NOAO DPP Document PL001

The NOAO High-Performance Pipeline System

F. Valdes[1], T. Cline[1], F. Pierfederici[2], B. Thomas[3], M. Miller[2], R. Swaters[3]

National Optical Astronomy Observatory
Data Products Program

October 12, 2006

[1]NOAO Data Products Program, P.O. Box 26732, Tucson, AZ 85732
[2]NOAO LSST Program, P.O. Box 26732, Tucson, AZ 85732
[3]Department of Astronomy, University of Maryland, College Park, MD 20742

Table of Contents

Abstract

The NOAO High-Performance Pipeline System (NHPPS) is an event-driven, multiprocess executor system developed to manage pipeline applications in a coarse-grained, distributed processing environment. It allows easy creation of distributed and parallelized data processing pipelines on a cluster of standard workstations using conventional host-callable programs or Python plugin methods. High performance is achieved by enabling multiple processes to run concurrently to minimize system idle time and by utilizing a cluster of processing nodes to cooperatively distribute pipeline steps and portions of datasets for parallel processing. In addition to the core system we describe optional infrastructure components that work with the core system and with pipeline applications.

Keywords: mosaic pipeline, operator

1 Introduction

High performance pipelines are becoming a necessity for handling the large data rates from current and future astronomical instruments. Examples from optical imaging are the many mega- and giga-pixel CCD cameras in use (NOAO: Muller, 1998; CFH: Boulade, 2003; MMT: McLeod, 2000) or under development (ODI: Jacoby, 2002; DEC: Wester, 2005; Pan-STARRS: Kaiser, 2002; LSST: Tyson, 2002). Not only are images large but the observing cadences can be on the order of an exposure every few seconds. For this type of observational data handling, pipeline systems tend to be I/O bound. Therefore, to maximize CPU usage pipeline systems must employ coarse-grained parallization of both the data and processing.

Coarse-grained data parallelization divides up the data into chunks that can be processed at the same time. Astronomical imaging datasets are well-suited to data parallelization because they may be easily split into smaller sub-images, each of which can be distributed for processing to any one of a number of nodes within a processing cluster. Achieving data parallelization requires specific design on the part of pipeline architects to ensure that their pipeline applications split large datasets to take advantage of distributed processing resources while minimizing the impact of slower network bandwidth. A caveat is that some processing steps are not completely data parallel because of the need to avoid boundary artifacts and ensure global continuity.

Coarse-grained process parallelization, meaning parallelization at the level of individual programs, keeps CPUs busy while I/O is taking place by performing multiple processing steps in parallel. An advantage of coarse-grained process parallelization is that modern, general purpose, operating systems are designed to multi-task efficiently provided there are a reasonable number of processes active at the same time. This avoids the need for more complex multi-threaded programming and special operating systems. It also allows use of conventional data processing programs, often from legacy systems.

We have developed a pipeline infrastructure, called the NOAO High-Performance Pipeline System (NHPPS), for creating pipelines which use these parallelization techniques. The system is based on an event-driven methodology.

An event-driven pipeline works as follows. A pipeline, by definition, "flows" data through a number of processing steps. Each step has some associated action which is typically, though not necessarily, a program. Some steps need to be performed in a certain order, some are conditional on the results of other steps, and some may be performed in parallel with other steps. Rather than writing down the chain of steps as a sequence with control flow constructs, such as the "if/else" in a scripting-style pipeline, an event-driven pipeline only describes the prerequisite events for each step to be performed. The most common prerequisite is the completion of another step, though there are many other types of possible prerequisites. The control flow in this methodology is implicit in the set of prerequisites for each step.

An event-driven approach has many advantages over a sequential one for describing a pipeline. The principle advantage is that parallel steps follow naturally. In fact, the order in which the steps are described is not significant, except as a general aid to understanding the expected data flow, so adding or removing steps is relatively easy. Another key advantage is that events can be generated externally; that is, the description does not have to include explicit information about how the events are generated.

The NHPPS currently includes the following types of events: the exit status of an action (typically the exit code from a completed program), the state of an (in-memory) blackboard, the creation of a (trigger) file, the passage of some amount of clock time, and the the occurrence of specific clock times. Support for additional types of events could easily be added; however, we have found these to be sufficient to allow creation of complex distributed and parallelized pipelines.

Two useful characteristics of an event-driven system are that the events control the flow of execution and that the events may be generated in a variety of ways. Pipeline steps can control their exit status values, create files, and modify the blackboard while operators or non-pipeline programs can create files or modify the blackboard through client programs. This provides the pipeline operator the capability to directly impact and control the pipeline application, or choose to let it run autonomously.

The NHPPS draws on the event methodologies from the OPUS pipeline management system (Rose, 1995) developed at the Space Telescope Science Institute and used by a number of projects. We have expanded and improved on these methodologies in many ways. Two key areas are 1) using a Pipeline Description Language (PDL) in a single file rather than a set of configuration files and 2) a lighter weight event handling implementation which uses one process for a pipeline rather than, typically, one for each step in the pipeline.

In this paper we describe the elements of the NHPPS core system and optional components which work with the core system to build and operate pipeline applications. The core components are a Directory Server, Node Manager, Pipeline Manager, Module Manager, and a Blackboard system. We also describe clients, server methods, and an XML-based PDL that control or interact with these components. These elements are sufficient to implement a high performance pipeline application.

2 Pipeline Applications

As with any complex programming problem, developing a non-trivial pipeline application benefits from the programming concepts of modularity and encapsulation. In our pipeline applications we decompose the data flow not only into process level steps but also into multiple, higher level, interacting pipelines. This is why we make the distinction between a *pipeline application* and a *pipeline*. The NHPPS was designed to support pipeline applications by managing multiple pipelines distributed across multiple nodes.

Higher level pipelines break down a pipeline application into a number of logical units that interact with each other through standard protocols and have limited functions, for example creating a calibration or other data product. These pipelines are analogous to methods or subroutines in programming languages while the steps in the pipelines are the programming statements. The standard protocols for connecting these pipelines are then equivalent to the way methods or subroutines call each other.

This approach has a number of advantages. First is the reduction of complexity in implementing each high level function. Another is the ability to extend the functionality of a pipeline applica-

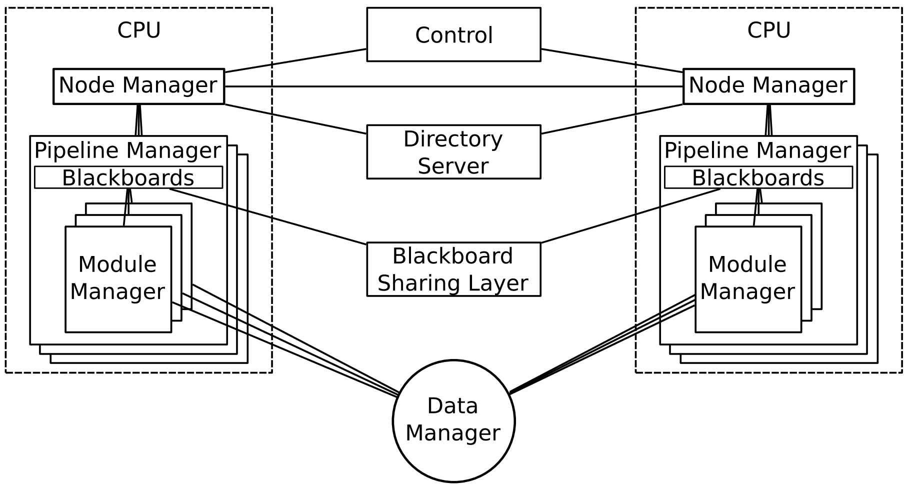


Figure 1: NHPPS Architecture components.

tion by adding new pipelines with the desired capabilities. A less obvious, though significant, advantage is the flexibility this decomposition provides in deploying work within a cluster of machines. Depending on the context, such as processing multiple exposures in parallel as opposed to processing multiple pieces of an exposure in parallel, one can optimize the utilization of the individual nodes for a particular purpose by controlling which pipelines are run on which nodes. This is a key concept behind the way the NHPPS supports multiple pipelines. It does not, however, provide a particular protocol or convention for how the pipelines interact. An example of a real pipeline application, including example protocols by which pipelines interact, is the NOAO Mosaic Camera Pipeline described by Swaters (2007) and Valdes (2007b).

3 NHPPS Core System

The NOAO High Performance System architecture consists of a Directory Server (DS) for a pipeline cluster, a Node Manager (NM) for each pipeline node, a Pipeline Manager (PM) for each pipeline on each node, a distributed Blackboard system for each pipeline, a Module Manager (MM) for each instance of each module in each pipeline, and control clients that interact with the system. Figure 1 illustrates this architecture and the communication connections. In this section we describe these architecture elements.

The NHPPS is able to distribute work across several machines, called nodes, within a processing cluster. In order to facilitate this, one node within the cluster runs a resource discovery server, called the Directory Server (DS), and all nodes in the cluster run a node management server, called the Node Manager (NM). Each NM contacts the DS and tells it that the node is available for processing. When data is being prepared for processing, a client program, `pipeselect`, queries the DS to determine which nodes are available, then determines which of those nodes will be given the processing task.

There is nothing in the logic of the NHPPS that requires more than one node. In this case, one can optionally eliminate the directory server and, in fact, if the pipeline application is run without starting the DS then it will automatically be confined to a single node. On the other hand, the DS allows new nodes to be added to a running pipeline with automatic resource discovery. Eliminating a node during execution works the same way but designing the pipeline application is more difficult because partial results and data may reside on the node. For cases of gradually shutting down an active node, operation protocols can be used to allow this to happen automatically.

The NHPPS may be run on a single machine, however it was designed to run on a multi-node processing cluster. Each node within the cluster runs a Node Manager and various pipelines within the Pipeline Application. It is important to note that not all pipelines in a pipeline application have to run on every node. In addition, there are components that provide services to the cluster. These are the Directory Server and, if used by the pipeline application, the Data Manager. The nodes and servers communicate with each other through sockets or through an RPC layer. In this section we discuss these components of the NHPPS, their socket messaging protocol, the Pipeline Description Language, and utilities which provide external interfaces.

3.1 Messaging

Our server components primarily communicate through a conventional socket interface. In our implementation the ports are externally configurable through user environment variables. This allows different users to develop and run pipelines on the same machines without conflict by simply assigning different values for the port variables. The port variables include both the host node and the port number so that the servers which serve a cluster, such as the DS and DM, can be easily changed. This is a feature which also allows easily exchanging the roles of cluster nodes when the cluster composition changes due to failure, removal, or addition of hardware.

One design decision we made for our high-level server components was to use a simple, ASCII-based messaging protocol layered on standard sockets. The protocol uses new-line delimited “keyword equal value” elements. The allowed keywords and values are defined for each type of message and server. There are a few reserved system class keywords that are available to all the servers. The most important of these is the COMMAND keyword which is used to call methods in the addressed server. In later sections we list the server methods whose names are the value string sent to the particular server with the COMMAND keyword.

This protocol was selected for its ease of implementation using text formatting and parsing routines found in common programming languages, allowing the protocol to be used across various languages. An added benefit is that this ASCII-based protocol allows an operator or programmer to monitor or debug the servers easily by sending and receiving packets using a terminal-based program such as telnet.

3.2 Directory Server

The Directory Server (DS) is responsible for maintaining a list of the nodes running the NM within a processing cluster. The NMs occasionally need to determine which nodes are in the processing cluster and this list is sent to the NM when requested. Note that the DS is not a communication intermediary. The NMs communicate with each other directly, the DS simply provides a registry service.

3.2.1 Externally Callable Methods

get_list, get_list_user, get_list_host are used to get the complete lists of all nodes in the pipeline cluster, all nodes with a NM running as a specific user, or all NMs running on a specific node.

add_user_host, del_user_host are used to register and de-register NMs with a DS.

3.3 Node Manager

The Node Manager (NM) is primarily responsible for starting, pausing, resuming, and stopping pipelines. Additional tasks include tracking available resources on a node, and communicating with the Directory Server (DS) to locate other available NMs within the pipeline processing cluster.

A NM is started on each node which is a member of the pipeline processing cluster on which the pipeline application is run. At startup, the NM contacts the DS to register itself as available for processing datasets. Note, this requires the DS to be running first. If no DS is running then the node will be unaware of other processing nodes, however it may still run a pipeline application restricted to a single machine.

To start a pipeline, the NM is sent a message from a client program, we have provided `runpipe` for this purpose, specifying a list of XML files which describe the pipelines to run and the number of instances of a pipeline to allow to run at a time. The NM, by convention, searches a directory specified by an environment variable for the PDL XML files and understands that the files will end in `.xml`. Therefore it is convenient to name the PDL files by the pipeline name. Doing so reduces running the pipeline to executing the `runpipe` command providing the desired pipeline names as arguments.

3.3.1 Externally Callable Methods

The NM, being one of the NHPPS servers, responds to messages which adhere to the communication protocol described in 3.1. When the request includes 'COMMAND=*method*' the NM executes the corresponding *method* below, returning its result to the caller.

start_pipe, stop_pipe start and stop Pipeline Managers on the node.

halt_pipe, step_pipe, resume_pipes control the state of execution of pipelines on the node.

get_load returns the CPU load on the node.

get_dir returns the 'data' directory on the node for a given pipeline, and optionally the amount of free disk space in that directory as well.

get_OpenDatasetCount_pipes gets the number of open datasets being processed under the given pipeline(s).

get_queue returns the number of datasets pending for processing on the given pipeline(s).

get_node_list returns a list of nodes in the pipeline cluster from the Directory Server.

test_osf returns a list of blackboard entries on the node.

cleanup_osf, cleanup_pstat are used to scrub clean the osf and pstat blackboards.

3.4 Pipeline Manager

The heart of the NHPPS is the Pipeline Manager (PM). In this architecture, each pipeline in a pipeline application has a corresponding PM on each node that runs the pipeline. The ability to control the number and location of pipeline instances is a key feature of our high-performance, distributed and parallel system. It is intuitive that instances of a pipeline running on multiple nodes provide distributed processing. What is less obvious is that running multiple instances, meaning multiple instances of each module in the pipeline, on the same node allows more efficient utilization of nodes with multiple CPUs for essentially the same reasons as with multiple machines.

Control of the number and location of pipeline instances provides the flexibility to customize a pipeline application to the problem and computing resources. In some cases a pipeline performing a high-level function, such as organizing many night's worth of data into datasets based on filter and night, needs only one instance on one node. In other cases a pipeline performing a data parallel function, such as processing a single piece of a larger format exposure, would have as many instances as there are pieces of the exposure or available nodes.

The PM is primarily responsible for parsing its pipeline's PDL file, setting up Module Managers for each Module in the pipeline, providing scheduling guidance to the Module Managers (MM), and creating shared Blackboards. The Module Manger is described in a following section but in outline it is responsible for a single pipeline stage. Architecturally we are describing a MM as a separate logical component. However, the PM operates as a single process, internally managing 'micro-threads' corresponding to instances of the Module Managers. These 'micro-threads' do not have significant context switch overhead. They therefore do not consume operating system resources, increasing the overall efficiency of the system.

A mentioned in the introduction, this micro-thread implementation for the MM is a key improvement over OPUS (Rose, 1995). In OPUS the equivalent of the MM is implemented as a separate polling process resulting in a potentially large number of processes. This is undesirable because the process table can become unmanageably large and the overhead in context switches can become significant.

3.4.1 Module Manager

Each instance of a Module within a pipeline has an associated Module Manager (MM) which performs two key tasks 1) checking that the events necessary for the Module to execute have occurred and 2) setting the environment and executing the Module's actions when the required events have occurred. The MMs operate concurrently. Therefore, all modules whose events are satisfied will initiate their associated actions and wait for their completion without blocking other MMs. This provides the desired coarse-grained parallel processing of different steps within a pipeline.

In data parallel processing it is common for the steps to become synchronized such that different datasets will be in the same processing stage at the same time on the same machine. We therefore allow multiple MMs, called instances, for each module in a pipeline. By providing multiple instances, we typically use one instance per CPU so that for a dual-CPU machine there are two instances, we find that the CPUs are best utilized with minimal idle time.

For the pipeline architect the main function of the MM is to execute a desired action which is typically a program. The program may be specific to a single module but often is more generic. In either case the program needs context information when it is run. The module manager provides this in two ways. Programs may be called with arguments, specified in the pipeline description language (PDL), and the MM supplies the command-line arguments when executing the program, translating any logical variables first. The MM also sets a number of standard environment variables which the program can then access. These include the dataset name or identifier, the pipeline name, the module name, the type of event, the module's blackboard flag when it is triggered, the start time, the process identification, and logical directories associated with the pipeline.

The basic functionality of the MM is implemented as a generator function; a programming technique where a function can return control to the caller and then be continued from where it left off. This means that, coupled with an appropriate scheduling service, Module Managers act as lightweight, micro-threads. The benefits of using generator functions are that they maintain their state between calls, return, or *yield*, at various points during execution and resume where processing left off the next time they are called. Additionally, they are run within the process space of the caller, in this case the PM, so they do not incur any operating system threading overhead.

The MM is particularly suited to a generator function as its two primary responsibilities may be broken into 4 distinct tasks which must be executed in sequence: 1) check the prerequisite events, 2) perform optional setup actions, 3) execute the primary action, and 4) perform optional cleanup actions, which leads to the the logical structure shown in figure 2.

Each time the `run` function is called, it executes until it reaches a `yield` statement. At that point, it returns to the caller. On subsequent calls, the `run` function starts execution immediately following the `yield` statement it ended with during the previous call.

The `run` function returns a value, `X`, to the caller when it encounters the `yield` statement. This value is a fraction of a second the MM wishes to be inactive before again being '`run`'. Typically, if the module's events have occurred, the value of `X` will be 0, meaning that the MM wishes to be `run` again as soon as possible to process a dataset. However, if the module's prerequisite events have not occurred, then the MM will more likely request some time to 'sleep' before checking the status of its prerequisite events again.

Figure 2: Pseudo-code for a MM generator function.

```
def run():
   while True:
       if eventsOccurred():
           runSetup()
           yield X
           runPrimary()
           yield X
           runCleanup()
       yield X
```

3.4.2 Scheduling

As previously mentioned, one of the tasks of the PM is to provide scheduling services to the MMs. In other words, the PM is responsible for calling the `run` function within each MM to ensure that the MMs which are not processing a dataset are checking the status of their prerequisite events, and the ones which are processing a dataset are performing their setup, primary, and cleanup actions.

The scheduling component within the PM keeps track of the times the `run` function in each MM was last called. The scheduling mechanism continuously loops over each MM in the PM, comparing the difference between the time it was last ‘`run`’ and the current time with the requested sleep time. If the difference is greater than the sleep time, the MMs `run` function is called, otherwise it is skipped.

After checking all of the MMs and running them as appropriate, the scheduling mechanism goes to sleep for a small period of time between loop iterations. The amount of time is configurable through the pipeline description language.

3.4.3 Blackboards

Blackboards store messages posted by the MMs. These messages may be requests for other MMs to begin processing data, status of current dataset processing, or even the status of the MM itself. Blackboards are visible to *all* of the MMs in the pipeline, and therefore MMs are able to, and do, ‘communicate’ with each other by posting and reading messages on the blackboards.

The PM creates two distinct blackboards in memory, which may optionally be mirrored on disk. These two blackboards track 1) the progress of individual datasets through the pipeline and 2) the status of pipeline modules to include, among other things, whether they are active or inactive, which dataset they are processing, and when they began processing.

As we have mentioned, these blackboards are shared among all of the MMs in the PM. This allows each module to check the status of the other modules in the pipeline.

Another function of the blackboard is to broadcast changes through a socket, the host and port of which are defined in the environment. It is not an error if the sockets host and port are not defined or if there is no client listening to the socket. Therefore, optional clients may be written to respond to blackboard events. There is currently one simple pipeline monitor client which can be connected directly to these messages or through a multiplexing server. These optional components are described in §4.2 and §4.4.

3.5 Pipeline Selection Utility

The pipeline selection utility, `pipeselect`, is a key tool in building distributed pipeline applications. The purpose of this utility is to discover available pipelines, rank their resources, and provide trigger information.

A module in one pipeline in one pipeline that wishes to trigger another pipeline must first discover the instances of the pipeline. It specifies the name of the desired pipeline, whether a list of the available pipelines is to be returned or a ranked list for a desired number of instances, and the required disk space available to the pipeline. In addition to the arguments, the utility uses a configuration file that currently defines whether a desired pipeline may be anywhere in the processing cluster, must be on the same node as the calling pipeline, or must be on any node other than the local node.

`pipeselect` contacts the local NM and requests a list of all the node managers it knows about. Normally the NM checks with the DS. Then using the information from the configuration file it contacts the potential NMs to find if they are running the desired pipeline. If a minimum disk space requirement is specified the list is limited to those nodes meeting the requirement. The trigger information for the pipeline consists of a directory path (in a network syntax) where a file trigger event may be created.

If a ranked list of a certain number of pipeline instances is requested `pipeselect` additionally filters the results based on the “load” of each pipeline. The measure of “load” is currently the number of “open” datasets in the pipeline. The list is then ordered from lowest to highest “load” and nodes with the same “load” are randomly shuffled. The returned list will then be as many trigger directories as the number of instances requested.

3.6 NHPPS host commands

The NHPPS provides an assortment of commands to begin, control, and end the execution of pipeline applications. The most important are outlined below.

plrun high level command to start a particular pipeline configuration.

plstatus reports the status of nodes, node managers, and pipelines.

pltest high level command to run a pipeline regression test.

plssh execute a pipeline command on all pipeline nodes.

servers starts the servers (NM, DS, etc. . .).

runpipe starts PMs for the specified pipelines.

osf_halt causes the pipeline to stop responding to triggering events.

osf_resume resumes normal pipeline activity after calls to either `osf_halt` or `osf_step`.

osf_step executes a single module of the pipeline.

osf_test lists the contents of the blackboard.

osf_update sets the status flag for a blackboard entry.

stopall kills all pipeline related processes on the local node.

stoppipe stops PMs of given pipelines.

stopserver kills servers (NM, DS, etc. . .).

pipeselect performs load-balancing while determining which nodes should process datasets. processes.

3.7 Pipeline Description Language

The Pipeline Description Language (PDL) is used to define the pipelines which create a pipeline application. It is implemented as an XML grammar. The language is specialized for describing the coarse-grained, event-driven logic discussed earlier. This language is one significant improvement we have made over the earlier OPUS implementation of the same methodology.

The fundamental component of the PDL is the module description. As noted below, common elements may be described at a higher level but ultimately these act in the flow of interpreting each module. In outline, a module description consists of prerequisite conditions, optional setup actions, the primary action of the module, and optional cleanup actions which may be executed based on the exit status of the primary action.

A pipeline is a collection of modules that define *settings* and *actions* which are executed when certain *events* occur. The modules typically reference each other within a pipeline; for instance, to specify that a previous module must be completed before another module may start. The natural use of logical module names rather than index numbers is a significant improvement of the language in comparison to the similar OPUS methodologies.

The PDL provides three levels – Application, Pipeline, and Module – for the pipeline architect to define the *actions*, *events*, and *settings*. This allows defining *actions*, *events*, and *settings* which are common to all of the modules in a pipeline application or all of the modules in a single pipeline in one place rather than duplicating these definitions in each module.

3.7.1 Settings

The PDL defines several control settings which modify the execution of the pipeline application. Among these settings are two which ensure that there are sufficient disk space resources to execute a module's actions and that the module's actions do not become hung or runaway processes. These are critical in a pipeline application to prevent a process from consuming significant amounts of system resources as a runaway process or hanging the pipeline application by preventing other modules and datasets from running and also to ensure that processes which require significant amounts of disk space are not run on a node with insufficient resources. The setting for detecting processes which execute longer than expected allows the architect to catch this situation and take action, such as notifying the operator and letting them diagnose the problem.

3.7.2 Actions

Actions perform the work of the pipeline. The PDL defines several actions which perform bookkeeping functions on the blackboard as well as a couple which handle trigger file management. Additionally, and more importantly, is what we call the *Foreign* action which is used to launch any program or script on the host system. This provides the pipeline application significant flexibility in the design of the programs which perform the core processing as they may be written in any language or data analysis package which is callable from the operating system.

The NHPPS is designed to perform up to 3 different sets of actions within a module: Setup, Primary, and Cleanup.

Setup actions are performed before the Primary action and serve to setup the environment necessary for the Primary action, perform bookkeeping actions on the blackboard, etc.

The Primary action is the action the module is written to execute. There is only one Primary action in each module and it is typically a *Foreign* action.

The Cleanup actions are sets of actions, one of which may execute based upon the Exit Code event after the Primary action is completed.

3.7.3 Events

As mentioned previously, the NHPPS is event-driven. The events which trigger the start of execution for a module are Time, File, and Status events. There is a special type of event, the Exit Code event, which may be used to trigger optional Cleanup actions in the module.

Time events repeat at fixed intervals between a start and end time, or occur only once at a specified time.

File events occur when a file matching a search pattern is found within a given directory. These files are called ’trigger files’ within the NHPPS and are a very useful method for starting the flow of execution within a pipeline due to their ease of creation, most commonly using `touch` from Unix or IRAF.

Status events occur when the blackboard status for a particular module matches a certain value.

Exit Code events are generated when a modules primary action is completed. This value is typically the value returned from the modules primary action. It may be used to instruct the module to perform a set of cleanup actions, based on the value of the Exit Code. This functionality allows a module to execute one of a number of different sets of cleanup actions based upon the Exit Code. By specifying different sets of cleanup actions, the module is able to handle cases where the primary action ended successfully, with a partial result, or failed.

3.7.4 Application Level

In many cases, the pipeline architect will find it convenient to define settings for the minimum amount of disk space required for a module to execute, or the length of time to provide a module to execute in one place, as opposed to defining these values in each module. For this reason, as well as to easily define Setup and Cleanup action sets that are used by all modules in a Pipeline Application, the NHPPS allows the pipeline architect to define these in a single XML file which is merged into each of the pipeline XML files at runtime.

3.7.5 Pipeline Level

For the same reasons the NHPPS provides an Application Level for Pipeline Application global definitions, pipeline specific global definitions may be defined in the pipeline's XML file. The settings within the Pipeline Level take precedence over those in the Application Level.

3.7.6 Module Level

The Module Level is where the actions and settings of a single step of the pipeline are defined. There are typically several modules within a pipeline and several pipelines within a pipeline application. Any setting in the Module Level takes precedence over the settings in the Pipeline or Application Levels.

3.8 Performance

A key requirement of the NHPPS infratructure is that they use minimal CPU resources. In other words, the goal of the infrastructure must be to allow the applications that it supports to take most of the CPU resources. The quiescent state of the NHPPS when it is simply waiting for data to enter a pipeline application, in this case the NOAO Mosaic Camera Pipeline, has a Linux kernel reported load less than 0.01 on the NOAO pipeline cluster of dual CPU 3.0 GHz Xeons. Note that this includes a large set of pipelines running as shown in figure 3. The overhead when data is flowing through the pipeline is hard to quantify but we believe it to still be minimal.

4 NHPPS Optional Components

In this section we briefly describe optional infrastructure components which were developed to work with the NHPPS and pipeline applications. A key design philosophy of the NHPPS is to minimize tight coupling and emphasize simple, weakly coupled optional components. By simple and weak coupling we primarily mean interprocess communication with our simple messaging protocol (§3.1). This allows easy addition of new or enhanced components. By optional we mean that either configuration or environment information determines if a component is to be used or failure to connect to a component is not an error.

Figure 3: Mosaic pipeline cluster status example.

```
Pipeline node status report: Fri Sep 29 17:09:09 2006
Current node is pipedmn

Node        NodeMgr     Available Pipelines
-----------------------------------------------------
pipedmn     Running     dir, dps, dts, ftr, ngt
-----------------------------------------------------
pipen01     Running     cal, day, frg, mdp, mef, pgr,
                        rsp, scl, sft, sif, xtc, xtm
-----------------------------------------------------
pipen02     Running     cal, day, frg, mdp, mef, pgr,
                        rsp, scl, sft, sif, xtc, xtm
-----------------------------------------------------
pipen03     Running     cal, day, frg, mdp, mef, pgr,
                        rsp, scl, sft, sif, xtc, xtm
-----------------------------------------------------
pipen04     Running     cal, day, frg, mdp, mef, pgr,
                        rsp, scl, sft, sif, xtc, xtm
-----------------------------------------------------
pipen05     Running     cal, day, frg, mdp, mef, pgr,
                        rsp, scl, sft, sif, xtc, xtm
-----------------------------------------------------
pipen06     Running     cal, day, frg, mdp, mef, pgr,
                        rsp, scl, sft, sif, xtc, xtm
-----------------------------------------------------
pipen07     Running     cal, day, frg, mdp, mef, pgr,
                        rsp, scl, sft, sif, xtc, xtm
-----------------------------------------------------
pipen08     Running     cal, day, frg, mdp, mef, pgr,
                        rsp, scl, sft, sif, xtc, xtm
-----------------------------------------------------
```

4.1 Data Manager

The Data Manager (DM) is a server that interfaces distributed applications to data management services. Communications is through a socket using our simple messaging protocol. In our system the Data Manager provides access to two primary services; a database, called the Pipeline Metadata Archive System (PMAS), and a Calibration Library.

The primary purpose of the DM is to support pipeline applications. It is a key component of the NOAO pipelines. A secondary purpose is to support management of processing information. Pipeline modules may be configured to communicate their status to the PMAS. In this configuration, the PMAS logs the start and end of module execution along with information about the triggers and nodes. This information may be used for diagnostics, reports, and various pipeline module purposes.

The current DM interfaces to a single database system. The database supports both the calibration library and the Pipeline Metadata Archive System. The cuurent DM also includes support for distributed proxy DMs which maintain copies of the data synchronized with a master DM. This allows an integrated data management system when pipeline applications run in several locations, such as at a telescope and at a data processing centers, and improves the scalability of this function for a large number of pipeline nodes needing frequent access to the DM.

As with the core components, there are a number of client programs and methods associated with this component.

4.1.1 Pipeline Metadata Archive System

The Pipeline Metadata Archive System (PMAS) serves the role of a metadata management system for the distributed pipeline system and applications. Access is through the data manager server interface and clients. It is ultimately a general SQL database. There are tables specific logging the status of modules in a pipeline as well as tables specific to the pipeline application.

The data manager server provides convenience methods for getting and putting information as well as a method to pass through SQL queries and return one or more records. The pipeline system provides client applications for getting and putting information from pipeline modules.

4.1.2 Calibration Library

The purpose of the calibration library is to maintain calibration information indexed by a set of attributes used to select an appropriate calibration for a particular observation. There are two distinct types of calibration information; parameters and files. The calibration library therefore requires two elements; a database and a file repository. The database has multiple roles. It provides the indexing as well as the calibration parameter information.

The indexing attributes in our calibration library are: detector, image identifier, filter, a quality rank, and starting and ending valid dates. The detector and image identifier attributes are needed to support multiple instruments and mosaics and the quality rank is used to give greater weight to calibrations which have been deemed of higher quality.

The date attributes are a fundamental characteristic of a calibration library. It allows evolution of the calibrations due to changes in the hardware and performance of the instrument. There are two main types of date dependence which occur in our library. One is for self-calibrations needed to account for day-to-day variation in the characteristics of the detector. Typical examples of this are bias and dome flat calibrations. The other is for significant changes in the hardware, such as replacement of a detector, for changes in the processing recipe, such as processing rules, and improvements in the instrument characterization. As part of the query to the calibration library the date of the observation is provided. Only calibrations with date ranges that include the observation are considered. When multiple choices are available the quality ranking and proximity to the middle of the range are used to select the best calibration. In future releases, the data manager will be enhanced to provide more sophisticated selection rules.

The calibration library supports an arbitrary set of calibration classes. The required classes are defined by the needs of the pipeline. Examples of classes are dome flats and crosstalk coefficients. A typical query is for a particular class given a date and date range, detector, image identifier, and filter. The result of the query is to return the parameter value or the path, in network syntax, to the client. For files there is additional information provided consisting of the file size and modification time. The purpose of this extra information is to allow the client to maintain a cache and only "pull" a copy from the calibration file repository when it does not have a current copy.

Our system provides client applications for getting and putting calibration information through the data manager server interface. The applications are used within pipeline modules. The "get" application implements a local cache to minimize file transfers. It may be driven from images where the selection attributes are determined from the image header and the return file and parameter information are recorded in the header.

4.1.3 Externally Callable Methods

findid finds the 'best' id of the requested object.

getcal, putcal get and put calibration files from or into the calibration library.

getkeywordvalue, getkeywordvalues return all of the values of a given keyword from the 'best' object which matches in the database.

get_repository returns the DMs repository.

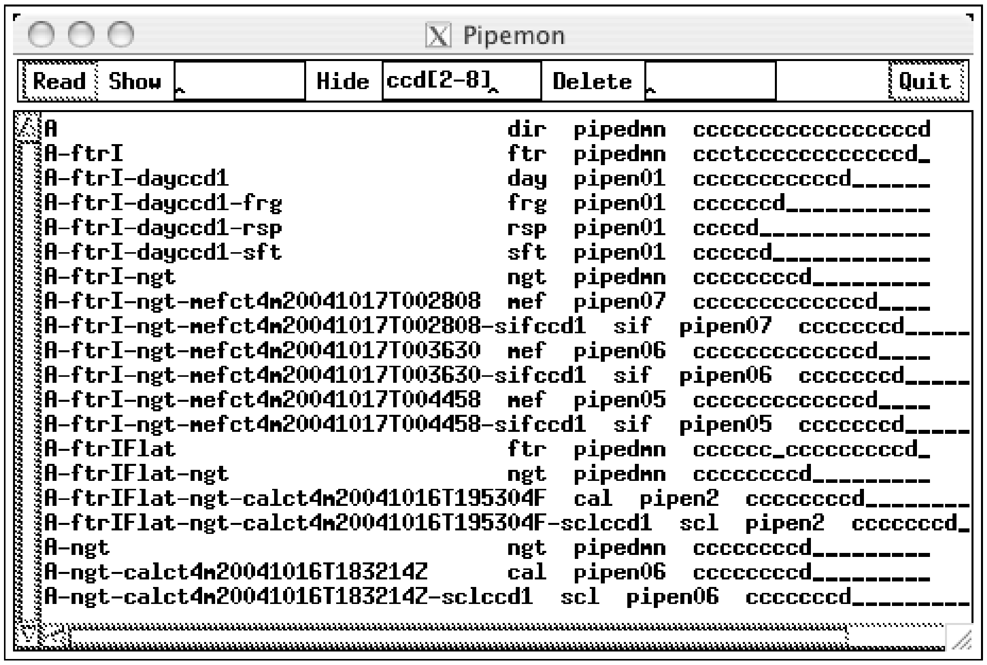
findrule searches through a set of rules and returns any which match.

is_mastermode allows a client to determine if this DM is running in 'master' mode.

newdataproduct, newprocess, finishprocess provide clients the ability to populate the PMAS database with relevant information.

setkeywordvalue sets a keyword value for the indicated object in the DM.

sql_query, sql_update provide clients with the capability to perform remote SQL queries and updates of the DM database.



```
A                                        dir  pipedmn  cccccccccccccccccd
A-ftrI                                   ftr  pipedmn  ccctccccccccccccd_
A-ftrI-dayccd1                           day  pipen01  cccccccccccd______
A-ftrI-dayccd1-frg                       frg  pipen01  ccccccd___________
A-ftrI-dayccd1-rsp                       rsp  pipen01  ccccd_____________
A-ftrI-dayccd1-sft                       sft  pipen01  cccccd____________
A-ftrI-ngt                               ngt  pipedmn  ccccccccd_________
A-ftrI-ngt-mefct4m20041017T002808        mef  pipen07  cccccccccccccd____
A-ftrI-ngt-mefct4m20041017T002808-sifccd1  sif  pipen07  cccccccd_____
A-ftrI-ngt-mefct4m20041017T003630        mef  pipen06  cccccccccccccd____
A-ftrI-ngt-mefct4m20041017T003630-sifccd1  sif  pipen06  cccccccd_____
A-ftrI-ngt-mefct4m20041017T004458        mef  pipen05  cccccccccccccd____
A-ftrI-ngt-mefct4m20041017T004458-sifccd1  sif  pipen05  cccccccd_____
A-ftrIFlat                               ftr  pipedmn  cccccc_cccccccccd_
A-ftrIFlat-ngt                           ngt  pipedmn  ccccccccd_________
A-ftrIFlat-ngt-calct4m20041016T195304F   cal  pipen2  ccccccccd________
A-ftrIFlat-ngt-calct4m20041016T195304F-sclccd1  scl  pipen2  cccccccd_
A-ngt                                    ngt  pipedmn  ccccccccd_________
A-ngt-calct4m20041016T183214Z            cal  pipen06  ccccccccd_________
A-ngt-calct4m20041016T183214Z-sclccd1  scl  pipen06  cccccccd_________
```

Figure 4: Example of the pipeline monitor showing processing from the Mosaic Pipeline. The first column is the dataset name (the root name is 'A' though in practice there is a more descriptive name), the second column is the pipeline, the third column is the node, and the fourth column are the status flags for each stage. Typically a successfully completed stage is 'c' and a successfully completed pipeline ends with 'd'. Only pipelines involving the first CCD of the mosaic are shown.

4.2 Pipeline Monitor

An essential operational tool for a pipeline application is a *pipeline monitor* that shows the state of the processing and indications of errors. A full featured monitor that provides convenient access to all the pipeline processing information, such as the blackboards, node resources, processing logs, etc., is a major undertaking. Currently we have implemented a simple monitor for the pipeline blackboards. Figure 4 shows an example of processing from the Mosaic Pipeline (see Valdes (2007b) for more detail).

The blackboards are the most important element defining the pipeline processing status. This was described in §3.4.3. We have found that the most efficient and lowest impact way to monitor the blackboards is to query all the blackboards for all the datasets and pipelines only to initialize or reload the monitor and thereafter respond only to update messages. A query for the all the distributed blackboard information, especially since this information can become quite large after days and weeks of processing, is fairly slow.

4.3 Message Monitor

The *Message Monitor* is a graphical tool to receive messages on a socket and display them in a scrollable and sortable window. The messages include a severity, a time, the source of the message, and the message. This is not a particularly novel client but is important operationally. Currently the NHPPS core components write messages to disk files and do not make use of a message socket. However, the message socket is heavily used by the current pipeline applications and it will be a natural evolution to direct NHPPS messages to the same socket so that they can be monitored with one or more message user interfaces.

4.4 Switchboard Server

The *switchboard server* is a general tool for connecting any number of input sockets to any number of output sockets. The connections can be defined statically or dynamically. Clients may contact the switchboard server and *subscribe* to messages from particular input sockets and either receive them back on the same connection or redirect the output to another socket. The design is such that it is not an error for clients to join and leave the switchboard.

Another feature of the switchboard server is to archive messages which can then be spooled back to new clients. This is an optional feature which we normally do not use.

The pipeline monitor is one application that typically uses the switchboard server. We have developed other prototype clients that also benefit from this server.

The switchboard server is an optional component since applications, such as the pipeline monitor, may be directly connected without the switchboard as an intermediary. However, the purpose and advantage of the switchboard is to multiplex and log the message streams. For example, without the switchboard only a single pipeline monitor can be used at a time. Utilizing the switchboard allows multiple pipeline monitors or, in the future, additional types of monitors, to display the messages at the same time. This allows the operator to check the status of the pipeline from multiple locations, even remote sites. Note that, as optional components, the switchboard server and clients connected to the switchboard or directly to the pipeline application may come and go without affecting the processing.

5 Implementation

This paper describes a version identified as NHPPS V1.0 which is a stable, production system. We continue to refine the implementation and add new features and components.

The NHPPS core components and clients are primarily written in Python, although we also utilize shell scripts for some simple clients. One especially important feature of Python is that it has direct support for generator functions making it particularly easy to implement the lightweight micro-threaded MM scheduling algorithm described earlier.

We have made use of the following third-party Python packages: EaseXML and Pyro. EaseXML is an XML parser which we use during the parsing of our PDL XML files. Pyro is an RPC layer which allows local access to remote Python objects. This capability is used to implement the

'Blackboard Sharing Layer' which permits the pipelines in a pipeline application to see a single, unified, blackboard across all pipelines as opposed to just the local blackboard for the individual pipeline.

The Data Manager is also implemented in Python and makes use of Python packages for sockets, SQLite for the database engine, and the host file system for the calibration file repository. Some of the key client programs are written in IRAF/SPP as well as Python. However, the client programs communicate using the simple NHPPS protocol (§3.1) and can easily be translated to other languages. The Switchboard Server and the Pipeline Monitor are IRAF applications. The Message Monitor is a Python TK application.

The NHPPS core system, optional components, and the Mosaic Camera Pipeline application (Valdes, 2007b) have been developed and deployed on a cluster of commodity dual-CPU machines running Linux. The design of the components is such that other Unix-based operating systems can be used. In fact the cluster can be heterogeneous. In the future we expect to migrate our system and pipeline applications to other hardware and operating systems.

The NHPPS has not (yet) been packaged as a software product nor do we have the resources to provide in-depth support. However, the core system is designed and structured with this goal in mind. We are open to evaluation and collaborative usage within our limited resources.

References

Boulade, O., et al., 2003, procspie, 4841, 72

Cline, T., Pierfederici, F., Swaters, R., Thomas, B., & Valdes, F., 2007, ASP Conf. Series, in preparation Pierfederici, F. & Miller, M., 2007, ASP Conf. Series, in preparation

Jacoby, G., et. al., 2002, procspie, 4836, 217

Kaiser, N., et al., 2002, procspie, 4836, 154

McLeod, B., Conroy, M., Gauron, T., Geary, J., & Ordway, M., 2000, Proceedings of the International Conf. on Scientific Optical Imaging, Cambridge: Royal Society of Chemistry

Muller, G., 1998 procspie, 3355, 577

Rose, J., et al., 1995, ASP Conf. Ser., 77, 429

Swaters, R. & Valdes, F., 2007, ASP Conf. Series, in preparation

Tyson, J., 2002, procspie, 4836, 10

Valdes, F. & Swaters, R., 2007a, ASP Conf. Series, in preparation

Valdes, F., Swaters, R., Pierfederici, F., Miller, M., & Zarate, N., 2007b, pasp, in preparation

Valdes, F., Swaters, R. & Dickinson, M., 2007c, pasp, in preparation

Wester, W., 2005, ASP Conf. Series, 339, 152