Astronomical Data Analysis Software and Systems IV
ASP Conference Series, Vol. 77, 1995
R. A. Shaw, H. E. Payne, and J. J. E. Hayes, eds.

The OPUS Pipeline: A Partially Object-Oriented Pipeline System

J. Rose (CSC), R. Akella, S. Binegar, T. H. Choo, C. Heller-Boyer, T. Hester, P. Hyde, R. Perrine (CSC), M. A. Rose, K. Steuerman

Space Telescope Science Institute, 3700 San Martin Dr., Baltimore, MD 21218

Abstract. The Post Observation Data Processing System (PODPS) for *Hubble Space Telescope* Science Operations Ground System was designed under the expectation that problems and bottlenecks would be resolved over time by bigger, faster computers. As a consequence, PODPS processes are highly coupled to other processes. Without touching the internal components of the system, the second generation system decouples the PODPS architecture, constructs a system with limited dependencies, and develops an architecture where global knowledge of the system's existence is unnecessary. Such a (partially) object-oriented system allows multiple processes to be distributed on multiple nodes over multiple paths.

The PODPS "pipeline" currently includes seven major processing steps organized in a linear sequence; each step builds on the previous step. Designed to run on a single minicomputer, the original architecture used VMS-specific interprocess communication facilities to control the operation of the system.

Today we are operating in a cluster environment with over 30 workstations and many gigabytes of disk space. As the volume of data and the amount of work done on these data increase, and as other reporting and monitoring requirements expand, it has become increasingly evident that the limitations imposed by the existing pipeline system are becoming a serious bottleneck.

We have proposed a solution to this bottleneck: opening up the architecture and taking advantage of the resources already available in the pipeline environment. Without touching the internal components of the system, the second generation pipeline at ST ScI decouples the PODPS architecture, constructs a system with limited dependencies, and develops an architecture where global knowledge of the system's existence is unnecessary. Using a simple blackboard architecture, such a (partially) object-oriented system allows multiple processes to be distributed on multiple nodes over multiple paths.

While the system is not being redesigned or rewritten in object-oriented languages, there is one major change to the system influenced by the dominant character of object-oriented systems: it is the objects themselves which know their own state, and that knowledge is what "controls" their processing. This simple understanding allows us to move from a procedural, linear, controlled, sequential processing system to one which is distributed, event-driven, parallel, extensible, and robust.

In the older system, the knowledge of an observation's state was maintained only by the controller process, which had exclusive access to a global memory area. The advantage of such an architecture of control is that race conditions and deadlocks are avoided by channeling access to the status information through a single point.

The disadvantages of this architecture are many. The worst aspect of this design for our purposes is the obvious control coupling: the controller "controls" and must know about all processes in the system in order to schedule activities. The result of the dependencies set up by this control coupling is that when any one of the system components fails, the entire system fails and must be restarted. Equally difficult is that the observation status information stored in memory is not persistent: if the controller exits, the memory of the status of observations in the pipeline is lost. While this architecture allows multiple copies of the system to be run independently on different machines (assuming separate paths), this is a trivial interpretation of distributed processing, and does not allow for multiple copies of selected processes focused on the same path. An effective solution to these problems is simply to eliminate the controller and to re-engineer the system around a simple blackboard architecture.

1. An Object-Oriented Blackboard Architecture

The blackboard model is both simple to understand and, in our case, simple to construct. The paradigm is that of a number of independent players with pieces of a jigsaw puzzle. One player might put a piece on the blackboard, and others will attempt to fit their pieces into the puzzle as connections become apparent. All the information necessary to complete the puzzle is posted on the blackboard, and seen by all players simultaneously. Additionally, the solution is achieved by independent players who need not communicate with each other, except through the blackboard.

Applying this model to the pipeline environment means that the status of each observation is "posted" to the blackboard. All processes can see the status of all observations at the same time. If a process notices that it has something to do (i.e., advance the status of a particular observation), then it does it. When complete, it reposts the revised status of that observation, and continues its search for something to do.

There are a variety of ways to implement the blackboard architecture: a shared global memory, a database relation, or a network of messages. But each of these solutions again requires a controller specifically to guard against the ambiguities of concurrency. Instead of selecting an architecture that requires an elaborate controller, we decided to use the concurrency controller already built into the operating system: the file system.

File systems constitute one of the central parts of any operating system, and have received a great deal of attention in their design to problems of concurrency. Especially in environments where multiple machines are clustered with common access to files on multiple devices, the operating system is required to deal explicitly with lockouts, exclusions, and race conditions. By using the file system we avoid duplicating a fairly complete, complex, and sophisticated piece of software with a proven ability to deal with concurrency problems.

Thus our "blackboard" is simply 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” consists of creating or renaming a file in that directory.

When an observation enters the system the first pipeline process creates an “observation status file” in the blackboard directory. The name of that file contains the processing start time, its status (which pipeline steps have been completed), and the nine-character encoded name of the observation. The file itself is empty—all the information which the system requires is embedded in the name of the file. For example the following entries might be found on the OPUS blackboard:

```
19941105141608-CCCCCP_____.Y1GV0101T
19941105141628-CCCCCW_____.Y1GV0102T
19941105150000-CCW________.U0CK0101T
```

The first indicates that the observation Y1GV0101T (which started pipeline processing on November 5, 1994 at 14:16:08) is currently being processed by the sixth component of the pipeline. The second observation, Y1GV0102T, has completed the fifth step and is waiting to be processed by the sixth step. Similarly, observation U0CK0101T has completed the second step and is waiting to be analyzed by the third step.

Each process in the system polls the blackboard directory for files with a particular status. For example the third processing step will poll for files with names like *-CCW*.* which indicates the first two steps have been completed and the observation is waiting for the third process. Upon finding a candidate, the process will attempt to rename the observation status file, replacing the “W” (waiting) with a “P” (processing). Because we can have multiple instances of this third process, this renaming can fail: another process might have renamed that observation status file already. This is not a problem, we just attempt to find another observation—better luck next time.

When processing for a step is complete, the observation status file is again renamed. If processing was successful, the “P” is changed to a “C” (complete), and the next process in the sequence of steps will eventually recognize this observation as a candidate. If processing is unsuccessful, the “P” is changed to an “E” (error), and the error collector process will notice a candidate for its handling. Once all processing, including archiving, is complete for an observation the status file can finally be erased (deleted) from the blackboard.

A polling environment raises a resource question: what percent of the CPU is consumed by the polling processes? While the answer to this question depends on the number of processes, the polling frequency, the nature of the CPU and the I/O bus, and other factors, it is still a bit of a red herring. When processes are busy converting, analyzing, calibrating, or plotting an observation, they are not polling. When there are no observations to process, that is, when the machine is relatively idle, there will be more polling going on as processes try to find something to do. In any case, on a VAX 4060 with twenty processes polling on a 5 second interval, less than five percent of the CPU is affected.

In implementing the blackboard pipeline we developed two categories of processes: those which have the polling loop implemented internally, and those which are run from an operating system shell which does the polling. The first category is for processes which have a significant amount of initialization to perform; those processes are designed to poll internally after that initialization is performed only once. Other processes, which have either a limited amount of initialization, or are treated as black-box executables, are handled as if they were simply part of a script.

Such a scripting shell clearly allows the pipeline to be extended in simple ways. Each process has an associated resource file which specifies the type of process (OPUS poller, IRAF task, DCL procedure, etc.), observation status triggers, as well as its success and exception triggers. To extend the pipeline to include another process, all that is required is the new resource file—no change to any existing code, other than a resource file, is necessary.

As implemented, there is nothing application-specific about either the problem or the solution. Any processing script which must be used on a large body of data sets can be turned into a blackboard distributed processing system. We have demonstrated this in a heterogeneous environment of ancient FORTRAN processes, modern IRAF tasks, and simple DCL procedures.

In order to facilitate monitoring and managing such a heterogeneous and distributed system, we have also developed a cluster of five Motif-based window managers to assist the operator:

the process manager to aid in monitoring what processes are doing (scanning a process blackboard);

the observation manager to assist in monitoring how observations are progressing, and whether there are any problem sets;

a disk monitor to continuously gauge the availability of resources;

a node monitor to graphically illustrate where activity is the heaviest; and

a network monitor to gauge traffic between processors and disks.

These Motif managers will be described more fully in a later paper.

Changing requirements are no longer seen as an exception in the software evolution process. We have learned that change is now part of the process, and we must build systems that are adaptable. Monolithic special-purpose systems have a proven track record of failure to adapt. A blackboard distributed processing system is, on the other hand, flexible, extensible, robust, and easy to build. By distributing the processing load across several workstations, the system can make better, if not optimal, use of an institution's resources. By using the concurrency protection built into the operating system, the blackboard architecture ensures a robust solution as well as a cost effective one.

References

Nii, H. P. 1989, in Blackboard Architectures and Applications, ed. V. Jagannathan, R. Dodhiawala, & L. Baum, (San Diego, Academic Press), p. xix