Astronomical Data Analysis Software and Systems XII
ASP Conference Series, Vol. 295, 2003
H. E. Payne, R. I. Jedrzejewski, and R. N. Hook, eds.

OPUS: A CORBA Pipeline for Java, Python, and Perl Applications

Walter Warren Miller III, Christopher Sontag, Jim Rose[2]

Space Telescope Science Institute, Baltimore, MD 21218

Abstract. With the introduction of the OPUS CORBA mode, a limited subset of OPUS Applications Programming Interface (OAPI) functionality was cast into CORBA IDL so that both OPUS applications and the Java-based OPUS pipeline managers were able to use the same CORBA infrastructure to access information on blackboards.

Exposing even more of the OAPI through CORBA interfaces benefits OPUS applications in similar ways. Those applications not developed in C++ could use CORBA to interact with OPUS facilities directly, providing that a CORBA binding exists for the programming language of choice. Other applications might benefit from running 'outside' of the traditional file system-based OPUS environment like the Java managers and, in particular, on platforms not supported by OPUS.

The enhancements to OPUS discussed in this paper have been exercised in both Java and Python, and the code for these examples are available on the web.

1. Introduction

While users always had the ability to provide external tasks written in any language, those tasks have a limited amount of flexibility to respond to the environment. With the advent of OPUS 4.2, users of the OPUS Pipeline Platform[1] now have the ability to write applications which are fully aware of the OPUS pipeline in any language which has a CORBA binding.

2. Background

The OPUS platform is a distributed pipeline system that allows multiple instances of multiple processes to run on multiple nodes over multiple paths. While OPUS was developed to support the telemetry processing for the HST instruments, it is a generic pipeline system, and is not tied to any particular processing environment, or to any particular mission. From this point of view the OPUS platform does not provide the mission specific applications themselves. Instead

[2]Astronomy Division, Computer Sciences Corp.

[1]`http://www.stsci.edu/resources/software_hardware/opus`

OPUS provides a fully distributed pipeline processing environment structured to help organize the applications, monitor the processing and control what is going on in the pipeline.

The basic architecture of the OPUS system is based on a blackboard model where processes do not communicate directly with one another, but simply read and write to a common blackboard. In the original implementation of OPUS, the blackboards are accessed through the (network) file system as a directory on a commonly accessible disk. In a cluster of workstations and larger machines, if the protections are set appropriately, any process can "see" any file in the blackboard directory: the "posting" of blackboard messages consists of either creating or renaming an empty file in that directory.

3. The OPUS Application Programming Interface

The OPUS Application Programming Interface (OAPI[2]) is an object-oriented, C++ interface to the OPUS environment distributed for Solaris, Linux, and Tru64 platforms. With the OAPI, internal-polling OPUS pipeline applications can be developed that take full advantage of the capabilities and flexibility offered by OPUS. The OAPI contains classes for interacting with the OPUS blackboards and their contents, for reading an assortment of resource files, for message reporting, for event handling, and for exception handling. Its functionality can be extended to include additional or customized features not yet provided by the library through traditional object-oriented techniques like inheritance and composition.

The OAPI was designed to satisfy the needs of two groups of software developers. On the one hand, it serves the programmer who wants to develop OPUS-savvy processes without regard for the implementation details of the OPUS system. Ease of use is a primary consideration for such a developer. On the other hand, it must be easily maintainable, backwards compatible with previous versions of OPUS, and offer the flexibility to meet future requirements of OPUS pipelines with little impact on existing code. These goals demand a general, abstract approach to the architecture with strict isolation of interface from implementation—a methodology that is often at odds with ease of use. The library follows a middle-of-the-road tack by promoting flexibility and ease of use through run-time polymorphism.

The OAPI exposes a set of interfaces defined by a set of core abstract base classes. Where applicable, the base classes provide an implementation, but far more often, specialized classes are derived from these base classes that provide the actual functionality of OPUS. Access to these derived types is achieved through a pointer to the base class and is transparent to the client. Using the run-time polymorphic behavior of C++ class hierarchies in this way is a powerful tool that helps preserve a high degree of separation between implementation and interface. Separating implementation from interface allows the use of generic algorithms to process different implementations of an object through a common interface thereby reducing code duplication and development effort. In addition,

[2] `http://www.stsci.edu/software/OPUS/oapi-how-to.pdf`

it permits evolution of the OAPI with minimal impact on the clients of the library since they only reference the interface exposed by the base classes.

4. Internal Pollers

An OPUS pipeline is defined by the set of applications that processes data or that performs tasks in a co-operative manner and the rules that determine when they should act and how their results should be interpreted. OPUS pipeline applications fall into two general classes: internal pollers and external pollers. Internal pollers are developed with explicit knowledge of the OPUS environment: they make direct calls into the OAPI library for initialization and for event handling, and must be linked against the OAPI library.

External pollers use a proxy application (xpoll) to communicate with OPUS and typically are wrapped by a shell script. xpoll interacts directly with OPUS and executes the external poller whenever work is to be performed by that process. xpoll communicates OPUS event data and receives process completion status through the external poller's environment. As long as a suitable shell script can be developed that meets the input requirements of the application, any application can be used in an OPUS pipeline. Both internal and external pollers share many of the same basic capabilities although internal pollers, by virtue of having direct access to the OAPI, are more flexible. The OPUS Sample Pipeline demonstrates both classes of applications (g2f is an internal poller; all of the other applications are external pollers).

Internal polling processes, like g2f, are programs written with knowledge of how the OPUS blackboard works. They are typically processes with some significant start-up overhead (e.g., database reading, etc.). The process is written to perform the start-up overhead and then enter a polling loop to wait for pipeline events. The process stays active as it polls for and processes events. Internal pollers are built using the OAPI to communicate with the OPUS system, and can respond to a reinitialization command.

5. New Capabilities

With the introduction of the OPUS CORBA mode, a limited subset of OPUS Applications Programming Interface (OAPI) functionality was cast into CORBA IDL so that both OPUS applications and the Java managers were able to use the same CORBA infrastructure to access information on blackboards.

The primary motivation for doing so was to improve scalability, but moving to distributed object architecture also freed the managers from running strictly in the confines of an OPUS user environment. It also reduced the amount of duplicate code that otherwise would be required in a multi-programming language environment.

Exposing even more of the OAPI through CORBA interfaces benefits OPUS applications in similar ways. Those applications not developed in C++ could use CORBA to interact with OPUS as internal pollers providing that a CORBA binding exists for the programming language of choice. Other applications might benefit from running 'outside' of the traditional file system-based OPUS environment like the Java managers and, in particular, on platforms not supported by OPUS.

To help users get started with their own internal pollers in Python or Java, sample code has been included with the OPUS 4.2 distribution which demonstrates exactly how to set up your environment, build the tasks, and run the applications.

A more complete discussion of the design of the **opususer** class is presented on the web.[3]

6. Example

While there is not sufficient space to present a complete example, you can inspect the Python code on the web[4]. That example is written in Python (1.52 or higher) and runs on Solaris 5.7/8 using the free OmniORBpy CORBA ORB. All the code that is necessary to wrap the OPUS API is presented in this example. Full instructions on downloading the necessary libraries, building, installing and running the application are included on that web page.

The particular example illustrates how to access resources in the process resource file and how to construct the polling loop for normal applications. To develop your own applications you might need only to substitute your own **processDataset** method.

References

Miller, W. & Rose, J. 2001, The OPUS CORBA Blackboards and the New OPUS Java Managers, in ASP Conf. Ser., Vol. 238, Astronomical Data Analysis Software and Systems X, ed. F. R. Harnden, Jr., F. A. Primini, & H. E. Payne (San Francisco: ASP), 325

Rose, J. et al 1994, The OPUS Pipeline: A Partially Object-Oriented Pipeline System, in ASP Conf. Ser., Vol. 77, Astronomical Data Analysis Software and Systems IV, ed. R. A. Shaw, H. E. Payne, & J. J. E. Hayes (San Francisco: ASP), 429

Walter Warren Miller III 1999, OAPI: the OPUS Application Programming Interface, in ASP Conf. Ser., Vol. 172, Astronomical Data Analysis Software and Systems VIII, ed. D. M. Mehringer, R. L. Plante, & D. A. Roberts (San Francisco: ASP), 195

---

[3] `http://www.ess.stsci.edu/gsd/dst/oapi/servers/oresource_java.html`

[4] `http://www.stsci.edu/software/OPUS/javafaq/opusfaq_pyint.html`