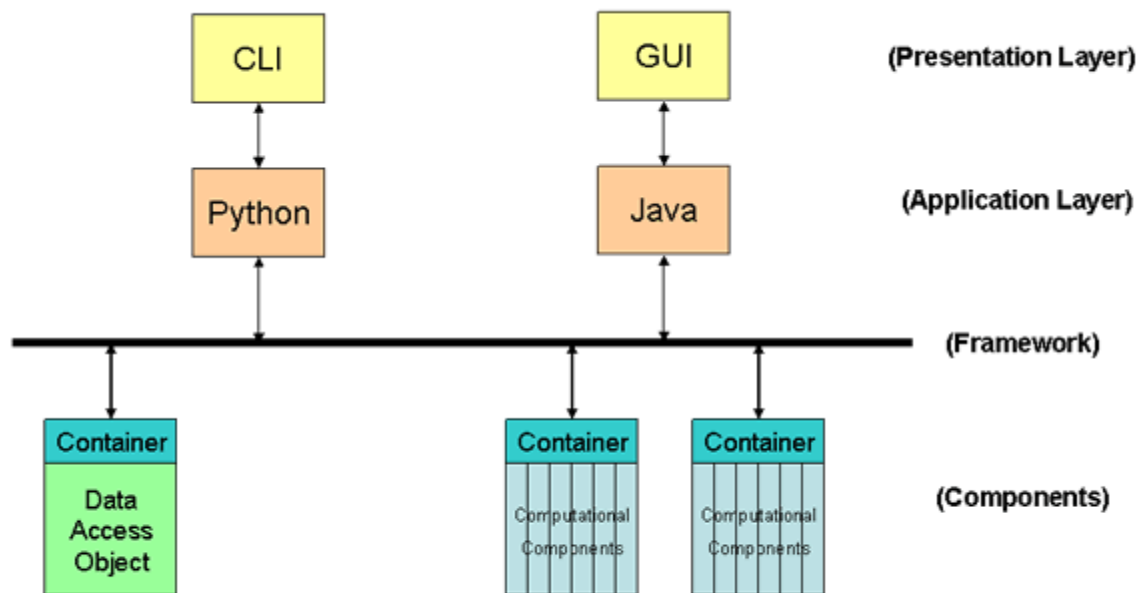
Component-Framework Architecture

This analysis, plus related requirements for things like scalability, multi-language support, support for legacy software, and so forth 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. Components execute within a **container** which defines the life cycle and run time environment seen by a component. The container is in turn controlled by an **execution framework** of some sort. At the highest level an **applications layer** brings it all together and is used to steer things. A **presentation layer** may also be required to present the functionality provided by the system to the outside world.

- **Presentation Layer.** Presents the functionality of the system to whatever outside agent drives the system, be it a human user, a telescope dataflow, a Grid workflow, a Web browser interface, or whatever. In implementation terms the presentation layer might be a CLI, a GUI written in Java or C++, a Web service interface written in Java or .NET, and so forth. In principle the same system functionality can be made available in all such contexts.
- **Applications Layer.** The applications layer is used to implement top level applications. In implementation terms the applications layer can be anything which can talk to the execution framework to execute components. For example, a scripting language such as Python, a high level programming language such as Java, a GUI written in Java or C++, a workflow engine for a pipeline, etc. The applications layer is high level "glue" code, with all of the heavy processing taking place in components.
- **Execution Framework.** The execution framework (often referred to merely as "the framework") defines the distributed virtual machine seen by the applications layer, and provides the functionality needed to allow applications to execute components. The execution framework provides 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. For a scalable system for use on a desktop or high performance cluster we need a fully distributed framework capable of managing the execution of components running on a number of compute nodes simultaneously.

- **Components.** A component is a computational object, with one or more service methods, which can be plugged into the framework. Components are often grouped into **component packages** of related components. Components provide all the real functionality of the system. Components can be written in any major language and can be either new code or can be produced by wrapping legacy code. Components written in different languages can be mixed together in the same system at runtime. For scientific use the highest priority is to support components written in compiled languages such as C, C++, and FORTRAN, however components may also be written in other languages such as Java, or may be produced by putting a component wrapper in front of scripting language code such as Python.



Component-Framework Architecture

The component-framework architecture outlined here is an **open architecture**. What this means is that, while we have an overall system architecture in mind, the major elements of the system can be used separately, and can be integrated into other system architectures. For example, most components can be used stand-alone and can be integrated into any externally-defined system. The execution framework is a separate product which can be integrated into any system. Any technology can be used for the applications and presentations layers. Components are interchangeable (in terms of interface if not semantics), as are execution frameworks. Any technology can be used to build system elements such as components or the system framework. While we baseline Python as the

de facto scripting language, there is nothing to prevent other scripting languages from being used as well. A major advantage of an open architecture is that the major elements of the system can evolve independently, making it easier to use new technology as it becomes available.

The component-framework architecture addresses the overall system structure and functioning, but says little about what goes on *within* a component. In general there is no guarantee that components from different sources will be interoperable. While this is an important problem it is one which is best addressed separately, e.g., by defining **standard data models** for the data objects operated upon by components. The execution framework itself, being system software, is neutral about what goes on inside a component.

Component-Container

Components execute within a container which defines the life cycle and runtime environment seen by the component. The container defines how a component is invoked and the interface to the external world seen by the component. In principle neither the component, nor the developer writing it, needs to know anything more about the execution framework, applications and presentation layer, or the environment within which the component will be used, than what is defined by the abstract interface defined by the component-container interface. This is important not just to make component development easier, but to ensure that components are re-usable in various contexts.

- In the simplest case the component is a pre-existing host program to be invoked with arguments on the command line, and the container is a process which runs the "component" as a managed subprocess. This approach has the advantage of allowing much existing software to be run without modification with only an adapter, but only a limited component-container interface is possible.
- More generally, the container is a library which is linked with the object code of one or more components to produce an executable process. Linking can be either at compile time (the usual case for most science code) or at run time via a dynamically loadable library (useful for smaller plug-in extensions). The container manages the interface of the component to the execution environment.
- Encapsulation of the component interface in the container allows **multiple framework protocols** to be supported. At the simplest level a single component can be invoked by running the container at the host level, with arguments on the command line. For fully distributed execution the container can be run from the execution framework via a protocol such as CORBA or SOAP which supports concurrent execution of components. In this case the full range of container services are available to the component for runtime messaging, communication with other components, passing parameters in both directions, and so forth.

The container provides standard **container services** which are available to any component. These include two-way parameter handling, a distributed environment mechanism, remote

method invocation, and asynchronous messaging. At the most basic level, **messaging** allows messages (arbitrary blocks of information, often tagged by a message class and related metadata) to be efficiently exchanged between components during execution. Messaging may be either point-to-point between two components, or broadcast, in which case a component produces messages which are broadcast to zero or more message consumers. Clients can dynamically register with a messaging service to indicate the classes of messages they wish to receive. Messaging includes support for **message streams**, including high performance point-to-point data pipes, a **logging** service, used by components to broadcast time-tagged log messages, and **property change events**, used to broadcast changes to the state of a component to subscribing clients, e.g., a GUI display.

In the most general case a component is a distributed object (DO) implemented as a class with methods. Some of these methods are standard methods defined by the container and used to implement the standard container interface and life cycle. Other methods are custom **service methods** defined by the component to expose whatever custom functionality the component provides.

Tasks and Parameters

An important type of component is a **task** component, which implements a single service method. Task components use a **parameter** mechanism to control the task. Unlike DOs, which can be long running and stateful, tasks are general stateless, with all state maintained in the client, in the framework, or in persistent external storage of some sort. Tasks have the advantage of having a relatively simple interface and simple runtime semantics, increasing deployment flexibility and aiding scalability. DOs provide finer grain functionality at the cost of a more complex interface and more complex runtime semantics.

Parameters are organized into named **parameter sets**. Every task has an associated parameter set, used to pass data operands and control parameters to the task. Parameter sets can also exist independently of tasks as named data entities. Some parameters must be given values in order to execute a task. Other parameters have default values and are "hidden"; hidden parameters need only be specified if it is necessary to override the default value. Parameter sets can also be used to return data from a task to the caller (e.g., to a script) or to pass modest amounts of data between tasks.

A parameter set consists of a set of **parameter** objects. Each parameter object has a number of attributes such as a parameter name, type, default value, min and max or enumerated value, query mode, prompt string, help reference, and optional unit and error values. Both primitive (string, int, float, bool, etc.) and abstract (e.g., "file", "cursor", etc.) parameter types are permitted. While a parameter set defines a simple flat namespace with no hierarchical structure, a parameter set may reference one or more other parameter sets, allowing simple hierarchical relationships to be defined. While a full description is required to define a parameter set, all that is required at runtime to invoke a task is a simple keyword

table or dictionary consisting of parameter name and value pairs.

Execution Framework

The execution framework connects all the other elements of the system, allowing client applications to execute components, and allowing components to communicate with each other. At the core the framework is a software bus which provides a standard way for components to connect ("plug in") and communicate via messaging. Major elements of the system framework include the following:

- **Distributed Virtual Machine.** More than just a software bus, the framework implements a distributed virtual machine (DVM) abstraction. The DVM manages the execution of components on multiple compute nodes as if they were a single machine, providing transparent scalability for cluster computing. Compute nodes may be dynamically added to or removed from the DVM during execution. On a cluster the application script would normally run on the head (login) node, with all the computation taking place in components running on compute nodes. A parallel file system of some sort would provide high performance shared access to storage for all compute nodes. On a workstation everything runs on the head node. Efficient execution on a single machine is a priority for desktop use. An optional software console provides facilities for booting and controlling the DVM. An optional logging service provides instrumentation for monitoring the execution of components.
- **Package Manager.** The package manager service is responsible for managing all information related to component packages and parameter sets. When a component package is "plugged into" the framework, the package metadata is ingested by the package manager, defining all components and making them immediately available for execution via the DVM. In addition to registering components and maintaining their metadata, the package manager provides ***persistent storage*** and distributed access methods for parameter sets.
- **Software Bus.** The software bus provides basic facilities for components and containers to plug into the framework, for execution of components, for exchanging messages between components, and so forth. Further information on messaging is given in the container discussion above.

To interface an applications layer component such as Python or Java to the framework, all that is necessary is to bind the framework services into the target language environment.

In implementation terms the framework can be anything which implements the logical model and services defined by the framework definition. This could be anything from a thin layer in an applications language such as Python (with the Python session directly executing components), to a fully distributed framework implemented using some technology such as CORBA or SOAP, possibly combined with a package such as MPI for high performance messaging.

Scalability

Scalability is provided by the DVM, by the execution framework and messaging system, and by the underlying cluster hardware and software, e.g., parallel file system. In the simplest (default) case, everything executes on a single node such as a workstation or laptop. In the case of a cluster, interactive components and application scripts execute on the head node, computational components execute on a variable number of compute nodes, and one or more storage nodes provide shared access to parallel file system, with a fast switch connecting all nodes. In a Grid scenario, the system described here is itself a compute or storage element in some externally managed Grid workflow (see the discussion of VO/Grid integration below).

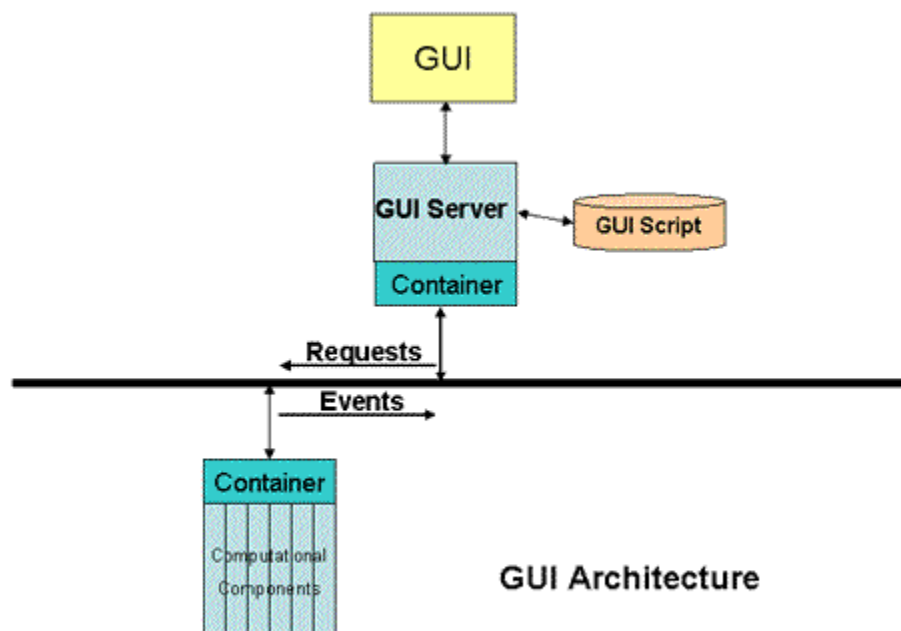
- Basic scalability is provided by the distributed execution capabilities of the DVM in combination with a parallel file system. This provides a basic data-parallel capability, with multiple compute tasks executing independently on separate datasets (examples of parallel file systems include, in no particular order, PVFS/PVFS2, GFS, IBM GPFS, Lustre, Ibrx, Mosix, etc.).
- Parallel algorithm support requires use of an existing technology such as MPI. A mechanism is needed to tell the DVM to execute N copies of a component (distributed over the compute nodes) which will function as a parallel unit. A parallel task appears as a single unit of computation to the applications layer.
- Transparent support for data-parallel computation (the same operation applied independently to N separate datasets) requires some support at the applications layer level, e.g., in Python. The same simple procedural SIMD script is used regardless of the number of compute nodes available, with the framework automatically scaling the computation to utilize the available resources. All that is required to implement this is some sort of RUN construct in the scripting language which tells the framework to apply the same operation (task) on N datasets in parallel.
- Parallel operations on extremely large datasets may require concurrent operations on portions of a large dataset. There are various ways to address this problem. One promising approach is to use a shared `__data access object__` (DAO) to mediate access to the object by multiple concurrently executing compute nodes. The object itself is stored in numerous smaller files at the level of the storage system (this is desirable in any case for an extremely large dataset), with the DAO managing access to the overall logical data object. The actual file i/o is parallel at the level of the individual object elements, with each compute node having a parallel (or nearly so) data path to the file element via the parallel file system. Alternatives would be to handle the i/o directly in the DAO, or a use a combination of DAO i/o and file i/o.
- Concurrent database access is something which has been provided for years by relational databases. The architecture is similar to a DAO, with the database server mediating concurrent access to a database or table. Modern database systems take this one step further and have the ability to distribute large queries over the nodes of a small cluster. Further gains are possible via segmentation of a database at the logical database design level.

The challenge in providing scalability in our case is managing the complexity and size of the

system. While we want to achieve transparent scalability, it is essential that the resultant system be "lean and mean" enough to be usable on a modest desktop system. This means that the system must be no more difficult to use on a desktop system than a nonscalable system, and the system size and startup time must be comparable to a typical desktop system.

User Interfaces

User interfaces are part of the *presentation layer*. User interfaces are not required in all circumstances, e.g., a server which exports functionality via Web services need not implement a user interface. An interactive desktop system will however require user interfaces such as a command language interface (CLI), various graphical user interfaces (GUIs) or visualization components such as an image display or data browser. User interfaces can also be providing using Web browser technology or via a scripting language such as Python.



CLI. The purpose of a CLI is to provide a streamlined environment for command entry. A good CLI will provide a simple syntax for command entry, e.g., without the need to parenthesize argument lists or quote strings. Parameter support should be included including parameter defaulting, prompting for missing parameters, parameter editing, and so forth. Shell-like facilities for command completion, i/o redirection, history, aliasing, and so forth are often provided. Simple control flow constructs may be provided in the language so long as doing so does not conflict with the efficient use of the CLI for command entry. Complex scripting is better done in a real scripting language such as Python or Perl.

While there are many ways one could implement a CLI, one approach which would work well given Python as the scripting language would be to implement the CLI as a Python module. Since the framework interface would be to Python itself, the CLI module would see only a Python API hence would be largely framework independent. While the CLI could be implemented as a separate component, putting it in the same address space as the Python interpreter would provide better integration with the scripting language and would allow an expert user to alternate between the CLI and interactive Python within the same session.

GUIs. A number of alternatives are possible for implementing GUIs. Implementing GUIs directly in the scripting language environment is one possibility. This can be a good way to provide a well integrated environment for user software development, or to provide dedicated GUIs for a CLI (such as a parameter editor), but if carried too far starts to overload the scripting language component. A more scalable approach is to provide a **GUI server** component which executes a downloadable (applet-like) GUI script, and which uses messaging to communicate with other components via the framework. The GUI listens for property change events transmitted from remote computational components via the framework, updating the GUI display as events occur, and sending requests to the remote component in response to user commands entered via the GUI. For applications where messaging performance is an issue, a dynamically loadable plug-in component can be used to move computational components into the address space of the GUI. The GUI server approach works well for adding optional GUIs to computational components, as well as for implementing large visualization components such as an image browser.

Legacy Software Integration

The principal strategy for adapting legacy software to this new architecture is to wrap legacy software as components. Most components would be computational components (science software). Legacy software could also be adapted for use in the presentation layer, for things like the CLI, data visualization, image display, and plotting. At the simplest level, integrating legacy components is merely a matter of writing an adapter. To fully integrate such software it might be necessary to modify the portion of the legacy software which implements the existing tasking, parameter management, etc., functionality to use the new container technology. One might also add some code to the component to use new facilities such as runtime messaging and logging. Studies of existing systems indicate that this is quite feasible. Since the container can support multiple runtime protocols it might be possible to provide backward compatibility to support legacy frameworks and user interfaces.

The most notable problem with integrating legacy software into a common architecture is not system software, but incompatible data models and data formats. While legacy software can function within a common framework without addressing this problem, interoperability of code from different systems will be limited.

Data Models

Much of science data processing and analysis concerns operations upon complex data objects such as flux and astrometrically calibrated sky projection images, spectral data cubes, multiband imagery, synoptic imagery, multiobject spectra, 1D spectra, spectrophotometric time series data, high energy event data, interferometric or single dish visibility data, pulsar data, GRB data, synthetic (model-generated) data, simulated data, and so forth. Data may be fully calibrated or may be raw instrumental data including instrument-specific metadata. Datasets may be produced by the combination of instrumental datasets, e.g., SED data, or dithered and stacked mosaic imagery.

To deal with such data in any general way requires a formal specification of the data model for the data. For astronomical data we call this the **science data model** (SDM). All data processing and analysis is defined in terms of the SDM for the data being operated upon. To store and transport data we must also define a data representation in some concrete external storage format such as FITS or XML. This mapping of the SDM to a specific data representation is called an **export data format** (EDF) for a particular class of data.

To process large, complex datasets it is generally necessary to go one step further and implement a class library for a particular type of data. This requires a choice of language (C, C++, Java, etc.) and often implies a dependence on lower level software, e.g., class code for a FITS container or an XML parser. Issues such as efficiency and scalability (for very large datasets) are generally dealt with in the class library. Ideally the implementation of such a class library is such that it can be re-used by multiple data processing "systems" (class libraries used to build components). In the general case, a **data access object** (DAO) may consist of an externally defined, formal data model, a storage representation, a class library, a DAO component which can interface to an execution framework, and an API which can be called by science code in a computational component. Given all this, multiple groups can then proceed to build interoperable data processing and analysis software for a given class of data.

Data models fall into two broad classes, data models for **calibrated** data, and data models for raw **instrumental** data. In general, VO is responsible for defining standard data models for standard classes of calibrated data, as well as standards for observational metadata and physical data characterization which are generally applicable to all astronomical data. The observatories which produce instrumental data are necessarily responsible for defining both the SDM and EDF for data from a specific instrument or survey project. Both of these are external interfaces which must be defined formally as external interfaces in order to allow reliable processing of data products by the general astronomical community.

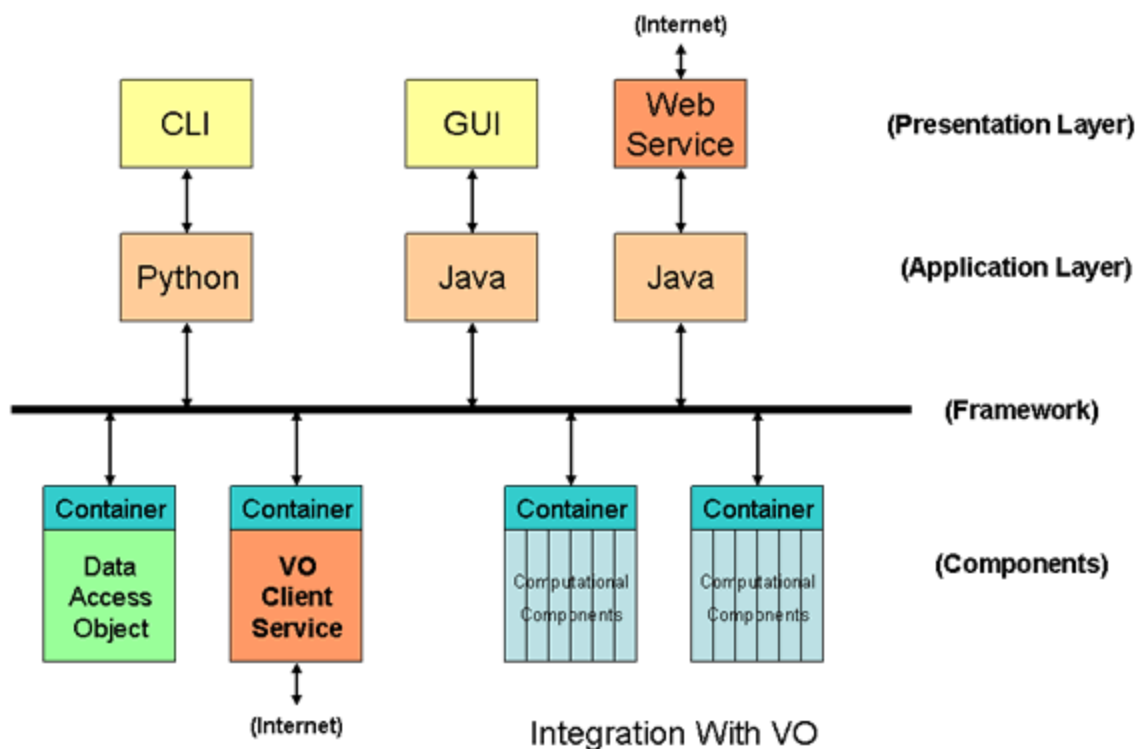
VO/Grid Interface

As noted in the introduction above, VO is mainly about middleware and how widely

distributed components talk to each other and interoperate. In the VO context, data and computational resources are exposed as services which are called by remote Web clients or as nodes within a Grid workflow. For the most part VO is not concerned with how computationally intensive services or client data analysis programs are implemented, but that is precisely our concern here.

VO Service. In this case we have some computation which we wish to expose to the VO as a Web service. Examples might be computation of a virtual data product for a data access service, or an analysis function of some sort. The basic strategy is to implement the Web interface with conventional Web technology (e.g., Java/Tomcat/AXIS), and use the data analysis system to handle the complex scientific computation required to generate the virtual data product to be returned by the service. For the purposes of our discussion here lets assume we are using Java for the Web interface; other technology such as .Net could be used as well given the necessary interfaces.

- The Web service interface is the presentation layer in our architecture, responsible for the interface of the processing engine to the caller (some VO client).
- The Web service functionality and related processing such as database access, authentication, etc. could be handled directly in the Java code which is well suited for this type of processing.



- Java is interfaced directly to the execution framework hence it is straightforward to drive the processing engine from a Web service front end.

- Computationally expensive, long running services are possible using the asynchronous services technology currently being developed for the VO.
- Computation of science data would be performed by a conventional application, e.g., a Python script or Java program, using the framework to execute computational components. The same application might be available in other contexts as well, e.g., from an interactive CLI.
- Aside from the Web service interface and parts of the top level application, all of the code involved is the same code we would use for other applications such as desktop data processing and analysis or a pipeline.

In general the components used in a data analysis system are too fine grained to be usefully exposed to the VO as Web services. Rather we usually need to write a new application which is designed to be used in the VO/Grid context, and expose this as a Web service. The application itself is written in the application layer (e.g., in Python or Java), using other applications plus the framework and conventional components to do the processing. Such applications may be complex and may involve extensive processing using many individual components.

Almost any computation could be exposed as a Web service in this way. Since the framework is scalable this would provide a way to execute arbitrary bits of computation remotely for large scale problems, e.g., to move the computation to the data. Given two copies of the system, one local and one remote, with the same component packages installed in both locations, it would be possible to dynamically deploy an application script interfaced as a Web service to run remotely. The script could be developed and tested locally and then deployed remotely, possibly on a much larger system.

VO Client. In this case we have an application which wishes to make use of VO resources as part of some computation. Probably this is some sort of data analysis application which needs access to remote data or possibly computation.

- Once again: science software, for various reasons such as maximal re-use, does not want to know about external details such as frameworks and protocols, so we want to hide the details of how to talk to the VO. A high level interface of some sort is required to hide details such as use of a Web services interface.
- Some VO functionality can be made directly available to the data analysis system by interfacing the corresponding Web service to the framework as a component (task or DO). Such functionality would be available to any client application using the same interface provided for locally executing components. (This same mechanism would be used to execute application scripts remotely as mentioned above).
- More complex services such as the client side of the VO data access layer (DAL), require some client-side functionality beyond just what is required to interface to a Web service. This should hide details such as query handling, authentication, asynchronous services, and caching of data fetched from a remote VO service.

For example, when a science application does a query, it usually does not want to get a VOTable back, rather the application would like some client-side code to submit the query, parse the response, and provide a high level API to the application to iterate through the query response line by line (much as a client would query a conventional database). Probably this would be implemented by a service running locally, which knows how to talk to remote VO services, but which can also talk to local data analysis software via the execution framework.

When accessing remote data, the local VO client-side service would not only fetch the data but cache it locally. All access to the data for data analysis would then be to the cached local copy of the data. In combination with capabilities for asynchronous data staging it would be possible for a remote data access service to deliver the data directly to the client-side cache without ever storing it on the server. By providing Web access to cached data the same mechanism could be used to expose data back to the VO for consumption by remote clients, for example to return data products from a local data access or analysis service. Within the local data analysis system, equivalent access would be provided for local and remote data.

The client-side data caching functionality described here is analogous in some respects to capabilities such as "VOSpace" in VO, since both facilities have the ability to cache data products as part of a VO workflow. The chief difference is in the client interface for data analysis and the need for transparent integration into a data analysis system. The VO side of the interface should adhere to VO standards, e.g., by implementing a future VOCache API and by using relevant standards for Grid and Web services.

Summary

The software architecture explored here is intended primarily to provide a processing engine for complex, data and computationally intensive scientific computation. A high level application, usually implemented as a script, drives computation via a distributed set of computational components controlled by a framework. The software has an open architecture, allowing a great variety of systems to be produced from the same underlying components.

Now that we have a software architecture in mind, we can go back and reconsider our various users and see how well a system based on the proposed architecture would server their needs:

- **Astronomer.** Processing of user or workgroup data is integrated with data analysis of user and public archival data. Various user-oriented tools, such as a CLI, advanced scripting environment, GUIs, and user-configurable software mix, are provided, executing efficiently in the desktop environment. Data and computation can be local or remote. The biggest challenge is probably managing complexity and keeping the resultant software sufficiently well integrated, lightweight, and "astronomer friendly" (particularly for user software development) to be successful in the desktop

property of the contributing authors.

Ideas, requests, problems regarding Lepus? [Send feedback](#)