Applications Framework Architecture and System Interfaces

DRAFT - Version 0.3

IVOA Note 2009 January 15

This version:
This Version-YYYYMMDD

Latest version:
http://www.ivoa.net/Documents/latest/latest-version-name

Previous version(s):
(none yet)

Authors:
D. Tody (NRAO), W. Cotton (NRAO), M.J. Currie (RAL), M. Fitzpatrick (NOAO), P. Grosbol (ESO), L. Paioro (INAF), C. Surace (LAM/OAMP), others TBD

Editors:
P. Grosbol (ESO, OPTICON), D. Tody (NRAO, USVAO)
Abstract

An applications framework is presented here which can be used to implement and run astronomical data processing and analysis applications. Typical use cases include instrument data processing packages and pipelines, interactive user data processing and analysis environments, VO (virtual observatory) data analysis workbenches and portals, and VO services of various kinds, including analysis services and data access services which generate virtual data products. The framework provides interoperability for applications packages and tools from many sources, and support for distributed and scalable execution. Applications see a simple API for interfacing to the framework, allowing scientific users to develop applications without having to deal with the complexities of the framework implementation. The applications framework is composed of a number of system elements which may be used either individually or as a complete applications framework. System integrators may use individual system elements or the entire framework, in combination with arbitrary external software. This document introduces the major elements of the applications framework and describes their function and interfaces at a conceptual level.

Status of This Document

This is an IVOA Note. The first release of this document was 2008 June 7.

This is an IVOA Note expressing suggestions from and opinions of the authors. It is intended to share best practices, possible approaches, or other perspectives on interoperability with the Virtual Observatory. It should not be referenced or otherwise interpreted as a standard specification.

A list of current IVOA Recommendations and other technical documents can be found at http://www.ivoa.net/Documents/.

Acknowledgements

This document builds upon many existing standards and related systems, including the earlier OPTICON sponsored studies of requirements and architecture for a Future Astronomical Software Environment (FASE), as well as many VO standards efforts such as those for the data access layer [3], [4], [5] for accessing astronomical data from an analysis environment, the Common Execution Architecture [6] for exposing arbitrary computational tasks to the VO, the simple application messaging protocol (SAMP) [7] for inter-applications messaging, and the various VO Grid standards for distributed execution of applications. Many astronomical data analysis systems and their developers have also contributed to the ideas presented here, including (AIPS, AIPS++,
CASA, IRAF, GIPSY, MIDAS, Starlink, STSDAS, ??). Some of these are potential candidates for migration of packages and tools to the new applications framework described here.

Contents

1 Introduction 4
  1.1 Project Goals 6
  1.2 Scope of this Document 7
  1.3 Key Concepts 7
    1.3.1 Applications 7
    1.3.2 Application Packages 8
    1.3.3 Tasks and Tools 8
    1.3.4 Parameter Sets 10
    1.3.5 Task Execution vs. Computation 11
    1.3.6 Applications Framework 11
2 Architecture Overview 11
  2.1 System Layers 12
    2.1.1 Presentation Layer 12
    2.1.2 Applications Layer 13
    2.1.3 Execution Framework 13
    2.1.4 Component 15
    2.1.5 Class Library 15
  2.2 Applications Architecture 16
    2.2.1 Host Execution 16
    2.2.2 Single Process With Inline Container 17
    2.2.3 Fully Distributed Configuration 18
    2.2.4 Task Structure 19
    2.2.5 Legacy Code 20
3 Framework Elements 20
  3.1 Packaging 21
    3.1.1 Logical View of a Package 22
    3.1.2 Package Structure and Contents 22
    3.1.3 Interface Definition 23
    3.1.4 Package Discovery 24
  3.2 Component Container 24
    3.2.1 Types of Containers 25
    3.2.2 Host Execution 26
    3.2.3 Parameter Management 28
    3.2.4 Messaging and Logging 29
    3.2.5 Components 30
    3.2.6 Container-Framework Interface 31
    3.2.7 Minimal Container 32
  3.3 Parameter Mechanism 32
1 Introduction

Astronomical research depends critically upon data processing and analysis software, much of which was designed and implemented twenty or more years ago. The effort spent on using, adapting, and developing software consumes a substantial fraction of research project budgets, with frequent duplication of capabilities in incompatible systems. The 2008 NASA Senior Review of data centers noted “NASA and their partner institutions have not agreed upon a common philosophy in the development of software... Whatever the agreed-upon
policies, they should be long-lived and have international commitments.” While the computing and data analysis paradigm has advanced rapidly in other scientific domains, astronomers are generally working behind the technology curve and outside of the software development mainstream.

The projects of the coming decade require a strategic investment in the software needed to reduce, analyze, and interpret the massive influx of complex new data. Our objectives are to:

- Improve the efficiency of research and the astronomer’s ability to deal with large data sets through better interfaces, visualization, and analysis tools.

- Exploit hardware advances such as 64-bit computing, multiple cores, cluster, grid, and cloud computing.

- Support direct access to distributed data sets, particularly through the Virtual Observatory.

- Increase software sharing, re-use, and transparency.

- Phase replacement of obsolescent software with new systems, with careful attention to widespread community involvement and transfer of legacy applications.

The applications framework described here will provide a shared infrastructure within astronomy for constructing various types of astronomical data processing and analysis systems. These include instrumental data processing packages or pipelines, data analysis workbenches or portals providing access to both local data and computation as well as remote VO (virtual observatory) resources, and VO services of various kinds, including both analysis services as well as VO data services which carry out potentially complex processing to compute virtual data products. The applications framework is intended to be used for conventional observatory data processing and analysis as well as VO-enabled analysis, allowing both to be integrated within a common environment, and allowing observatory data processing systems to benefit from VO technology and vice versa.

The applications framework project directly addresses only development of the framework itself; use of the framework to construct complete facility systems is left to the various projects which might use the software. This approach promotes reuse and interoperability, while giving system integrators full control over the final system they deliver. Many types of systems could potentially make use of the framework software or individual framework standards. A common system for processing data from multiple observatories and archives is possible provided the framework standards are adopted by the major projects, enabling interoperability of software at the level of individual applications and tools.
The applications framework project is intended to develop both standards (subsystem and interface specifications) and reference implementations for the key elements of an overall integrated applications framework that is a system framework for developing and running astronomy applications. So far as possible the elements of the system are intended to be usable separately, so that system integrators can pick and choose what is most useful for their system. Nonetheless it is also a goal for the elements of the system to be able to work together to form a complete applications framework. Interfaces are defined independently of implementation, allowing multiple implementations from different sources to interoperate at runtime.

### 1.1 Project Goals

Some key goals of the applications framework project include the following:

- **Promote application interoperability**, so that applications from different observatories and projects can be used within a common environment, simplifying things for the astronomer and enhancing reuse of common tools and libraries.

- **Integrate VO technology and resources with the data processing environments commonly used by astronomers.** This can potentially provide a whole new multiwavelength dimension to typical research data analysis software, and significantly increase the impact of VO on everyday astronomy.

- **Provide advanced capabilities to astronomical software**, e.g., for distributed computing and scalability or for data access and management. This leverages work done on related technology for VO, e.g., for distributed authentication and distributed computation. Scalability is an increasingly important capability for processing large datasets and for data analysis within the VO, as well as to exploit the increasingly parallel architecture of modern computers.

- **Keep application development simple**, so that astronomers and scientific programmers can develop scientific software without having to deal with the complexities of modern software technologies and environments. The abstraction required to do this also helps increase the reusability of software components and applications, and helps isolate science software from the underlying technology required to provide advanced capabilities.

Achieving project goals such as execution of applications packages from multiple sources within a common runtime environment, and interoperability of such applications with tools from other sources, is only possible if we are able to establish standards for open interfaces such as those required to load and execute an application or application package, or for an application to
communicate with visualization tools via messaging. Hence a major goal of the applications framework project is to develop such standards. The intention is to separate interface from implementation to allow multiple implementations of major system elements, to provide flexibility and the ability to adapt to specialized environments, and allow for evolution of the technology used for implementations.

1.2 Scope of this Document
This document is intended primarily to provide a conceptual design for the applications framework, identifying the major system elements and their interfaces, and describing how each element of the system functions. A high level system architecture, consistent with what is presented here, was described earlier [1] (this document has recently been updated). A requirements document [2] describes the requirements of an overall data processing and analysis system based upon the applications framework.

This document extends the earlier architecture to propose a specific breakdown of the overall framework into key system elements with externally defined interfaces, potentially allowing interoperable implementations of subsystems from multiple sources, mixed within a single runtime system. While we do not yet attempt to define the interface of each element of the system in detail, a conceptual design is presented which outlines the function of each system element and shows how they can be used together to provide a complete system framework.

Although the primary audience for this document is architects and designers of software for astronomical data processing systems, it may also be of interest for astronomical users with a thorough knowledge of software design. For such astronomical orientated users, it may be better to first read some of the use cases (section 5), which illustrate how the proposed framework would be used to implement typical use cases, and then go back to read the more detailed elaboration of the system concepts and system elements.

1.3 Key Concepts
1.3.1 Applications
A framework application is a top-level program which relies upon resources provided by the framework for some portion of its functionality. A typical application is written in a high level language and relies upon separate reusable components or class libraries, often written in compiled languages, for the bulk of its processing. Applications may execute in a distributed fashion and may be scalable, with the application framework providing the mechanism to manage the distributed computation on behalf of the application.
In the simplest case an application uses the framework but is not itself callable from the framework. In other words it is a “top level” application, such as would typically be called by a user – or any other entity which is external to the system. Although applications are often intended for use by human users, they may also be more specialized applications, such as an automated data reduction pipeline, or a Web service. In the most general sense an application is any top level program which applies the resources provided by the system to solve some specific external, real world problem.

The applications framework defines the framework facilities available to the application, but does not define the form of the application itself. This is intentional to allow many types of applications to use the framework middleware (for example a CLI-callable task and a Java servlet are quite different at this level). Although the framework does not specify how an application is to be implemented, it is expected that large systems which use the framework as part of the environment they define for applications development will want to do so.

1.3.2 Application Packages
An application package is a collection of related applications which can be loaded or deployed within a running system and immediately used. While a package may have external dependencies, often any components used by the package will be included in the overall package. In general a package may contain any number of applications, possibly organized into sub-packages, including any components, dynamically loadable libraries, or other runtime files required for use of the package. Packages are self-defining, containing a description of the contents of the package, interface definitions for all components, online help documentation, and so forth. Packaging is extremely important to ensure a modular system which is easily extensible and easy to use. The package mechanism also makes it much easier to take software from multiple sources or projects and run these within a common framework.

1.3.3 Tasks and Tools
As we shall see later in section 3.2, one of the main functions of the applications framework is to execute components which run within a container. This provides a standard mechanism for defining and invoking computational or data access components at a higher level than that of a class library, providing many other capabilities in the process including transparent distributed, concurrent execution and scalability.

In order to make components manageable, two standard types of components are defined. The simplest is the task, which is a component with only a single method. Tasks are the most fundamental unit of computation at the level of the execution framework. A task is driven by a parameter set describing the parameters defined to control the execution of the task. Tasks have the
advantage of being simple and scalable, with maximum flexibility in terms of dynamic runtime deployment and execution. Since a task has only a single method it is stateless, at least in terms of execution by the framework. When a task is run it starts up, executes, and shuts down as one operation.

Tasks may be serial or parallel, with multiple instances of the task executing concurrently in the case of a parallel task (to the application both cases appear to execute as a single task). Parallel tasks implement the SPMD model and typically operate upon distinct segments of the data concurrently (in which case the task may be serial internally), or implement a parallel algorithm of some sort, e.g. using MPI (OpenMPI, MPICH, etc.). Serial tasks execute as single processes and are serial from the perspective of the execution framework, but may exploit finer grain parallelism internally, for example using a multi-threading technology such as OpenMP, which can provide some degree of automated parallelization of code at compile time (GCC 4.2 and later, Microsoft Visual C++, and others include OpenMP support).

Both coarse and fine grain parallelism techniques may be combined, for example a parallel task may be multi-threaded internally to exploit both multi-core and cluster resources in a single task. Processor cores can also be used to execute separate processes in the case of a parallel task, although care must be taken to avoid I/O and memory bandwidth bottlenecks. GPU computing (e.g., using CUDA) is another fine grain parallelization technology at the same level as OpenMP, and would have a role similar to OpenMP in task parallelization.

A more general case of a component is the tool, which is a component with multiple methods, each of which has its own parameters. Tool components are stateful, i.e., starting and stopping the component are distinct operations, and any number or sequence of tool methods may be invoked while the tool is running. Computations which depend only upon their inputs (almost any data processing operation) usually fit the task model well, with the overall computation being decomposed as a sequence of tasks, each of which may be singular or parallel. Services which manage the runtime state of some resource are more likely to require a tool interface, for example a data visualization tool or a data access object. The standard parameter set interface is used both for tasks and for tool methods, although more direct APIs are possible in the case of tool interfaces and are warranted in some cases. Tools execute serially, but there may be multiple instances in use simultaneously and a tool may implement fine grained parallelism internally, e.g., via multi-threading, GPU computing, or other such techniques.

In order to simplify the text hereafter we will often refer to a component informally as a “task”, meaning “a component which can be either a task or a tool”. In cases where there is a distinction between a task and a tool this will be pointed out explicitly in the discussion.
1.3.4 Parameter Sets

Parameter sets provide a simple, standard mechanism for managing keyword-value type data within the context of the application and execution framework.

The most common use of a parameter set (pset) is to define the input parameters of a task; however the concept is more general than that. Formally a parameter set is defined as a special data storage class used to manage a set of parameter (keyword-value) objects. Subclassing of the base parameter set class is used to specialize the parameter set for a specific use-case, for example the input parameters of a particular task. There may be any number of instances of a specific parameter set, for example to define the default parameters for a task, to define the parameters used to execute a task instance, or to specify custom parameter default values for using a task in a specific context.

Parameter sets may be used for task output as well as for input. In the case of an output parameter set, the resultant pset may be used as input to another task, or may be used directly by the application. Parameter sets may also be used to represent stand-alone data entities independent of any particular task. While complex data may require more sophisticated storage mechanisms, e.g., use of an astronomical image format or a table in a relational database, parameter sets provide a simple yet convenient and flexible mechanism for managing a certain class of data within the application environment.

A parameter set is a collection of parameters or parameter objects. The type of a parameter may be any simple type (boolean, integer, float, string, etc.), and in addition custom application-specific types and any associated methods may be defined by a new parameter set subclass. Additional semantics may be associated with parameters by the use of UCD and UTYPE tags, as in VO. In particular a hierarchically structured data model may be mapped (within reasonable limits) to the flat parameter space of a pset using the UTYPE construct.

Parameters may be either scalar or vector valued. Parameter sets may be parallel, defining a slightly different set of parameters (based on the process rank within a parallel process group) for each instance of a parallel task, with a single parallel pset being used for the parallel task. Parameter sets may be serialized in a text format (e.g. VOTable) for interchange and storage, but may also be passed about or referenced internally in an efficient binary object format to optimize access where the runtime mode of access permits it. Parameter sets can rival compiled binary APIs in terms of efficiency in cases where binary pset objects are passed by reference, particularly where a parameter is array-valued.

While psets may have vector valued parameters and may be serialized to external storage, the intention is that psets are small enough so that hundreds of them can be efficiently managed in memory at one time. Hence bulk data is not stored directly in a pset; rather a parameter is used to reference the external
object. The value of such a parameter may be a file pathname, URI, DBMS table name, or similar reference. A further advantage of such a reference is that it may refer to virtual data which is generated on the fly at access time.

### 1.3.5 Task Execution vs. Computation

We distinguish between execution of applications and the actual computation or data access performed by an application or one of its components. The framework functionality required to manage and execute applications is essentially the same regardless of what the application does internally or what kind of processing it performs. For example, an application which accesses remote data via the VO may be very different internally than one which processes instrumental data. An application which processes data from an O/IR instrument may use completely different code internally than one which processes radio or high energy data. This is especially true where legacy code is to be adapted to run within a common framework, but is likely to be the case even for new code if the processing requirements or development teams vary significantly for different classes of application.

In general the class library used within a component to process or access data or perform some other computation, will vary for different applications or classes of application. A single unified class library supporting all applications is probably infeasible for a large system, especially if multi-wavelength analysis or processing of data from different observatories or branches of astronomy is a goal. The execution framework on the other hand can be a common standard since it is largely independent of the computation performed by a task.

### 1.3.6 Applications Framework

An applications framework defines the environment for applications development as well as provides the runtime environment for executing applications. The full environment required by applications includes both the execution framework and any application specific class libraries used to implement an application or one of its components. Applications may also make use of standard system components, e.g., for user interaction, logging, data visualization, or data access. While our focus in this document is primarily on the execution framework and overall system architecture, it should be noted that a full applications framework supporting a certain class of applications will in general require standard or shared components and class libraries as well as a common execution framework.

### 2 Architecture Overview

In this section we provide an architectural overview to introduce the discussion of system elements provided in section 3. A more detailed review of the applications framework system architecture is provided in the original architecture whitepaper [2].
2.1 System Layers

The diagram in Figure 1 shows the layers of a typical system based upon the applications framework. Each layer is described in more detail below.

![Diagram of system layers]

Note that both the applications layer and components share the same interface to the execution framework. Although we distinguish between the two for various reasons (applications are often scripted, components may be parallel, components provide a good mechanism for structuring large applications and enabling software reuse) they are closely related. In some cases components may be called as applications, or applications may be distinct but may implement a component interface.

2.1.1 Presentation Layer

The *presentation layer* is the interface of an application to the external world. This could be a CLI, a GUI, a Webapp or Web service (e.g., servlet) interface, a
Applications Framework Architecture and System Interfaces

VO/Grid adapter such as CEA, and so forth, depending upon the nature of the application and how it is to be called. The same application might be called in multiple ways.

There may or may not be a formal interface between the presentation layer and the application. In some cases the presentation functionality is integrated directly into the application, e.g., a visualization tool with an integrated GUI. In other cases, e.g., a CLI or servlet interface, the presentation layer may be able to call any number of applications so long as they implement the interface defined by the presentation layer. In this case the presentation layer abstracts presentation-related functionality from the application. In other cases the presentation layer may be able to directly call components via the execution framework.

2.1.2 Applications Layer

An application is a program module of some sort which can be called from the presentation layer and which uses the resources of the applications framework for some portion of its functionality. The applications layer is the layer at which a top level application is implemented. Applications are often written in a high level scripting or compiled language such as Python or Java, with most of the functionality provided by lower level components or libraries. Applications can however also be written in compiled languages, or in simple cases might be self-contained, e.g., written entirely in a high level language. In some cases components might be directly callable as applications, and vice versa. Applications are often collected into applications packages which can be dynamically installed into the environment at runtime.

2.1.3 Execution Framework

The execution framework is the primary system framework, responsible for managing and executing applications. Non-trivial applications are typically composed as a high level application which calls lower level components which provide most of the functionality and computation. Often the application and the components it calls will be written in different languages. Applications or their components may need to call or communicate with other applications or components during execution, e.g., an application might generate messages during execution which drive a visualization tool.

The execution framework provides the functionality required to execute applications, including automatically executing any required components, providing communications between the application and its components, providing a parameter mechanism to manage the parameters required to invoke a component method, and so forth. Capabilities such as distributed or parallel execution are provided by the execution framework.
In the most general case the execution framework includes the following major elements:

- **Container.** The container is the mechanism used to execute components (tasks and tools). It defines a standard interface which all components must implement in order to be called interchangeably by the container. Likewise it defines a standard container API which provides capabilities to a running component for parameter access, messaging, error handling, logging, standard I/O stream handling, environment, and so forth. The container hides the details of the framework implementation from the component, and makes it possible for the component to be used in multiple modes of execution, e.g., via a host system CLI as well as via the framework. The container may be in the same address space as the component, or may manage the component as a subprocess using some sort of adapter. Containers are multithreaded and can simultaneously manage multiple components, or can be limited to a single component if the component is not thread safe.

- **Package Manager.** The package manager is a service which manages applications packages and their metadata. The package manager can dynamically load and unload packages, manage all parameter set instances and metadata, and execute tasks within a package. The package manager can also dynamically bind a package interface into the target environment, e.g., define Python bindings to all the tasks within a package when a package is loaded in a Python environment. All package manager functions are global for a user session providing runtime coherency in a distributed environment. Applications (as well as system software such as a CLI) interact directly with the package manager to introspect packages, display or edit parameter sets, or execute tasks.

- **DVM (distributed virtual machine).** The DVM provides basic process execution and messaging capabilities, handling most details of distributed execution, scalability, resource management and allocation, fail over, etc. transparently to applications. The DVM makes a distributed environment such as a cluster appear to be one virtual machine. The DVM can be “booted” stand-alone without any higher level functionality such as the container and package manager. A DVM console provides low level DVM-specific management and monitoring capabilities.

Applications normally see only the package manager interface, while components normally see only the container interface. In general neither applications nor components interact directly with the DVM, although it provides much of the basic functionality for both the package manager and the container. The DVM provides generic system-level functionality, hence is one of the most promising elements of the framework to exploit non-astronomy software for
distributed computing. In the simplest implementations the DVM can be eliminated or collapsed to a thin layer.

As we shall see in section 2.2, an individual system does not necessarily need to implement all these system elements, and in any case their complexity is largely hidden from applications software. Nonetheless they are all required for the most general use-cases.

### 2.1.4 Component

A **component** is a dynamically deployable processing module, e.g., *task* or *tool*, which can be invoked by the execution framework by executing the component within a container managed by the framework. A component implements a standard interface defined by the container it runs within, thereby making interchangeable components possible. Components generally do not know how they are being called, and are reusable in many contexts or via different modes of execution. Components are often written in compiled languages for reasons of processing efficiency, language stability, language neutrality, or reuse of library code, however high level languages can also be used to implement components, or a component interface might be provided for applications. Components may be dynamically loadable objects, may be statically linked with the container, or may be separate executables which are merely wrapped with a component adaptor (in which case no changes to the legacy or foreign component are required).

### 2.1.5 Class Library

Libraries are used to implement either applications or components. Class libraries are the most fundamental abstraction for structuring and reusing code, however the low level nature of a library limits how it can be used. Libraries are a lower level construct than the component because they lack any standard interface profile, the interface is defined directly in the implementation language, capabilities for dynamic deployment are limited, and binary-level linking prior to execution is generally required. The low level nature of a library can however maximize both efficiency and reuse in different contexts. Libraries can take many different forms, for example the client side interface to a service is often exposed to a client application as a library API.

Library code is normally used to implement components; however it is also possible to interface a compiled library to the application layer directly using an automated code binding tool such as SWIG if enhanced efficiency or finer grain control demands it.
2.2 Applications Architecture

One of the goals of the applications framework design is to provide a simple applications architecture, hiding most details of the framework from applications developers. In addition to simplifying science applications development this is important to provide greater flexibility in using and deploying applications, and to isolate applications from the details of the framework implementation and technologies, thereby allowing applications and the framework software to evolve independently.

In the following sections we examine how applications are put together and several different levels at which they can be used.

2.2.1 Host Execution

An important feature of the container is to permit host execution of components. By this we mean executing a component (task or tool) from the host OS level, such as from a Unix shell. This is the simplest mode of execution, eliminating all framework elements except the container. The DVM, package manager, packages, etc. are not used in this mode, and only individual tasks are executed.

Host execution is used for component development as well as for runtime execution of a task. It is important for development as tasks (especially file-oriented data processing tasks) can be developed as simple programs, run from the host command line or any IDE, and debugged with a conventional debugger without having to deal with the complexities of debugging a distributed system. Tasks which pass unit tests in this mode should execute correctly in a more complex runtime environment as well.

The developer sees only the container API and the API of whatever class libraries are used by the application. Tasks can be called from the command line
with a conventional CLI syntax, optionally augmented with a parameter set passed in as a file. Parameters can be set on the command line, passed in via a parameter set file, or set via built-in defaults defined as part of the parameter set or task implementation. Tool methods are handled in much the same way, using a CLI built into the container instead of the host CLI (this is necessary since “tools” are stateful).

Although the container (framework) interface seen by the developer appears simple, the container contains the necessary functionality required to execute the component via more complex modes of execution, such as in-process or distributed mode via the execution framework, including fully scalable modes such as a parallel task. It is also possible to execute a task via “host” execution, but have it connect to the framework to access framework services such as a data access object or logger.

Host execution also provides an important mechanism to maximize reuse of components in different contexts. Since a component can be executed via a standard interface which works on all platforms (host shell, execv, etc.), this mode provides the greatest flexibility for invoking components from different environments.

2.2.2 Single Process With Inline Container

In this mode the application layer and component container execute within the same process, with the application layer replacing the host CLI. Capabilities for distributed execution and scalability are limited in this configuration but the package manager and container are provided. For example, we might have a Python session as the application layer, possibly with CLI or GUI capabilities, with one or more containers and the package manager running as modules within the Python session in the same address space as Python.

Either an in-process container or a managed subprocess container could be used, or both. The in-process container (meaning that the container and the components it manages are in the same process) would run either Python
components or compiled (C/C++/FORTRAN) components in the same address space as the Python interpreter, enabling optimized calls from Python code to components and efficient in-memory sharing of parameter sets. The managed subprocess container would provide a limited capability for distributed execution by running external tasks as subprocesses of the Python session (PyRAF for example has a similar architecture).

This mode is important as it provides a simple process structure with efficient execution of both Python code and either Python or compiled language components within a single process, while also providing a simple mechanism for execution of tasks as managed subprocesses. Since this is done by using the general framework in a limited configuration it is easy to scale this to a full up distributed configuration.

### 2.2.3 Fully Distributed Configuration

In the most general case the framework supports fully distributed execution, managing multiple processes on one or more computers, using messaging to connect all components of the system. In a typical scenario using the full framework on a workstation or laptop we might have Python process used for application scripts and hosting a CLI, a Java process used for a GUI, and one or more containers running components. This is illustrated in the figure below.

Containers may execute either within the same address space as the application or in a separate process possibly on a separate cluster node managed by the DVM. The application runs in a single process with the framework managing execution of any components, distributing the work load among the set of nodes managed by the DVM. This more distributed case is little different than the single process case, the main difference being that the container runs in a separate process and messaging is used to link the application (via the package manager) to the remote container and component.
The advantage of distributed execution is that it is more scalable and modular, not trying to put all of a large system in a single process. Different languages such as Python, Java, and C/C++ are more easily used in a system given a multiprocess configuration, allowing the best language to be used for the job, or allowing external software to be more easily integrated. The primary disadvantage is that communication is less efficient than if modules share the same address space, but this is not a problem for many applications if they are designed with this in mind, and if a certain application requires tighter integration the integrated container can be used.

2.2.4 Task Structure
Tasks (or tools) may be written in any supported language. The best language to use depends upon many factors, for example C, C++, or FORTRAN might be best for numerical codes, whereas Java might be best for Web-oriented code, and Python for user scripting. Languages can also be combined, for example a task might be written in Python, using SWIG to bind in a C or C++ library which is used for the bulk of the processing. The framework does not care what language is used for a component so long as the component container interface is observed.

The following are all typical cases:

- A component written in C or C++ with a C container.
- A component written in Java with a Java container.
Applications Framework Architecture and System Interfaces

- A component written in Python (or Python+C), with a Python container. In this case the C container is probably what is actually used, with a Python binding. Components are written in Python but internally may call any Python code within the same Python session; hence Python code may be referenced both within a component and elsewhere in Python.

Components are automatically bound into the application layer by the package manager when a package is loaded, hence may be called directly from within an application. This is discussed further in section 0. A component may call another component provided the package manager API is available within the process containing the component. While technically feasible, inter-component requests can cause problems in a distributed system and in practice guidelines will be needed to restrict where this is permitted; better mechanisms for inter-component communication such as asynchronous messaging may be preferred.

Although a distinction is made between applications and components, in terms of framework interfaces they are closely related, and both may reside in the same process or share some of the same code. The distinction is primarily in how they are used, with components emphasizing programmatic reuse and implementing the component container interface, while applications emphasize top level usage addressing some specific use case. The basic guideline is that applications should see only components, the package manager API, the presentation layer interface if any, and other elements of the applications environment (Python, Java, etc.); components see only class libraries and the container interface and don't know how they are being used.

2.2.5 Legacy Code

[To be added]. Describe the strategy for interfacing legacy code to the framework. Within astronomy currently, most “legacy” (existing) code is either task-oriented (a program with parameters), or written in a high level language such as Python, IDL, or the IRAF CL. The task-parameter paradigm remains a valid way to modularize data processing software and carries over to the new architecture. The main change required to legacy code is to replace the “main” routine with what we refer to here as the container. The bulk of the code should require few or limited changes.

Note that we do not address code build systems here; while this is an important topic, it is language specific, and generally specific to the legacy code as well.

3 Framework Elements

In this section we move beyond the overall system architecture to propose a specific technical architecture which provides the specified functionality. This includes the following framework elements:
• Packaging. The packaging mechanism, including the form of an applications package, what it contains, package metadata, versioning, discovery, and interface definition.

• Component Container. Outlines the functionality provided by the different types of containers, and the elements of the component container interface, including parameter handling, environment, messaging, task execution, modes of execution, and so forth.

• Parameter Mechanism. Describes the parameter model, parameter sets, parameter set definition, common parameter set serializations, parameter management, and runtime management of parameter data by the framework.

• Messaging. Describes the messaging facilities defined by the container and provided for use by components and applications, including inter-tool messaging, producer-consumer events, synchronous and asynchronous requests, point-to-point messaging, streams, and so forth.

• Package Manager. Describes the functionality of the package manager, including loading and unloading of packages, management of parameter set metadata and instances, automated binding of task interfaces into the applications layer, and task execution via the package manager.

• Distributed Virtual Machine (DVM). Outlines the functionality of the DVM and the types of DVM, including basic low level process execution and process control, low level messaging, and resource management.

• Framework Services. Introduces some important framework services, for example for logging and for managing concurrent access to shared data objects.

The concept and function of each of these is detailed in the sections which follow. Finally we look at several common use cases and examine how they would be implemented based upon this proposed technical architecture.

3.1 Packaging
All applications are organized and managed as dynamically loadable applications packages (1.3.2). In this section we examine the structure and contents of a package; manipulation of packages by the applications framework is via the package manager and is discussed in section 0.
3.1.1 Logical View of a Package

The name of a package is specified in dot notation, e.g., “vao.voclient”. Packages are organized hierarchically, hence “vao.*” would refer to all packages within the “vao” package hierarchy. Package names are global within the runtime environment of the framework, but need not be unique, for example multiple versions of a package are permitted. Some TBD runtime mechanism such as a search path may be used to resolve multiple instances of a package. Packages are referred to in the runtime system by the full package name except for the use of the wildcard notation to load packages.

A package includes the following elements:

- The fully resolved package name in dot format.
- The package version in numeric dot format (1.2.3).
- The package title, a short description of the package.
- The vendor name or organization responsible for the package.
- The globally unique IVOA identifier for the package, if any.
- Any other package metadata, e.g., indicating the supported platforms, and any external dependencies.
- The package parameter set.
- Online documentation for the package.
- All applications, tasks, or other files forming the content of the package.

The package parameter set defines parameters which are defined globally for the package and any tasks or subpackages. Package documentation is by default in HTML, although other formats (e.g., Latex) are possible if the implementation permits. This documentation is intended to be used online; hence it should be self-contained for the essential content (external links to nonessential information are permitted).

3.1.2 Package Structure and Contents

Packages are intended to be easy for a user to find and install into a running system, hence it is important that they be “well packaged” software. A goal is for the user to be able to get a package as a file of some sort and install it into the system as a single operation, using the package manager.

To accomplish this end a package is distributed as a ZIP file. ZIP is used as it is a widely implemented standard including desirable features such as compression and a directory or manifest which permits random access to the file, e.g., to retrieve package metadata (Java also uses ZIP; a JAR or WAR file is a ZIP file).

The contents of the zipped package include the following:

- Directory or manifest in ZIP format.
Applications Framework Architecture and System Interfaces

- All package and task (component) metadata.
- All package data, including applications, components, and any files required by the package. This may include any libraries, JARs, etc. required by the package.
- Documentation for the package including all applications and components.
- Interface definitions for all applications and components. For tasks and tools this include parameter set and message definitions.

Packages are fully self-documenting, listing all included packages, applications, components, etc., and their interfaces. Since packages may have subpackages, a packaged installation may include many actual package objects in a single packaged installation file.

A package may support only a single platform, or a range of platforms, as indicated by the package metadata. All files required to run applications on a given platform (other than external dependencies if any) are included in the package distribution. In the case of compiled languages multiple binary objects (executables or libraries) may be included in the distribution. In the case of a language such as Python or Java the minimum supported language version should be specified. Operating system dependencies (Linux, Windows, Mac, etc.) may also require separate binaries. An alternative to supporting multiple platforms within a single package distribution is to prepare a separate distribution for each platform. Which approach is best is up to the package vendor.

3.1.3 Interface Definition
To achieve the goal of dynamic loading and use of arbitrary packages it is necessary that packages be self-describing, including describing the package contents as well as the interfaces of all software components within the package. To provide a uniform way of describing package metadata and interfaces, XML is used for both purposes (possibly also for documentation since HTML can be generated from XML).

[example of component interface specification in XML]

Interface definitions are required for all components (tasks and tools), parameter sets, and all messages used within a package. The interface of tasks and tools are defined in the same way, the only difference being that a tool has multiple methods. Parameter sets may be defined as distinct entities within a package, but are often defined implicitly as part of specifying the interface of a component, as the parameters of a task or tool method define the parameter set for the method. The interface of a component also specifies any messages produced or consumed by the component.
3.1.4 Package Discovery

Use of a package by the framework is independent of how packages are initially discovered. While packages may be manipulated directly as files, packages are often stored in, and retrieved from, repositories of some sort. A repository can be queried directly to discover what packages it supports, e.g., by querying based upon the package metadata. Global discovery of packages is also possible by registering packages in the IVOA registry, and using registry query techniques to search for or browse registered packages.

3.2 Component Container

While the component-container mechanism defined by the applications framework may sound complicated, the same basic mechanism has been common in astronomical applications for decades. Any program which can be run from the host command line and which does some standardized parameter processing implements a form of what we call the container – a “main” routine which provides a standardized external interface for an integrated suite of programs (IRAF, AIPS, MIDAS, Starlink, GYPSY, MIRIAD, CASA, CIAO, Terapix, etc. all have something like this). The container facility we describe here is the same basic mechanism but allows greater flexibility in how a task is called.

The container mechanism is important mainly for two reasons: to provide a standardized interface for running components (tasks and tools), and to provide important basic capabilities to task code, such as management of the input parameter set and logging. At the same time one can add important functionality without significantly increasing the complexity as seen by a task developer.

The capabilities provided by the container mechanism include the following:

- **Task execution.** Components (tasks or tools) may be executed directly from the host system, e.g., from a host operating system command shell, from a debugger, or from some foreign framework such as pipeline system or batch processing system. Tasks may also be executed from the applications framework described here, including from a container which is embedded in a high level language or environment such as Python.

- **Parameter management.** The container provides built-in facilities for management of task parameters, including processing of parameter set files, processing of parameter sets passed via a framework, processing of parameters passed on the host command line, and provision of an API which the client program (component) can use to access parameters. An environment variable mechanism is also provided which is capable of maintaining host-compatible environment variables in a distributed, multi-process or cluster environment.
Applications Framework Architecture and System Interfaces

- Messaging and logging. Messaging capabilities are essential in a distributed environment to enable applications to execute in a distributed, scalable fashion. Messaging includes synchronous and asynchronous requests, producer-consumer events, property change events, high performance binary streams, and logging capabilities to monitor task execution or errors.

- Built-in help and introspection. In an open system where software can come from many sources it is essential to provide some facility for built-in help (to allow a human to discover the capabilities and interface of a program), as well as for programmatic introspection (to allow framework or client software to discover the capabilities and interface of a container or component).

In addition to providing standard facilities to a component such as parameter management and messaging, the container isolates the component from the details of the execution framework it is called from. This is essential to allow different types of frameworks to use the same component. At the same time the container simplifies and abstracts the environment seen by a component developer. To a first approximation the component developer sees only the container API and the APIs of the class libraries used by the component. A client application does not see the container at all; it sees only the task it is calling.

3.2.1 Types of Containers
Various types of containers are possible although all share the same interface to the framework.

- Integrated Container. The container is implemented as a library which executes in the same address space as the components it manages. All container capabilities are potentially available to the task since the container is fully integrated. Components (tasks or tools) may be statically linked with the container, or may be dynamically loadable. Components may be thread safe and capable of executing concurrently within a multi-threaded container, or may disallow multi-threaded execution in which case the container will run only a single component at a time (in this case it is still possible for the container itself to be in a separate thread than that used to run the task). For many numerical tasks a new container (process) may be spawned to run a single task.

- Managed Subprocess Container. In this case the component executes as subprocess of the container, using a custom adapter to call the component complying with whatever execution protocol the component defines. This makes it possible to call almost any external program merely by writing an adaptor module, without requiring any changes to the external program. Execution may be less efficient than in the case of the...
integrated container, but this may not matter for compute intensive tasks. Advanced container capabilities such as the full range of parameter semantics and messaging may not be provided, but basic capabilities such as simple parameter passing, process control, and capture of the standard output and standard error output should be possible (the managed subprocess is the converse of host execution as described in section 3.2.2).

In both cases the container is implemented as a library. The container may execute as a separate process, or on a thread within an existing process, e.g., within an applications language such as Python or Java. In the case of a separate process a custom “main” is used to pass the process arguments and environment to the container.

In the case of an embedded container, i.e., a container which is embedded into the same process used for the application layer, e.g., a Python session or Java JVM, then it becomes possible for components to execute within the same address space as the application. This essentially eliminates the DVM (eliminates distributed computing overhead and latency) for calls to components which execute within the embedded container, and makes possible certain optimizations provided the container and package manager are compatible implementations. While less general and scalable, this makes tighter integration of the application and component possible, e.g., for “plug-in” type applications.

For the applications framework we will probably require two containers, one for compiled languages and one for Java. The compiled language container will be written in C or C++, and would be used to execute compiled language components (C, C++, FORTRAN, SPP, etc.). The Java container would execute Java components. In either case the managed subprocess container could execute anything. To embed the container in a language such as Python a binding layer would be used. In principle it is possible to mix languages at a finer grain level (Java calling C or vice versa) but this has proven problematic in the past; a distributed multi-process approach is likely to be more reliable. Hence for example, a Java application could call a component written in any language so long as the container is executed as a separate process via the DVM.

3.2.2 Host Execution

In the case of host execution a task is executed at the “host” level via the container, passing arguments on the command line as strings. This allows a task to be called by a host shell or by any other software which can “exec” the process, e.g., a debugger, an IDE, or a foreign framework of some sort. For example, the shell command line

```
sh> foo.e imheader image.fits @imhead.par
```
would execute the task “imheader” within the process “foo.e”, directing the task to print the header of the image “image.fits”, using the parameters from the file \texttt{imhead.par}. What happens in this case is that when \texttt{foo.e} is executed at the host level, control transfers to the custom process “main”, which passes the argument list to the container and tells it that it is executing in host mode. The container parses the argument list, performing parameter resolution against the task parameter set (command line arguments override pset defaults), then calls the \texttt{imheader} task. Once the task completes the process exits.

One could go one step further and make each task contained in the \texttt{foo.e} executable available as a host command, in which case the host command required to execute the imheader task could reduce to the following:

\begin{verbatim}
sh> imheader image.fits
\end{verbatim}

where we have omitted the parameter file reference, causing the task to fall back on built-in parameter defaults. Since in this case the task is being called from a host shell, all shell capabilities such as i/o redirection, pipes, background execution, etc., would be available.

Additional parameters on the command line could be used to pass arguments to the container, as in the following example:

\begin{verbatim}
sh> foo.e -C joe123@localhost imheader image.fits @imhead.par
\end{verbatim}

In this case the argument “-C” would tell the container to establish a connection to the framework session “joe123” on localhost, causing the container to attempt to connect to a running framework session at startup time. This would allow the container and task to interact with framework services or other components (tasks and tools) during runtime, even though the task was not started by the framework.

Other container capabilities such as introspection of interfaces, or access to built-in documentation could also be accessed at the host level via command line examples. For example,

\begin{verbatim}
sh> foo.e --help imheader
\end{verbatim}

would print the online “help” documentation for the \texttt{imheader} task. The command

\begin{verbatim}
sh> foo.e --help
\end{verbatim}

would print general documentation identifying the container and its contents, as well as providing help on how to interact with the container to access additional documentation or execute tasks.
These examples are for a Unix-based host shell. For a host such as Windows the command syntax might be different, but the executable content and semantics would be the same.

### 3.2.3 Parameter Management

One of the most important functions of the container is parameter management. Parameters and parameter sets (psets; see section 3.3) are used to control the execution of a task and return output from a task. Task code uses a language-specific API provided by the container to access parameter data.

Parameter data can be input in a variety of ways when a task (or tool method) is executed. Parameters may be input:

- On the command line, in the case of host execution.
- In a text file, in the case of host execution.
- Via the runtime framework when a task is executed.

In addition to input parameters the container implements a built-in parameter set class (parameter set definition and any associated methods) for each task it manages. This is used to define default values for parameters, to constrain the values of parameters, and to provide help and introspection capabilities so that a client or the user can query the parameters of a task (while the package manager also has this information, it also has to be provided at the container level to support host execution).

When a task is executed the container may perform **parameter resolution and verification**, to supply any missing parameter values and verify that the parameters have valid values, before executing the task. This is done as follows:

- A parameter set instance is created from the built-in pset class.
- Any input parameters provided (see above) overwrite the default values.
- Parameter set verification is performed using pset class-specific logic.

Parameter processing is enabled for host execution, but is optional for execution via the framework, as parameter resolution and verification may already have been performed, e.g., via the package manager.

If an invalid parameter value is found the task does not execute and an error is returned. Of course a second level of parameter verification will occur within any well written task, but performing parameter validation prior to actually executing a task can avoid problems with task cleanup if parameter errors are found during task execution. It is also more efficient, especially for a parallel task.

The container provides a **parameter set API** as part of the API defined by the container. The component (task or tool) uses this to access parameter values at...
Applications Framework Architecture and System Interfaces

runtime. In general task or tool method parameters are not input as function arguments, rather they are passed via the parameter API. The capabilities provided by the parameter set API include the following:

- Create or open a new pset instance (the main input pset is always available).
- Get a parameter value given the parameter name.
- Set a parameter value given the parameter name.
- Add a new parameter not defined as part of the base pset (for output).

While a task always has one main input parameter set, in general there may be any number of input or output parameter sets, provided these are declared as part of the component interface so that they may be automatically marshaled or disposed of by the framework. Input psets may be shared, e.g., multiple tasks may share an algorithm or calibration pset. Output parameter sets are automatically returned to the client at task termination, and are available for editing during task execution (if real time interaction is required messages may be used during task execution). While parameter values are normally fixed at the time of task execution, this is not necessarily the case, i.e., a “get” of a parameter could cause some action to generate the parameter value, such as a prompt, suspension of a task waiting for input, or input of the next parameter value from a list.

In host execution mode parameters and parameter sets are normally input or output in text format. In framework execution mode text format is the default, but other forms of serialization are possible provided the sender (e.g., package manager) and receiver (container implementation) agree upon the format. This provides scope for optimization, for example a pset instance could be transmitted in a binary format or passed via reference (5.2).

3.2.4 Messaging and Logging
Various types of messaging are available during task execution to allow the task to communicate with external applications or components. These include:

- Standard input and output streams (stdin, stdout, stderr, etc.).
- Special purpose high performance point-to-point data streams.
- Producer/consumer (broadcast) events.
- Synchronous and asynchronous requests to other components.
- Property change events.
- Logging messages or events.

A component may be either a data source or sink for any of these messages. Messages produced by the component are defined as part of the component interface. Typical uses of messaging include linking a GUI to a data processing task to display the state of processing during execution, externally monitoring
and controlling execution of a task, accessing a data access object from within a task, or generating log messages to monitor task execution. The container defines a standard messaging API and messaging model regardless of how messaging is implemented within the framework. Messaging is discussed in more detail in section 3.4.

### 3.2.5 Components

As discussed earlier in section 1.3.3, a *component*, in framework terms, is a *task* or a *tool* depending upon whether the component has more than one method. Task components may be either serial or parallel. Data processing components are most commonly implemented as tasks, whereas the tool interface is used for stateful objects which may run indefinitely accepting client commands, such as a data visualization tool or data access object.

In terms of implementation a component is just some class code which has a standardized interface. The standardized interface allows the component to be run from any compliant container, making it possible to easily interchange components or component packages, and to call components from applications with a standardized client interface including the parameter mechanism.

Components may be written in any language provided a container is available, for example C/C++/FORTRAN, Java, Python, and so forth. Adding a component interface to an existing program typically only requires writing an adapter layer which implements the standard component interface and converts method calls into calls to the underlying class code.

Components may be dynamically loadable or may require static linking, and may be thread safe and capable of running in a multi-threaded container, or may require single threaded execution (meaning only one component may execute at a time within the container). Much legacy code for example is neither dynamically loadable nor thread safe. Most Java and Python code and some C/C++ class code (where the effort has been made) is capable of both dynamic loading and multi-threaded execution. For most data crunching this is not an important issue so long as tasks execute in a distributed fashion.

A special case is the managed subprocess, where a foreign program is run from the managed subprocess container (3.2.1) using an adaptor to implement the component interface at the process rather than class level. Typically this involves generating a custom parameter file or command line to execute the foreign program. In this case the “component” can be any existing program and no changes to the external program are required.
3.2.6 Container-Framework Interface

Thus far we have been concerned primarily with the interface between the component and the container, but the container actually has a second primary interface, the interface between the container and the rest of the execution framework, most notably the package manager (2.1.3, 0). For simplicity we call this the interface of the container to the framework, i.e., the container-framework interface. This interface is used whenever a task is run from the applications framework. It is not used for host execution.

An important aspect of the container-framework interface is that, for applications packages from different sources to be interchangeable, this interface needs to be specified down to the level of the wire protocol, which must therefore be an open standard. If we could get everyone writing applications packages to use the same container implementation this would not be necessary, but this is probably unrealistic for a large open system where applications packages may come from many sources. To achieve a fully open system it is necessary that applications packages can be developed independently by different groups, be distributed in binary form (linked with the container) and used sometime later on a remote system with any compliant framework implementation.

On the other hand it is desirable to be able to make use of existing messaging and HPC libraries or frameworks at the core of our applications framework, especially when it comes to the full up distributed, scalable use case. This is only possible if the framework interface is defined as an API, because any existing message bus technology will define its own wire protocol for use internally.

To meet both of these requirements we need to define both an API and a wire protocol. The following two interface strategies are required:

- **API.** With the interface-as-API approach the interfaces required are the container API (on the component side) and the package manager API (on the client side). Everything else in between can be considered part of the framework implementation. In the API-based approach both the container and the package manager are part of the same implementation, sharing the same underlying framework implementation. This makes it possible to fully optimize the data path between the client application and the component, and make full use of existing messaging or cluster processing technology for the core framework. The complexity of the framework implementation may vary considerably depending upon the capabilities provided, e.g., for scalability.

- **Wire Protocol.** In this approach we define down to the level of the wire protocol how to talk to the container, to query container metadata and capabilities, pass parameter information, execute tasks, pass messages during task execution, control process execution, and so forth. To avoid
having this interface become overly complicated only the most important capabilities are provided and advanced features such as optimized data transfers may not be available.

At runtime when a container is executed to prepare to run a task, the package manager will determine from the package metadata whether or not the supplied container has a compatible implementation. If so, optimized execution and enhanced framework capability (such as fully distributed, scalable execution with efficient binary message transport) will be possible. If not, the framework (both package manager and container) will fall back on the open wire protocol and basic capabilities will still be provided, providing interoperability of packages, framework elements, and applications from different sources.

### 3.2.7 Minimal Container

Application packaging and the container are the only parts of the framework included with exportable applications packages. Specification of a minimally compliant container is important to ensure that applications packages from various sources meet basic requirements for interoperability.

A minimally compliant container should provide at least the following capabilities:

- Query container capabilities.
- Query list and type of components.
- Host execution of tasks.
- Framework execution of tasks via open messaging protocol.
- Parameter input and output in text format.
- Parameter resolution and verification.
- Full container API (parameters, messaging, logging).

It is TBD whether a minimal container needs to actually implement everything in the container API, but the full API should be provided so that components can link against the container implementation. Calling an unimplemented container API routine at runtime might result in a runtime error, or the call might simply be ignored.

A fully compliant container will implement the complete container API and semantics and may optionally support more advanced framework protocols and capabilities.

### 3.3 Parameter Mechanism

As introduced in section 1.3.4, task and tool operations use a parameter set (pset) to drive execution and optionally return a result. Parameter sets may also be used to control processing algorithms which are shared by multiple tasks, or to describe simple data entities conveniently within the runtime framework.
environment. The parameter set mechanism has a long history within astronomy, providing a standard mechanism for describing the interface to a task separately from the task itself, allowing generic tools to be used to query, manipulate, and verify the parameter set of any task. Most of the legacy astronomical data processing systems, FITS, and VO all have a concept of a simple parameter mechanism used to define the attributes or parameters of an object, be it the input parameters of a data processing task, the attributes of a WCS object, or the parameters of a VO data access service.

A \textit{pset} is a collection of \textit{parameter} objects, which are typed objects with a name, a value, and additional attributes such as a description and UCD or UTYPE tags which may provide additional information about the semantic content of a parameter. Since psets and parameters are objects, they can have associated methods, including those for content validation and for any additional functionality specific to the sub-class being defined.

Parameters and parameter sets in themselves are generic containers and have little intrinsic meaning; they merely provide a standard way to represent keyword-value type information. The meaning of a parameter or parameter set is defined by how it is used, by relating it to some object such as a task, service operation, algorithm, data model, and so forth.

In what follows we make a clear distinction between the parameter \textit{model}, which defines the semantic content of a parameter or parameter set, and the techniques used for parameter set \textit{definition} and \textit{serialization}. There can be any number of ways that a parameter set is defined or serialized for external transmission or storage, but the semantic content of the parameter set is the same in all cases.

### 3.3.1 Parameter Model

#### 3.3.1.1 Parameter

A \textit{parameter} is a simple object with a name, a value, and other standard metadata used to say something about the parameter, as in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>This is a simple identifier, unique within the controlling namespace (e.g., parameter set).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datatype</td>
<td>The storage type such as boolean, int, char, float, etc.</td>
</tr>
<tr>
<td>Value</td>
<td>The scalar or vector value of the parameter. At this level only simple 1-dimensional arrays are provided. NULL is a legal value, indicating that no value has been set.</td>
</tr>
<tr>
<td>Description</td>
<td>A one-line description of the parameter.</td>
</tr>
<tr>
<td>Unit</td>
<td>The unit specified as a string in a VO-compliant syntax (optional).</td>
</tr>
<tr>
<td>UCD</td>
<td>The type of physical quantity which the parameter represents (optional).</td>
</tr>
<tr>
<td>UTYPE</td>
<td>The field of a data model associated with the parameter (optional).</td>
</tr>
</tbody>
</table>
Additional parameter attributes may be defined by sub-classing the base parameter class to add attributes specific to a particular type of parameter. For example, a parameter used in a user interface might have a “mode” parameter to specify whether the parameter is normally hidden, or a service might have a “std” parameter specifying whether the parameter is a standard part of the interface or is a custom extension added by a particular service.

The UCD tag may be used to specify the type of quantity which the parameter value represents. UTYPE identifies a parameter with a field of a data model, and makes it possible to store the attributes of an object in a flat set of parameters, even if the data model has some internal hierarchical structure.

When using a pset to store a data entity or return output results, UTYPE may be used to enhance the parameter model, e.g., to associate an error estimate with a numerical value, using two or more associated parameters as in the following example:

<table>
<thead>
<tr>
<th>Name</th>
<th>UTYPE</th>
<th>UCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redshift</td>
<td>Derived.Redshift.Value</td>
<td>src.redshift</td>
</tr>
<tr>
<td>RedshiftStatError</td>
<td>Derived.Redshift.StatError</td>
<td>stat.error;src.redshift</td>
</tr>
</tbody>
</table>

The core attributes of a parameter as shown here are the same as in VOTable. This is intentional, as a parameter set, a VOTable, and a FITS header are all flat table-like namespaces used to store keyword=value data. Ideally parameterized data (including parameterized data models) should be able to move easily between any of these three common astronomical representations.
3.3.1.2 Parameter Set

A parameter set is a logically related set of parameters, typically used to represent an interface of some sort, or data model. For example, the input parameters of a task are an interface and might be represented as a parameter set. Parameter sets may also be generated by a task and returned as output. A data entity with an associated data model can also be represented as a parameter set.

A key aspect of the parameter set concept is that a pset stands on its own as a data object. The input parameter set for a task is merely associated with the task; it can be inspected or manipulated separately as a full-fledged object without running the task. This allows parameter sets to be operated upon with generic tools regardless of how a parameter set is used.

As an object, a parameter set has a defining class, and may have any number of pset instances. The runtime instances only need to store the actual parameter data; all class metadata and methods are provided by the pset class definition. All parameter sets are derived from the base parameter set class, which provides standard methods for operating upon generic psets (but probably no actual parameters!). All actual parameter sets are derived from this base class, and every task, tool method, or data entity has its own specific parameter set class. These customized (subclassed) parameter sets define the standard set of parameters associated with the specific interface or data model. Parameter set subclasses may define custom methods as well.

3.3.2 Parameter Set Definition

A parameter set definition may include any of the following:

- The parameter set class name and superclass.
- Any global parameter set metadata (description, version, etc.).
- Any number of parameter definitions.
- Default values for parameters, if any.
- Definitions of any parameter constraints or dependencies.

A parameter constraint defines the legal values for a parameter, e.g., minimum or maximum values, or an enumerated list of acceptable values. Parameter dependencies may optionally be specified, for example the value of a parameter may be an expression which references other parameter values, including simple indirection where a parameter takes its value from another parameter (defaulting to a package parameter for example).

While specifying simple parameter constraints is useful for basic parameter verification, full verification of a parameter set can only be done by class code associated with the parameter set itself. The most common case of this is when a task is run and it custom verifies the value of its input parameters as they are
used during execution, reporting any errors. Another possibility is for the pset itself to have an optional validate method which can be called by the system to validate the parameter set.

Parameter set definition can occur in many ways. Some of the most common are the following:

- When a package is loaded, reading the package metadata and creating in-memory objects for all parameter set classes defined therein by the XML-encoded package and task metadata. This may include package psets, task and tool method psets, and psets for any pset-based data entities defined by the package.

- By loading a parameter set definition stored externally in a file. This is essentially a schema, and includes all the metadata necessary to define a new parameter set class.

- By runtime interaction with the package manager. In this case the parameter set and all its parameters are created at runtime by making calls to the package manager API.

It is also possible for a parameter set instance to be self-defining, if it contains the parameter set metadata in addition to the data values. A parameter set serialized as a VOTable for example would probably include both pset metadata and the actual parameter data values.

Making it possible to add new ways to create a parameter set object is important to allow integration of the parameter mechanism in new contexts. In OOP terms, this is equivalent to defining a new type of constructor. In all cases the parameter set data model and semantic content is the same.

### 3.3.3 Parameter Set Instance

There may be any number of runtime pset objects (instances) created from the same parameter set class.

For example, to run a task one might create a new instance of the input pset, set certain parameter values, use this edited pset to run the task, and then throw the instance away at task termination. If multiple instances of the task are run concurrently they would each have their own instance of the input pset. Alternatively, since a pset stands alone as an object and input psets are read-only, a single pset instance could be created and then used as input to any number of task instances. Similarly, if a task produces an output pset and multiple instances of the task run concurrently, each would produce a separate instance of the output pset. It is up to the controlling application to decide what to do with input and output pset instances.
A parameter set instance can be created in many ways including the following:

- Using the package manager API. The package manager globally manages parameter set classes and instances.

- In the applications language, using a language construct, if psets are interfaced as full class objects in the target language. For example

  ```java
  Pset p = new ImheaderPars("dev$pix", long=true);
  ```

- As the output of a task.

The contents of a new parameter set instance depend upon how it was created. By default one gets whatever the default constructor creates: normally this will provide default values for many of the parameters, setting others which cannot be defaulted to NULL. Other constructors, such as a load-from-file constructor, might set a more specific set of parameter values. The package manager API can be used directly to query and set individual parameter values, as can the applications language if psets are integrated into the language as objects. A pset can also be copied, duplicating all parameter data in the new copy.

A pset instance need store only the parameter data, that is the actual parameter values; one copy of the pset metadata is stored in the pset class regardless of how many pset instances there are. For an in-memory pset instance parameter data may be stored internally in binary format. Hence if a parameter is array valued the array data may be stored and operated upon as a binary array, permitting efficient operation upon the data.

### 3.3.4 Parameter Set Serialization

The package manager is responsible for the creation and management of new instances of parameter sets, specific for each task or tool operation. The parameter set instances can be passed by reference or by value.

In the case of a pset passed by reference, the recipient is passed a reference to an in-memory object managed by the package manager, and the form of the reference address depends on the framework execution mode. In the case of single processes with inline containers (see section 2.2.2) the parameter set reference is a simple memory address or pointer. In the fully distributed configuration the reference address depends on the framework implementation used.

In the case of a parameter set passed by value, then a suitable serialization of the related pset should be provided depending on the wire protocol adopted by the framework implementation. In most cases all that is required is a *keyword*
**3.3.5 Parameter Sets for Parallel Tasks**

In the case of a parallel task one has a single task for which the framework runs M instances (processes) concurrently on N compute nodes, each of which has S slots (processors or cores). It is up to the framework to allocate tasks to nodes or slots, but the application needs to know M, as well as the rank of each task, with rank=0 being the controlling task if any. Typically the task will distribute portions of data to each of the M instances, with the process rank telling each task instance what portion of the data to work on. It is up to the framework to tell the application how many processes it is allowed to run, depending upon the available resources.

In general, allocation of portions of the problem to different processes has to be left up to the specific task as this is algorithm dependent. In a trivial case (assuming MPI semantics), if the task needs to perform the same operation on 512 datasets and 1 compute process is allocated, two instances of the task will be run, one with rank=0 and the other with rank=1, with the rank=0 instance running on the head node and being the master (controller), and the rank=1 instance being the slave, or compute process.

The single slave or compute process will need to process all 512 datasets in sequence while the rank=0 master monitors the operation, controls the processing, or whatever is appropriate for the specific processing algorithm implemented by the task (for a completely data parallel operation the rank=0 instance might not even be needed). If the same task is run on a cluster and 64 compute processes are allocated, again a single master will be run, and the 64 slave instances of the task will run concurrently, with the instance having rank=1 sequentially processing datasets 1-8, the instance having rank=2 processing datasets 9-16, and so forth. The master and the M slaves are instances of the same task, with the rank telling each task instance the role it is to play, and what portion of the problem it is to work on.
In the case of a parallel task we have two options:

- Each task instance gets the same task pset and sees the same parameters. Processing is entirely controlled by the process rank and number of worker task instances. This is suitable for parallel algorithms where processing is controlled by the master, communicating with slave processes using messaging at runtime (this is traditionally how a message-based parallel processing library such as MPI works).

- Each task instance gets the same task pset, but it is a **parallel pset**, with the process rank defining the slice of the pset seen by the task instance. In this case data segmentation is done when the pset is composed, defining the parameters seen by each process rank. The work to be done by each worker process is fully specified by the rank-indexed slice of the pset it sees. In this case the rank=0 instance can probably be omitted and the master is the top level application which computes the parallel pset. The M worker instances run concurrently, but independently, as determined by their input parameters.

In the case of a parallel pset the task is completely data parallel (at the process level), with the operation of each task instance controlled by the rank-indexed slice of the pset seen by each task instance. All task instances run concurrently, just as a normal sequential task would run, with processing controlled by the rank-indexed slice of the pset seen by each task instance. Note that while the task is fully parallel, actual processing operations typically involve executing a number of tasks in sequence, and only some of these tasks may be parallel to this extent.

A parallel pset is just like a simple scalar pset except that it is a **table** with one row for each process rank; each row is essentially the same as a conventional scalar pset. Parameters which are the same for all ranks can be expressed as fixed values for the entire table (parallel pset). Parameters for which the value is rank-dependent are stored as table columns. A field of the table can be either scalar or vector valued as for a simple scalar pset. In the case of a parallel pset the container can apply the process rank to slice the pset transparently to the task, so that the task sees what appears to be an ordinary scalar pset.

### 3.4 Messaging

Messaging is an essential element of any distributed system. Without messaging a distributed system cannot function, as messaging is needed to spawn remote processes, execute and monitor remote tasks, drive real time GUI displays, talk to services, mediate concurrent access to data, and for many other functions.
3.4.1 Framework Messaging Facilities

Within the applications framework, messaging is used mainly at two levels: by applications to link the components of a distributed application during execution, and as part of the system framework itself to support the functioning of the framework.

Framework messaging facilities are provided both for applications and for the functioning of the framework itself, and include the following:

- Applications-level messaging is provided by the container and package manager to provide communications between components and between an application and any components it calls.

- Messaging is also used internally within the DVM to manage distributed execution (e.g., to spawn, monitor, and control container processes) and to execute tasks within containers.

All framework-related messaging done by a component should use the messaging facilities integrated into the container. The same messaging facilities and API are provided by the package manager for use by client applications to communicate with components during the execution of a task or tool. If the container and package manager share the same implementation, normally the same messaging library will be used for both facilities.

A DVM provides lower level messaging capabilities, however these are specific to the particular DVM implementation and should not normally be used by applications or components, less the application become dependent upon a particular implementation of the DVM. An implementation may use the DVM messaging capabilities as the basis for applications-level messaging, by implementing the container or package manager messaging API as a layer on top of the messaging facilities provided by the DVM.

3.4.2 Types of Messaging

If we examine all the different use-cases for messaging we find we need all of the following types of messaging:

- **Synchronous and asynchronous requests.** A request is essentially just a command which is sent to a remote component or service. An RPC (remote procedure call) is a synchronous request. Asynchronous requests are often preferable in a distributed system.

- **Producer/consumer events.** In this case a component produces events, usually whenever the state of the component changes, or some event occurs. A client subscribes to classes of events of a particular type or from a particular component. Events are broadcast to all subscribers.
Such events are always asynchronous, and the producer does not know or care if any external clients are monitoring the events it broadcasts.

- **I/O streams.** This is a point-to-point stream of data, including the standard I/O streams of a process (standard input, output, and error), and custom high performance data streams between two components.

- **Property change events.** This case is not really a fundamental type of messaging, rather it is a message pattern based upon the producer/consumer event mechanism. In this case the state of a component is represented as a set of properties, and an event is generated whenever a property changes in some way. A client subscribes to property change events to monitor the state of a component. This can be combined with asynchronous requests to provide a fully asynchronous alternative to conventional synchronous RPCs.

- **Logging messages.** As with property change events, logging is a message pattern based on the producer/consumer event mechanism (but it is important enough to warrant special support). A standard logging API is used to generate debug, log, or error messages, which are broadcast to any subscribers. A logger component subscribes to all log messages and can display messages during program execution, or save log messages to a file or database for later analysis.

It appears that just about any messaging use-case we have for a data processing and analysis system can be satisfied with these messaging primitives (in particular the first three, maybe just the first two) provided there is sufficient flexibility in the messaging semantics, such as what type of payload is permitted in a message.

### 3.4.3 Messaging Model

At the applications level the messaging model should be defined as an abstraction independent of the underlying transport, so that the messaging semantics and message content are the same regardless of what technology is used for the messaging framework. This makes it possible to use a standard messaging system (MPI, PVM, D-BUS, XMPP, ActiveMQ, Beep, etc.) for lower level messaging without affecting the messaging semantics seen by applications.

The required messaging model should include the following characteristics:

- **Separation of message delivery from content.** A message type (MType) and other message metadata are required to describe the message sufficiently for delivery purposes, e.g., so that a client can subscribe to producer/consumer messages matching a certain message type pattern, or so that a client can use the MType to determine how to dispose of an
incoming message in an automated fashion. The message content can then be anything, and will in general depend upon the type of application and the message patterns it uses. Both text-based and binary message content are needed for different use-cases and transport protocols. We wish to standardize the message description and delivery semantics, but allow the message content to vary depending upon the class of application.

- The send-message primitive should support both a send to a specific component, and a broadcast where the message is delivered to all subscribed components. The basic send primitive should be asynchronous and should support message buffering (otherwise efficient streaming of data is not possible, or senders will block when attempting to send a message).

- The receive-message primitive should support buffering of incoming messages, and return the next message from the incoming message queue.

- Flushing of buffered messages should be possible to permit synchronization when this is needed.

Message buffering is a necessary feature for high performance messaging to support message streaming (of all message types, not just “I/O streams”), and to avoid having a process block while waiting for a message to be delivered. Synchronization between two or more processes requires flushing the message buffers, i.e., processing all buffered messages until the buffers are empty (in the most general case this is the same as a “barrier” in MPI). Messages originating from a single process are guaranteed to be delivered in order, but there is no guarantee on absolute message order if multiple processes are simultaneously sending messages. Message delivery is guaranteed or an error will be reported, however this does not mean that a recipient will actually do anything with a message; that is a matter for the applications messaging protocol.

The message type (MType) is a simple string identifying the message class, e.g., `image.display` or `image.notify.frameChanged`. Message subscription can then be class-based using simple patterns.

Subscribing to and unsubscribing from message classes is dynamic. A component may dynamically connect to and disconnect from a running system, and may subscribe to and unsubscribe from messages at will.

### 3.4.4 SAMP Messaging Protocol

Within the IVOA work is in progress on a high level applications messaging protocol, the Simple Applications Messaging Protocol (SAMP). Initially this
mainly targets high level, loosely coupled messaging between desktop tools: requests are typically at the level of entire objects, for example “display this image” or “display this table”. However SAMP 1.0 already has many of the characteristics required of a general applications messaging protocol as described herein, including Mtypes, asynchronous messaging, and both directed and producer/consumer messages.

A version 2.0 edition of the SAMP protocol is planned which should be general enough to support basic messaging as described in the previous sections. If so, SAMP 2.0 could provide the open wire protocol required for basic interoperability of separately compiled and linked applications packages at the “wire” level. More sophisticated messaging implementations would still be needed for use-cases such as high performance distributed computing, and would probably be based upon some existing messaging infrastructure. These would provide API-level interoperability as described in section 3.2.6, with a fall-back to the open wire protocol when necessary.

SAMP 1.0 provides a messaging system based upon a broker service (Hub) used to deliver SAMP messages. SAMP messages are encoded as maps (sets of keyword-value pairs) of standard keywords including a string which defines the meaning of the message (the MType) and a map which defines the message content expressed as a parameter set. Additional keywords are also permitted but might not be used by all receiving applications (e.g. the keyword may define a non-critical option available in one form of image display application but not in another). The structure and semantics of the set of parameters in a message are defined by the message type. The message parameters are not strongly typed.

Three message delivery patterns are defined:

- Asynchronous call/response
- Synchronous call/response
- Notification

With the call/response pattern the sender application receives a response from the target application or applications (SAMP allows requests to be sent to multiple components). The response may be delivered either synchronously or asynchronously, using a message-ID to link the response back to the original request. This pattern thus supports both synchronous and asynchronous RPC.

The notification pattern is an asynchronous event notification where the sender does not receive a response.

To receive requests (calls), components must declare the range of Mtypes they support. To receive notifications, applications or components must subscribe to the classes of MType they wish to receive.
SAMP defines a standard low level implementation, the “Standard Profile”, which uses XML-RPC as underlying wire protocol (hence SAMP 1.0 is text-based). The SAMP specification requires all implementations to support the Standard Profile, however additional profiles may also be co-implemented, allowing several profiles to be transparently supported in a heterogeneous messaging environment.

3.5 Package Manager

The package manager is a framework service used to manage applications packages, including any associated metadata and parameter sets, and any tasks and tools included in the package. Tasking support includes execution of any tasks provided by the package. The package manager is the primary client interface for interacting with packages and tasks. Client applications may interact directly with the package manager via the package manager API, or indirectly by accessing objects (tasks, psets) managed by the package manager which are interfaced directly into the applications language and environment.

The package manager service is globally managed for a framework session. Multiple client applications, possibly running in separate processes, may simultaneously access the package manager to access parameter set data or execute tasks. As with messaging, parameter set data is globally available within a session and may be used for communication between multiple components or applications. Package and parameter set data may be persistent across sessions.

3.5.1 Managing Packages

Package management includes the following functions:

- **Deploying packages.** The package is deployed (installed) into the system. Summary package metadata is extracted and the package is placed on the list of loadable packages. The package ZIP file may or may not be unpacked into file storage depending upon the implementation of the package manager (normally it will be unpacked). Packages may also be undeployed. Multiple versions of a package may be present in the system.

- **Loading packages.** All package metadata is loaded into the package manager, including all parameter set classes and task definitions, and all tasks are readied for execution. Some package-defined parameter sets may be instantiated (e.g., the package parameter sets). Packages may also be unloaded, releasing storage in the package manager.

Package installation or deployment may be done at any time, including during system execution, but does not have to be a runtime package manager function;
it could be done by other related system software, such as a package installer. Package deployment only has to be done once and then the package may be used any number of times in any number of sessions. Packages (at the file level) are shared and may be used simultaneously by multiple user sessions.

Packages must be loaded at runtime before any tasks or parameter sets can be accessed. Loading a package caches all package, pset, and task metadata in memory in the package manager and readies the package for runtime use. Package loading is done on a per-session basis, i.e., multiple user sessions will each have their own package management runtime context and independent data storage (except for the read-only shared package data).

The package manager provides an API which applications can use to access package metadata, including listing the available packages, querying the metadata of a package, and listing the components, parameter sets, message classes, or other objects defined by the package.

### 3.5.2 Managing Parameter Sets

In a distributed system multiple programs, possibly written in different languages, may simultaneously need access to the same parameter set data. The problem may be mitigated somewhat by sending read-only copies of parameter set data to tasks, but this does not entirely solve the problem as tasks may return pset data to be shared with other tasks, or may edit and update shared global psets. The problem is similar to the case of a database server which may provide simultaneous access by multiple clients to shared data. This has been an issue in actual astronomy data systems for years and is not a theoretical problem.

To address this problem parameter set data is managed by the package manager, which normally resides in the same address space as the application scripts (e.g., both in the same Python session). When the package manager loads a package it interprets the XML-formatted package metadata, including parameter set definitions, and creates runtime instances of all pset classes in memory. Pset instances may then be created from these classes at runtime. Pset instances are binary objects within the memory space managed by the package manager.

The package manager provides a client API, bound into each target application language, which may be used to access and manipulate parameter sets. The functions provided include the following:

- Query pset metadata.
- List the parameters stored in a pset.
- Create, destroy, or copy pset instances.
- Add new parameters to a pset.
- Set or get individual parameters within a pset.
• Verify the contents of a pset.
• Serialize or deserialize a pset as a byte stream.
• Save a pset to external storage, or load a pset from external storage.

When a new pset instance is created from a pset class, the specific constructor used determines the contents of the pset. Often a pset instance will contain only parameters defined by the pset class, however it is also possible to add non-standard parameters to a pset instance, to pass information not formally part of the formal data model of the pset. Pset editing includes construction of parallel psets.

As an alternative to the explicit pset access API, it may also be possible to interface psets directly into the application language as language objects, in which case the application would not need to directly interact with the package manager for basic pset manipulation. Also, merely executing a task may not require any direct manipulation of the task parameter set, as we shall see in the next section.

3.5.3 Managing Tasks
When a package is loaded the package manager compiles a list of all components within the package. The information recorded includes:

• The type of component (task or tool, serial or parallel).
• For each task or tool method, a list of all input and output parameter sets.
• The language and container type required to run the task.
• Whether the component is dynamically loadable or statically linked.
• The location of the object code for the task (dll, executable, etc.).
• Any constraints on where and how the task can be executed.

For example, in a typical simple case we might have compiled tasks which are statically linked with the container. There would be a single executable and probably a single input parameter set, and the container and task could execute on localhost (the user workstation) or on any compatible node within the resources managed by a DVM session.

In addition to compiling all this metadata, when the package manager loads a package it may also create language bindings (native language functions) for all task and tool methods within the local process. This is done automatically from the package, task, and parameter set metadata, which together fully define the tasking interface for the package.

For example, in a Python session the package manager at package load time would create Python functions for all task and tool methods defined by the package. Each Python function is a wrapper for the actual task which is executed via the package manager. Executing a task from a Python script would
be a simple matter of calling the Python function. This could cause the following steps to happen:

- Application script calls Python function with native Python arguments.
- Control is transferred to the package manager.
- The package manager performs parameter resolution on the task pset.
- The task is executed using the pset resulting from parameter resolution.

Parameter resolution takes the arguments from the Python function and uses these plus the task input pset class to create the input parameter set required to run the task. More precisely, a new instance of the task input pset is created, the arguments from the Python function are used to override the referenced parameters, pset validation may optionally take place, the pset instance is used to run the task, and the temporary pset instance is destroyed once the task completes.

It is also possible for an application to prepare one or more psets in advance and use these to run the task. This may be useful for cases where a static pset can be used to run many task instances, or in cases where the application wants to have greater control over pset creation, such as generation of a parallel pset.

Spawning a container process to run a task is initiated by the package manager but is the responsibility of the DVM, as discussed in section 3.6. Once a container is available to run a task, the package manager may run the task merely by sending a request message to the container, including any input psets as message data. If the targeted container happens to be in the same address space as the package manager the DVM may be bypassed and task execution becomes comparable to a function call.

### 3.5.4 Distributed Package Management

While logically the package manager is a global framework service, it is implemented as a library so that it may be embedded in the same process used to execute applications code, e.g., a Python or Java session. Access to the API is local to the containing process, but API functions may execute either locally or remotely (this is much the same approach as for the container).

For example, consider the typical case of a user Python session executing tasks which may run either locally or remotely. The package manager is implemented as a library within the same process as the Python interpreter, with the package manager probably executing on its own thread. Manipulation of package metadata and psets from the Python session is efficient since everything is in the same process and address space.
While this is the most common case, it is also possible in a large system for other processes to need access to the package manager to run tasks or access parameter data. To provide this the package manager functionality is replicated in each remote process, but the API may execute via RPC to the master package manager, or possibly using some combination of RPC and locally replicated package data.

This approach implements the package manager as a shared framework service while providing optimized access for the main session process, which probably accounts for 99% of the access to package manager resources. Most task execution is straightforward, with runtime pset instances being passed back and forth via messaging, without need for a remote task to access the package manager – but they can do so if some use case requires it. An alternative approach if context can be more localized is to have two or more package manager instances, each in a separate process.

### 3.6 Distributed Virtual Machine (DVM)

The distributed virtual machine (DVM) provides basic process execution, control, and interprocess communication facilities for use in a distributed system. The DVM implements a virtual machine abstraction, making a dynamically configurable set of host computers (usually cluster compute nodes) appear as if they were a single machine.

The physical DVM may be some subset of the nodes within a large cluster, or a single computer, transparently to applications software and the user. If the target
system is a desktop computer the DVM may only provide a conventional multiprocessing capability, with a few processes running on the user laptop or workstation; this is the default, requiring no special configuration on the part of the user. On a large cluster the DVM may be configured to use local resource allocation and resource management facilities so that DVM-based applications play well within a shared cluster computing environment. The purpose of the DVM is to hide all of this from applications as well as from framework elements such as the container and package manager, in order to provide transparent scalability, allowing applications to execute transparently either on the desktop or on a remote cluster or Grid system.

The distributed processing functionality provided by the DVM includes:

- Distributed environment variable mechanism.
- Process execution and control (spawn, kill, exit code, etc.).
- Capture and transfer of process standard I/O and error streams.
- Basic messaging capabilities (request, producer/consumer, async).
- Resource allocation and resource management (on larger systems).
- Fault recovery and checkpoint-restart support (on larger systems).
- Integration with parallel processing software such as MPI (larger systems).
- DVM console and logger services, and stand-alone DVM boot capability.

In the simplest case the DVM provides only process execution and control, basic messaging, capture of process i/o streams, and a distributed environment mechanism, supporting execution on a single computer. A usable system for use on single-computer systems, supporting the package manager, container, distributed computing, and messaging can be built given only these DVM capabilities. More capable DVM implementations may provide all of these capabilities, but still be capable of executing in a minimal default configuration on a single node.

We don’t anticipate implementing the full-up scalable DVM from scratch. Scalable computing in a large cluster or Grid environment is complex and requires good integration with the software commonly found in these environments. Hence the DVM would be implemented as a layer on top of existing open source HPC/Grid software (OpenMPI/OpenRTE is an example of such a technology, and provides everything we discuss here). Such systems provide builtin MPI integration and support for basic messaging, as well as integration with standard HPC resource allocation and management systems and related technology (Bproc, PBS, SLURM, Xgrid, Yod, Myrinet, Infiniband, etc.).

The most basic DVM however, would probably be implemented from scratch within the software we control, to provide a well integrated, lightweight implementation supporting common desktop usage. The key is that the DVM abstraction be simple to use for basic distributed computing, and at the same time general enough to scale up to larger systems.
3.7 Framework Services
[To be added]. Discuss the two primary framework services:

- **System Logger.** A basic logging service is provided as part of a full-up DVM implementation, however a more fully-featured logging tool will be required by the applications framework. This is a tool component which subscribes to logging messages from all log message publishing components in the system. Messages may be filtered or sorted and displayed in a GUI, or may be spooled to persistent storage in real time.

- **Data Access Object (DAO).** A data access object provides concurrent, shared access to a data object, so that multiple clients may simultaneously access the object without compromising data integrity. The simplest case of this is basic locking of entire files, which is what the basic DAO would provide. The basic DAO may be subclassed to provide access to specific types of objects, such as instrumental data, tables, images, spectra, and so forth. These more specialized DAOs implement an object model and mediate concurrent access to object metadata, as well as provide file-level locking for bulk data associated with the object. Large objects are stored as many files, each of which may be individually allocated for exclusive access by a compute process (if write access is required), or made available to multiple clients if read-only access is sufficient.

Other standard framework services are possible, however these quickly become application specific. An open-ended variety of “tool” type components are possible to provide standard services via the framework.

4 System Installation and Management
[To be added.] Discuss the distinction between applications packages, which are defined in terms of the applications framework, and packaging of the system framework itself, which must be defined in terms of the facilities available on the supported target platforms.

The concept of an applications package has already been discussed here (section 3.1). In the case of packaging the framework itself for installation on a host computer, what we would probably propose at this point is that:

- The primary distribution for end-users should be precompiled, packaged software which has been tested on each supported target platform. A separate binary distribution is required for each target platform, as with virtually all Linux, Windows, MacOSX, etc. end user software distributions.
Developers of course, may use the build system to build the system from source.

- The form in which the base system is packaged is TBD. RPM, DEB, BSD “ports”, Windows EXE or MSI packages, etc. are examples of packaged distribution, however these are all platform-specific and fairly maintenance heavy package formats. A more platform neutral approach, used for example by Sun Java, is a gzipped TAR or ZIP with an install script. This has the advantage that it can be installed to an arbitrary directory root, but only provided the runtime software is capable of runtime configuration to run at an arbitrary root pathname (this can fail if fixed pathnames are compiled into the code). Most software can be configured to run at an arbitrary root provided file paths are constructed at runtime from base paths defined in the process environment. In addition to Java, Python and IRAF already demonstrate that this is possible.

- Having the software be capable of being installed at an arbitrary root is important to allow users to install the software without administrative privilege (important in many environments as well as for user convenience), and less obviously, to allow multiple versions of the system to be simultaneously installed, with the one to be used to be selected at runtime.

It appears that multiple versions of Python can be simultaneously installed on a host computer, and in particular that Python can be integrated into a system, including runtime specification of the directories where Python modules and other runtime files are installed. This means that it should be possible to fully integrated Python into a packaged distribution, including all required modules. The system integrator and packager would perform all the integration and build all the required modules, providing a platform-specific binary distribution for installation by the user. Java of course can also be integrated and configured in this fashion; while probably not all that important for the JRE, it could be important for any standard classes used by the system.

Ideally the final system should be self contained or very nearly so, precompiled, and ready to install and run by the end user. Once the base applications framework is installed, the runtime framework can be used to install and run applications packages.

5 Analysis of Specific Use-Cases

5.1 User Perspective

This Use Case concentrates on how a typical astronomical user might interact with a system based on the architecture and design given above. The basic scenario is a user who just received some new data which needs to be reduced
and compared with other recent results. The amount of data is assumed to be limited so that it can be processed on the users desktop system with a modern multi-core CPU. Before the user starts, the software system has been installed in a configuration which supports parallel execution of tasks (using the multiple CPU cores) on a single computer node. The astronomer would then typically have to go through the following steps:

- **Start-up system**
  - The environment is started by a single command (e.g. from a Unix shell window). This command does two things namely:
    - Checking that system services required (e.g. package manager and DVM) are available and starts them if they are not.
    - Provide a Python shell through which the user can issue commands.

- **Check for updates and packages**
  - A package manager tool is started from the Python shell. This tool, including a GUI, will show loaded packages (e.g. a standard pipeline reduction package) but also have options to list all available packages and updates to them.
  - The user selects the special pipeline package for her data and asks it to be downloaded and installed.
  - Finally the package manager is requested to apply all available patched to the packages loaded. Although a background process could do this, it may be safer to have the user explicitly requesting which updates to install.

- **Retrieve and view data**
  - The new data were delivered on DVD's and the user copied them on to a local directory before starting the session.
  - She checks the data using standard Python commands (e.g. PyFITS) which were imported by default when Python was started.
  - An image display tool is started from Python to check a few data frames.

- **Start reduction pipeline**
  - Having verified the basic quality of the data set, the user starts the special pipeline tool for her data which she just downloaded.
  - This tool provides a GUI which both can be used to configure the pipeline reductions and show its current reduction status. The tool is asynchronous so that new Python commands can be issues while this GUI is active.
  - The pipeline is configured to reduce the new data and process them as background processes running of a subset of the available CPU cores.
  - The user starts the pipeline processing from the GUI through which she also can view the progress

- **Check for similar data through VO**
• While the pipeline reductions are running, a VO data discovery tool is executed.
• It is requested to search for similar data available at other archives.
• Since only very few data sets are found, it is decided to retrieve them from the archives and store them locally.
• This is done using the VO tool.

• **Check pipeline reduction status**
  • During the search for archival data, it is noticed that the pipeline has finished as indicated on the status display on the pipeline GUI.
  • The pipeline would also issue a message saying that it had finished. If the pipeline GUI had been terminated, the user could check for this message to verify the pipeline status.

• **Verify pipeline results**
  • The pipeline results and the archival data are compared using Python commands (e.g. statistics and arithmetics in standard Python packages).
  • A table display tool (e.g. TopCat) is started to view some of the quality control data produced by the pipeline.
  • The image display tool (still running) is used to overlay the new results with the archival data.

• **Terminate session**
  • Being satisfied with the results the user decides to make it a day and logout from the environment.
  • If the pipeline was still running, this logout would only terminate the Python shell while the services (e.g. package manager and DVM) would still be alive and continue the pipeline processing. The user would then next morning when she would start the environment get connected to the environment already working.

5.2 *Inline Task Execution*

5.3 *Remote Task Execution*

5.4 *Parallel Execution*

5.5 *Pipeline Processing*

5.6 *VO Data Service*

5.7 *VO Application*
5.8 Integration of Legacy Code

5.8.1 AIPS

[To be added]. Bill Cotton has contributed the discussion below of ParselTongue, which is a Python front-end for AIPS tasks. The basic approach for interfacing AIPS tasks to the new framework would be to use the managed subprocess container, using the ParselTongue Python code (or some comparable implementation) for the AIPS adaptor layer in the container. The build system would use the current ParselTongue heuristics to reverse engineer the AIPS task Interfaces and parameter sets to build a framework-compatible AIPS applications package.

Bill’s thus far unedited text follows.

The ParselTongue package (Kettenis et al 2005) is a Python-based scripting environment for running Classic AIPS tasks with an architecture very similar to the design discussed in this document. As a fully operational system, ParselTongue has all of the necessary features to run the Classic AIPS package in a completely unmodified form. ParselTongue scripts perform their functionality by executing AIPS tasks and can be controlled by values in AIPS data structures.

In the notation of the design discussed in this document, the Python scripting engine in ParselTongue represents the Presentation and Applications layers. Operations involving AIPS tasks or data are all performed in a Python "Container" which is a ParselTongueServer process for remote processing or the same functionality in the same address space as the scripting engine if local. Xmlrpc is used as the software bus for communications among various independent components of the package. Use of xmlrpc allows distributed operations over a cluster or network as well as those entirely confined to a workstation or laptop. Python structures passed across xmlrpc are converted to/from XML.

Data areas in AIPS are described as "disks". The definition of an AIPS disk in ParselTongue includes the URL of the relevant host for remote data. ParselTongue then decides if a given task execution/data access is local or remote, and which host if the latter and calls the appropriate Container.

AIPS tasks receive their parameters in a binary task interface file and can be scalars or arrays of either floats (as floats), booleans (as floats), integers (as floats) or strings (as floats). The order, type and size are defined independently in the task and in the CLI (POPS) but must be defined to be the same. The names, order and allowed ranges of the input, output and updatable parameters for a given task are contained in a task specific "Inputs/Help" file which is read by POPS and used to write the parameter values to the binary task interface file. All parameters in POPS are global and are defined in a file ("POPSDAT.HLP")
which gives the types and sizes of the global parameters. The functionality of POPS is replaced in Python functions in ParselTongue.

One complication of the AIPS/POPS structure is that multiple copies of POPS may be running. In order to avoid conflicts over the task interface file, each POPS session is given a unique "POPS number" and a section of the task interface file is allocated to each POPS number.

Some AIPS tasks also return parameters such as statistical parameters of images. These values are returned in the binary task interface file when the task completes and POPS extracts them to its parameters where they are available from the CLI.

In the ParselTongue implementation, all detailed knowledge of tasks resides in the "Container" which both generates task parameter set/user documentation objects and executes the tasks. All AIPS data access is also through the Container. This design allows operation in environments where the scripting engine is run remotely from the data/server processes. Access to data, especially data headers is critical in the control of data processing scripts.

The major functions needed for executing an AIPS task are:

- **Derive task parameter set/user documentation**
  
The scripting engine sends the Container a request for the task interface object for a particular task. The Container then obtains this information by parsing both the task "Input/Help" file to get the names, order, etc. of the parameters and the POPSDAT.DAT file to get the definitions of the parameters. The Container then passes the task interface object back to the scripting engine. Task interface objects contain input parameters, output parameters (optional) and user documentation.

- **Set parameter values**
  
The Python application script then sets the parameter values on the task interface object, possibly based on values obtained from AIPS data structures.

- **Convert parameter values into binary file read by AIPS tasks**
  
In order to execute the task, the scripting engine sends the request, together with the task interface object to the Container. The Container then extracts the task parameters and writes them into the AIPS task parameter interface file.
After the scripting engine sends the task execution request, it loops processing the tasks logging messages and waiting for the task to complete.

- Fork/exec the AIPS task

Since the AIPS tasks are Fortran programs, the command line interface is exceedingly primitive. In order to pass the "POPS number" needed to access the task interface file, a link is created in /tmp with the name of the executable followed by the POPS number. The container then does a fork/exec of this link. The task first calls a c routine that extracts the POPS number from the name of the "executable" link used to start it, reads its parameters from the binary file and performs the desired operations.

- Trap/return task messages written to stdout

While the task is executing, it writes logging messages to stdout which are trapped by the container and passed back to the scripting engine where they are displayed and optionally written to a text file.

- Wait for task completion

ParselTongue execution of AIPS tasks is synchronous. When the AIPS task finishes, it writes a completion code into the binary task parameter file. This value is read by the Container and if it indicates an abnormal termination (or the task terminated without updating the completion code) an exception object is generated and passed back to the scripting engine. The scripting engine can then either handle the exception or terminate the script. The Container also extracts any output parameters from the task interface file and copies them to the task interface structure.

One feature of the AIPS/POPS task interface not implemented in ParselTongue is the "SHOW/TELL" feature whereby an interactive user can modify the parameter values of a running task, or tell an iterative process that it's done enough. This is implemented by the same mechanism as for the initial parameter passing, by updating values in the binary task data file which the task (optionally) reads when its processing might usefully be modified. Since ParselTongue only runs tasks synchronously, this feature is not useful; but, it is highly desirable for a system with asynchronous task execution.

The Obit package (Cotton, 2008) is a further development of the ParselTongue environment using the same framework for Obit as well as AIPS tasks.

Cotton, W. D., 2008
"Obit: a Development Environment for Astronomical Algorithms",
PASP, 120, 439--448
Applications Framework Architecture and System Interfaces


5.8.2 CASA
[To be added]. CASA, currently under development at NRAO, is derived from AIPS++. CASA mostly consists of a large body of class code written in C++. This has been reworked in recent years to “componentize” the code at the tasking level, and can be repacked in the form of applications packages containing a combination of tasks and tools with parameter files. In particular the current tasking interface in CASA describes the component interface in XML, much as has been described here, including definition of parameter-based interfaces for tasks and tool method (CASA has both tasks and tools). High level scripting is currently being done in Python, including pipeline processing software. Hence the current class code and Python applications are compatible with the new framework architecture and it is thought that these can be migrated to the new framework with a modest effort.

5.8.3 IRAF/PyRAF
[To be added]. IRAF already has a task-parameter structure similar to what is described for the new framework. The “IRAF Main” would be replaced by the container described here, plus interface adaptors for facilities such as for parameter access in task code. Most of the IRAF applications packages would port with minimal effort. The CL scripts could be rewritten in Python, but a better alternative might be to use the Python-based CL emulator from PyRAF. A reworked version of the PyRAF CL emulator could make a very good CLI for the new system. Essentially all other PyRAF Python code would carry over to the new system with minimal change since it is Python based, and Python is fully supported in the new system.

5.8.4 Starlink
[To be added.] Architecturally this is similar to IRAF. Some years ago an adaptor was written to put a new front-end on Starlink packages and make them compatible with the IRAF packaging mechanism and parameter set model. This suggests that the problem of repackaging legacy Starlink code for the new framework is comparable to that required for IRAF.

5.8.5 ESO Common Pipeline Library (CPL)
[To be added; some draft text follows].
The Common Pipeline Library (CPL) is used by ESO as a common basis for implementation of standard reduction pipelines for data coming from instruments at its VLT observatory. Beside numerous useful functions for image processing and data manipulation, it also defines a concept for Pluggable Data Reduction Modules (PDRM). Such modules must implement as small standard interface which controls their life-cycle (i.e. initiation, creation, execution, and termination). This makes it possible to treat PDRM's as dynamical libraries and load them at run-time by a pipeline manager process.

The concept of PDRM and their small, well defined interface are fully compatible with that of components and tasks described above. This would make it very easy to include CPL PDRM's into a system based on the architecture outlined. To achieve access to a PRDM, one needs to provide special implementations of two parts of the system namely:

A CPL PDRM compatible container:

- A special CPL container would need to be written. It would convert between standard parameter sets and the special recipe parameter structures used by CPL. Further, it would explicitly call the CPL life-cycle functions to start, and terminate PDRM's.

- This would provide the basic ability to execute PDRM but would still rely on the CPL internal functions for logging and error handling. For a full implementations of CPL based components, it would be required also to replace the CPL interfaces for logging, messaging and error handling with implementation based on the similar functions provided by the DVM.

An CPL add-on for the Package Manager:

- The packaging of PDRM's is likely to differ from that of other packages. In order for the Package Manager to identify and load CPL based pipeline packages, it would be needed to provide either a special add-on which can find CPL packages and translate them to the standard representation, or an off-line, stand-alone process could be created to explicitly convert CPL packages to the standard format.

No code of the actual CPL modules would need to be changed as they still can rely fully on CPL for their internal processing.

Appendix A: “Appendix Title”

list of interfaces
Glossary
Glossary of terms [To be added].

References