Kepler Science Operations Center pipeline framework

Todd C. Klaus*[a], Sean McCauliff[a], Miles T. Cote[b], Forrest R. Girouard[a], Bill Wohler[a], Christopher Allen[a], Christopher Middour[a], Douglas A. Caldwell[c], Jon M. Jenkins[c]
[a]Orbital Sciences Corporation/NASA Ames Research Center, MS 244-30, Moffett Field, CA, USA 94035-1000
[b]NASA Ames Research Center, MS 244-30, Moffett Field, CA, USA 94035-1000,
[a]SETI Institute/NASA Ames Research Center, MS 244-30, Moffett Field, CA, USA 94035-1000;

ABSTRACT

The *Kepler* mission is designed to continuously monitor up to 170,000 stars at a 30 minute cadence for 3.5 years searching for Earth-size planets. The data are processed at the Science Operations Center (SOC) at NASA Ames Research Center. Because of the large volume of data and the memory and CPU-intensive nature of the analysis, significant computing hardware is required. We have developed generic pipeline framework software that is used to distribute and synchronize the processing across a cluster of CPUs and to manage the resulting products. The framework is written in Java and is therefore platform-independent, and scales from a single, standalone workstation (for development and research on small data sets) to a full cluster of homogeneous or heterogeneous hardware with minimal configuration changes. A plug-in architecture provides customized control of the unit of work without the need to modify the framework itself. Distributed transaction services provide for atomic storage of pipeline products for a unit of work across a relational database and the custom Kepler DB. Generic parameter management and data accountability services are provided to record the parameter values, software versions, and other meta-data used for each pipeline execution. A graphical console allows for the configuration, execution, and monitoring of pipelines. An alert and metrics subsystem is used to monitor the health and performance of the pipeline. The framework was developed for the *Kepler* project based on *Kepler* requirements, but the framework itself is generic and could be used for a variety of applications where these features are needed.

Keywords: Kepler, pipelines, distributed processing, cluster computing, photometry, pipeline, framework

1. INTRODUCTION

The *Kepler* pipeline framework is a generic platform designed for building applications that run as automated, distributed pipelines. The *Kepler* SOC[1] uses this framework to build the pipelines that process the ~23 GiB of raw pixel data downlinked from the *Kepler* spacecraft per month on four computing clusters containing a total of 64 nodes, 512 CPU cores, 2.3 TiB of RAM and 148 TiB of raw disk storage[2].

A key consideration when designing a distributed pipeline application is how to define the unit of work for each job that is processed on the cluster. For some applications, the unit of work remains constant throughout all of the modules that make up the pipeline. For example, in an image processing pipeline, the unit of work might be a single image. The image is modified as it passes through the various modules of the pipeline, but one image comes out of the end of the pipeline for every image that goes in. For this type of application, the pipeline execution logic can be fairly simple. For each image to be processed, a single job is executed for each module in the pipeline.

For other applications, the nature of the unit of work changes from module to module. For example, in a photometric data reduction pipeline, a set of images flow into the pipeline, but a light curve for each star comes out the other end. Even in this case, there can be a single job for each module if the module that performs the aperture photometry performs this task for all stars in the image as a single job, preserving the simple model of one job per module. However, as the size of the data set grows, it may make more sense to partition the data to create multiple jobs per module so that they can run in parallel on the cluster. The *Kepler* science pipelines fall into the latter category.

The *Kepler* photometer[3] consists of an array of 42 CCDs, each with 2 output amplifiers, for a total of 84 'module/outputs.' To keep the solar panels facing the sun, the spacecraft rotates 90 degrees about the optical axis every 90 days, changing the group of stars that fall on each module/output. For the front end of the *Kepler* pipeline

(calibration[4], photometric analysis[5], and pre-search data conditioning[6]), it makes sense to partition the data into 84 units of work (one per module/output), and then further partition these units temporally (at most 90 days per unit of work), while for the back end of the pipeline (transiting planet search[7] and data validation[8]) each target can be processed independently, and all available time samples are needed. In particular, the data validation module is designed to run only on those targets that exceeded the detection threshold in transiting planet search, so the unit of work cannot be predefined. These use cases[9] imply that the framework needs to support a different unit of work type for each pipeline module, and the unit of work for one module may be dependent on the results of a previous module.

The *Kepler* pipeline framework attempts to maximize the level of customization for applications by allowing them to specify how the unit of work is defined for each pipeline module while minimizing the amount of custom code that must be written. The application need only supply the code that generates the unit of work descriptors (known as the unit of work generator) for each module and the code for the modules themselves. The framework takes care of turning the unit of work descriptors into pipeline jobs (one job per descriptor), executing and monitoring those jobs, and automating the transitions between modules, with support for different transition types (see section 4). Modules need only process the data specified by the unit of work descriptor, they do not need to know which module needs to be run next, or even where they sit in the pipeline configuration. Modules read and write their data to/from a common data store, they do not communicate directly with each other[9].

In addition to the customizable unit of work, the framework also provides parameter management features. The application can define parameter sets, where each parameter set contains a collection of parameters for a specific purpose. These parameter sets can then be used to control the behavior of the application-specific pipeline modules and unit of work generators. The framework keeps track of revisions to the parameters for data accountability purposes. By integrating these elements into the framework in a generic way, the framework provides a comprehensive data accountability record for each job while minimizing the amount of application code. Each pipeline data product can be tagged with a single identifier (job ID) that leads back to the complete data accountability record for that product, including parameters used, software version used, versions of various models used, the unit of work that it was part of, and the entire state of the pipeline configuration at the time the job executed.

The framework is object-oriented; applications extend framework classes and implement framework interfaces to build pipelines. The framework uses a push model; application code typically does not call framework APIs, rather the application implements framework interfaces and the application code is invoked by the framework as needed (e.g., to generate unit of work descriptors or to process a job). While the framework is designed to enable distributed processing on clusters of many nodes, it also scales down to a single workstation for development and test scenarios. The framework builds on many existing open-source components (JMS[10], Hibernate[11], Apache Commons[12], JBoss JTA[13]).

2. PLUGGABLE ARCHITECTURE

As alluded to above, there are three main plug-in points where application-specific classes extend the framework. The `UnitOfWorkGenerator` and `UnitOfWorkDescriptor` interfaces allow customization of the unit of work, the `PipelineModule` abstract class allows for implementation of pipeline modules, and the `Parameters` interface allows for the definition of the parameters used by the pipeline modules and unit of work generators. The figure below illustrates this, using the *Kepler* data validation module as an example.

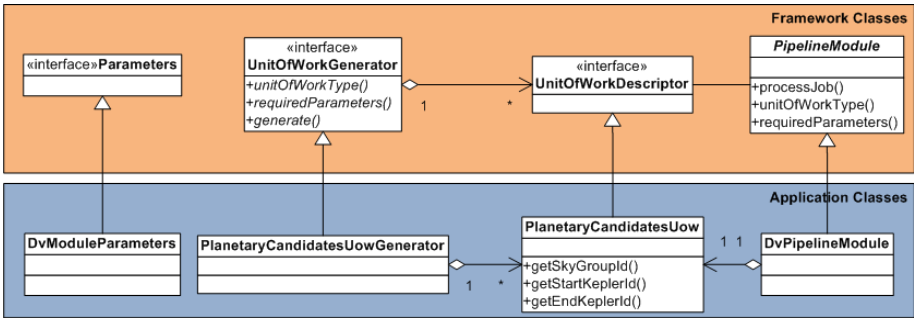

Figure 1: Framework Extension Points

2.1 Pipeline Modules

The `PipelineModule` abstract class provides the entry point for the module. To implement a new module, the application creates a new class that extends `PipelineModule` and implements the abstract methods. When processing a job, the framework will create an instance of this class and invoke it via the `processJob` method, passing it the unit of work descriptor and any parameters configured for the module. This class also contains methods that the application can use to specify the parameters required by the module and the expected unit of work type. These methods are used by the framework to enforce valid configurations in the configuration GUI.

2.2 Unit of Work Generators

As mentioned previously, in order to enable distributed, parallel processing for pipeline modules, the framework needs to be able to generate multiple jobs for each module at run time, with one job per unit of work. However, deciding how many jobs to generate and the specification of the unit of work for each job is delegated to a custom unit of work generator provided by the application. This allows custom definitions of the unit of work to be developed and configured in the pipeline without changes to the pipeline framework code itself.

The application can implement the `UnitOfWorkGenerator` and `UnitOfWorkDescriptor` interfaces to define custom unit of work types. The `generate` method of `UnitOfWorkGenerator` is responsible for generating a collection of objects that implement the `UnitOfWorkDescriptor` interface. These objects contain the unit of work descriptors, and the framework turns each of these descriptors into a job to be executed on a cluster node.

Once these classes are implemented, the pipeline operator can configure a module via the graphical console to use this unit of work generator. In order to assign a generator to a module, the associated module must accept the `UnitOfWork` type produced by the generator. The framework enforces this configuration constraint by verifying that the type returned by `UnitOfWorkGenerator.unitOfWorkType` matches the type returned by `PipelineModule.unitOfWorkType`.

Generators can also use parameters from the parameter library to control their behavior. The `generate` method takes as an argument all of the parameter sets configured in the trigger for the module by the operator. The framework validation logic verifies that all required parameter sets are configured correctly prior to launching a pipeline.

The `UnitOfWork` interface is simply a marker interface as it contains no methods or fields. It serves as a common base class for all classes that implement the interface so that objects of this type can be handled and persisted generically by the framework. The concrete classes contain the actual specification for the unit of work. For example, the unit of work for the *Kepler* calibration module contains the CCD module number, the CCD output number, the start time, and the end time[9].

2.3 Parameters

The framework provides parameter management services for application parameter classes that implement the `Parameters` interface, described in further detail in section 3, below. The framework persists the parameters in the database as part of the data accountability record and passes them to the pipeline modules as part of job execution.

3. DATA MODEL

The framework uses a relational database to store pipeline configurations as well as the meta-data created during pipeline execution, as shown in the class diagram below.

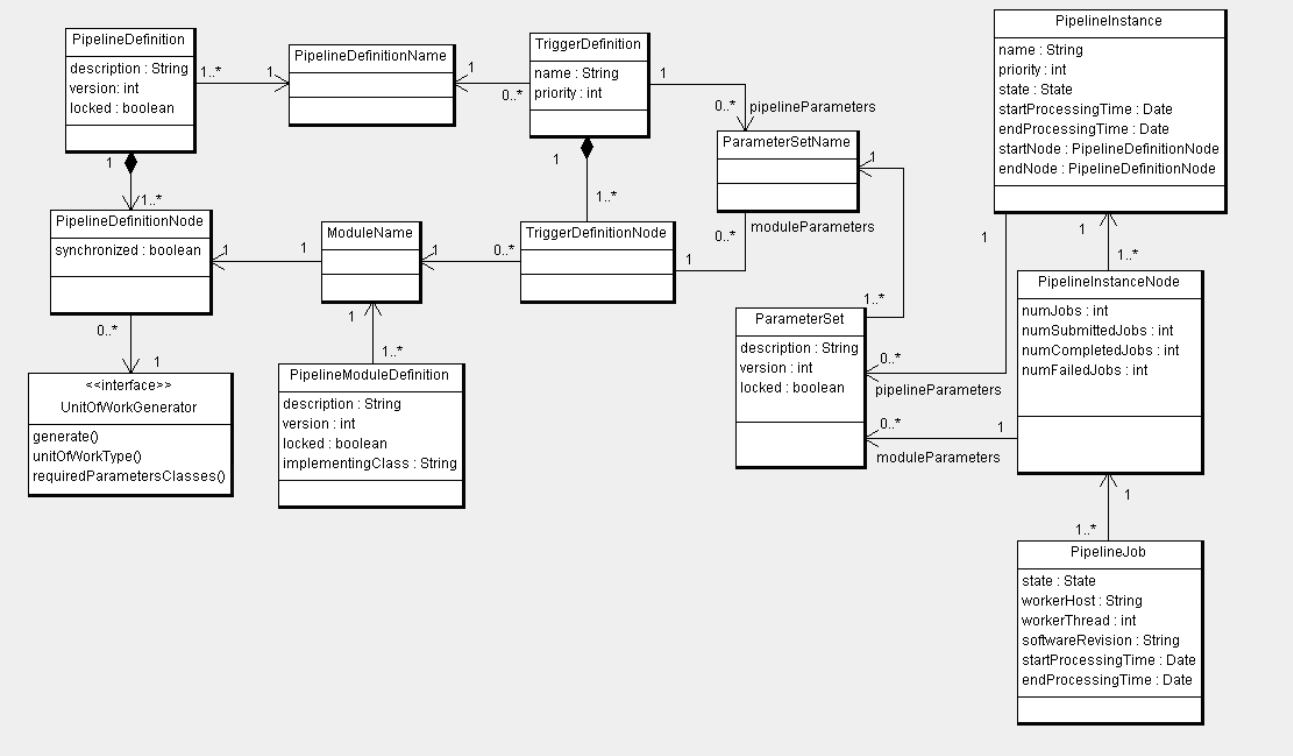

Figure 2: Framework Data Model

The entities on the left represent the various elements of pipeline configurations, including pipeline definitions, the module library, the parameter library, and the trigger library. Pipeline definitions specify the modules that make up a pipeline, their execution order, and the unit of work generator for each module. These configuration objects employ a versioning mechanism (described below) that ensures that the state of the configuration as it existed at the time each pipeline instance ran is preserved for data accountability purposes.

The entities on the right (`PipelineInstance`, `PipelineInstanceNode`, and `PipelineJob`) store the meta-data created by the framework during pipeline execution. These objects include the parameters used by the modules in the executing pipeline, the unit of work descriptors, and the details about each job, like what cluster node it ran on, the version of the software used, and the current state. The meta-data also includes the current state of the data model registry, a framework entity that keeps track of the current versions of various data models used by the pipeline modules in a generic way (by name and version). Examples of data models tracked in this registry by *Kepler* include models that describe various characteristics of the *Kepler* photometer, like electronic noise levels and optical vignetting, but any model with a unique name and version can be tracked in this fashion. Linking the state of this registry to the pipeline instance meta-data makes it possible to determine the exact state of the models at the time any data product was generated.

The execution meta-data objects, along with the versioned configuration objects, together form a complete data accountability record for every pipeline job. The job identifier alone can be used to trace back to all of these data model elements, so by storing the job identifier along with the data produced by the job, the complete pedigree of the data is preserved, allowing the data to be reproduced at a later time by reproducing the environment (parameters, software version, etc.) that originally generated them.

These framework entities are stored in a relational database and accessed via an Object/Relational Mapping (ORM) layer implemented with Hibernate, an open-source ORM framework. Because of the Hibernate layer, the underlying relational database can be changed without changes to the code. Use cases include Oracle for large cluster deployments and HSQLDB[14] for single-workstation deployments, although the choice of relational database is limited only by the list of databases supported by Hibernate.

These entities are managed using either a graphical console, an XML import/export interface, or via a programmatic interface.

3.1 Pipeline Definitions

A pipeline definition is a tree data structure where each node in the tree is associated with a module definition from the module library. While each node contains bi-directional references (parent and children), the tree can be considered a directed tree for execution purposes, since execution always progresses from the root node towards the leaf nodes.

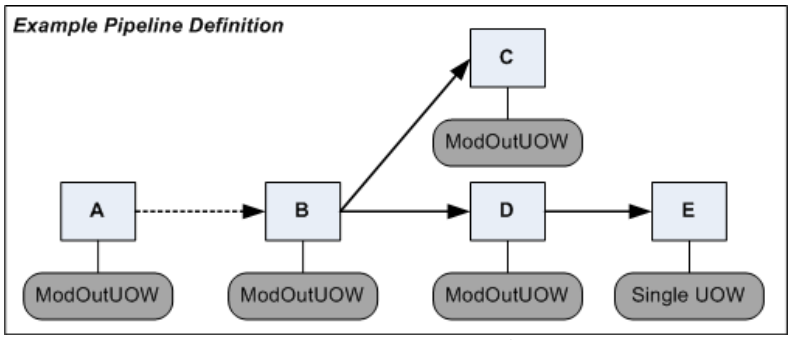

Figure 3: Sample Pipeline Definition

Figure 3 shows an example `PipelineDefinition` with a tree of `PipelineDefinitionNode` objects, where **A** is the root node. Execution starts with the root node (**A**) and proceeds towards the child nodes. In cases where a node has multiple children, all of the children execute in parallel. In this example, the jobs for **C** and **D** execute in parallel following completion of the jobs for **B**. Execution of a node actually consists of executing all of the jobs generated for the node, as described in more detail in section 4, below.

Each `PipelineDefinitionNode` must specify a unit of work generator. The generator is specified using the fully-qualified name of the Java class that implements `UnitOfWorkGenerator`.

3.2 Module Library

A module definition specifies the fully-qualified name of the Java class that implements the module. This class must extend the abstract class `PipelineModule`. The module definition may also contain the name of an external executable (such as a MATLAB executable in the case of many of the *Kepler* modules) that is invoked as part of the job processing. Module definitions configured in the database are known collectively as the 'module library.' Individual module definitions are shared by the pipeline definitions that use them.

3.3 Parameter Library

The parameter library is a collection of parameter sets. Each set contains a collection of typed parameters represented by an application-provided Java class that implements the framework `Parameters` interface. Some parameter sets are used by unit of work generators to control how the work is distributed across individual cluster nodes. Other parameter sets are used to control the behavior of the algorithms that make up the pipeline modules.

Each parameter set is represented by a Java class that implements the `Parameters` interface. The `Parameters` interface is simply a marker interface, so it contains no methods or fields and serves only to identify the semantics of being a parameter set.

3.4 Trigger Definitions

Pipeline triggers are used to launch new pipeline instances. The main purpose of a trigger is to maintain a mapping between the pipeline definition that will be used to create the new pipeline instance and the parameter sets that will be used for that pipeline instance. The main purpose of triggers is to allow the operator to pre-configure the pipeline so that launching the pipeline is as simple as firing the trigger. Triggers can be fired by the operator using the graphical console, or they may be fired automatically, such as when a data ingest completes.

Each trigger contains a list of nodes with one node per module in the referenced pipeline definition. The purpose of the trigger nodes is to maintain a binding to the parameter sets that will be used by the pipeline instance when the trigger is fired. The trigger also contains the priority that will be used for the new pipeline instance.

Each pipeline module implements the `requiredParameterTypes` abstract method of `PipelineModule` to specify the parameter set types required at run time. The trigger satisfies this requirement by specifying (by name) which instance of each parameter set type will be used.

Parameter sets can be bound to the trigger at the top level (`TriggerDefinition`) or module level (`TriggerDefinitionNode`). Parameter sets that are bound at the top level (in the context of the trigger) are referred to as 'pipeline parameters,' while parameter sets that are bound at the node level are referred to as 'module parameters.' Pipeline parameters are visible to all modules and unit of work generators in the pipeline, whereas module parameters are visible only to the module(s) to which they are bound. Pipeline parameters are useful in cases where the same parameter set types are needed by several modules and the operator wants to use the same parameter set instance for all of the modules. A given parameter set type may not be defined as both a pipeline parameter and a module parameter in the same trigger.

3.5 Generic Persistence of Application Objects

Application objects such as `Parameters` and `UnitOfWorkDescriptors` must be stored in the database as part of the pipeline configuration (in the case of `Parameters`) and job meta-data (`UnitOfWorkDescriptor`). To avoid the need to customize the database schema for each application, these objects are persisted by the framework in a generic way.

The contents of each `Parameters` class and `UnitOfWorkDescriptor` are persisted to the database via the `BeanWrapper` class. `BeanWrapper` is a template class that can convert any class that conforms to the JavaBeans specification to and from a collection of name/value (`String`/`String`) pairs.

Each `Parameters` class conforms to the JavaBeans specification[15], which allows the framework to provide a generic GUI editor and to persist them in a generic fashion. Each `Parameters` class contains a default constructor, a field for each parameter, and a getter and setter for each field (per the JavaBeans specification). All primitive Java types, plus `String` (including arrays of these types), are supported.

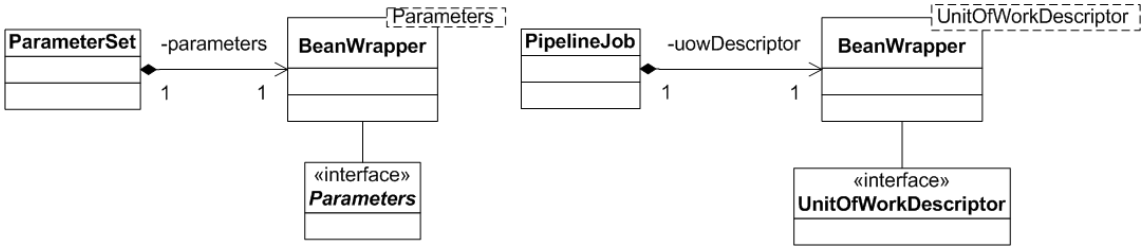

Figure 4: Generic Persistence with BeanWrapper

`BeanWrapper` utilizes the Apache Commons[12] BeanUtils package to convert from name/value pairs to the typed fields of the wrapped class (in this case, an instance of `Parameters` or `UnitOfWorkDescriptor`) and vice-versa. The `BeanWrapper` class is also a Hibernate class and contains the following fields, which are persisted to the database.

- **className** : `String` – The fully-qualified Java class name of the wrapped class
- **properties**: `Map<String,String>` – The names and values of the fields in the wrapped class in `String` form.

Note that the wrapped class itself is not persisted except via the properties field indicated above. While parameter values are persisted as `String` regardless of the type, the type information is retained in the wrapped class. This type information is used to validate parameter values entered by the operator via the graphical console or via an XML import.

3.6 Locking and Versioning of Configuration Entities

For data accountability purposes, the state of the pipeline configuration data model must be preserved when a new pipeline instance is launched. Typically, launching new pipeline instances occurs more frequently than changing the configuration, so it would be inefficient to copy the configuration each time a pipeline instance is launched. Additionally, configuration changes typically only involve a small portion of the configuration such as changing a handful of parameters so it is equally inefficient to make a complete copy of the configuration each time the operator makes a change. For these reasons, a copy-on-write versioning and locking mechanism is implemented for each main entity type. `PipelineDefinition`, `PipelineModuleDefinition`, and `ParameterSet` are independently

versionable. `PipelineDefinition`s are versioned as a unit. The nodes that make up the pipeline definition are not independently versionable.

Each of these entities contains a 'version' field and a 'locked' field. When a new pipeline instance is launched, the latest versions of all entities referred to by the trigger used to launch the pipeline are marked as 'locked' (if not already locked), and references are made between the pipeline instance and the specific versions of the entities that were used. When the operator attempts to modify a locked entity, a copy of the entity is created with an incremented version number, and the framework sets the 'locked' flag on the new version to false. The modifications are then made to the new version of the entity. This approach guarantees that once a particular version of an entity is bound to a pipeline instance, it will never change, thereby solidifying the data accountability record for that instance. These versioning and locking operations happen automatically and are transparent to the operator.

4. PIPELINE EXECUTION

Pipeline execution consists of executing the individual modules that make up the pipeline definition, starting with the first module and ending with the leaf modules. The start and end modules can also be overridden for a given pipeline instance, allowing execution of small segments of a pipeline. Execution of a module consists of executing all jobs associated with the module. There is one job per unit of work, so the number of jobs is determined by the unit of work generator configured for the node.

Jobs are processed by workers that run on the cluster nodes. There is one worker process per cluster node, but each worker process can be configured to run any number of threads, typically one thread per available CPU core. Jobs are distributed to the workers via a Java Message Service (JMS) queue, using ActiveMQ[15] as the JMS provider. Jobs are not pre-assigned to specific workers; rather the jobs are placed on the queue where worker threads can claim them when they are ready to accept a new job. In this way, jobs are dynamically load-balanced across the cluster. Workers can be dynamically added or removed from the cluster without configuration changes. Newly added workers will simply start processing pending jobs.

There is no centralized controller for pipeline execution. Instead, the pipeline transition logic is performed by the workers in a distributed fashion. As each job completes on a particular worker machine, the worker is responsible for executing the transition logic that generates the jobs for the next module. Module transitions can be synchronized (i.e., all tasks for a given node must complete before the next node starts) or unsynchronized (i.e., tasks for a given unit of work can only start for the next node when that unit of work for the current node completes). The framework component responsible for pipeline launching and the transition logic is known as the pipeline executor.

4.1 Pipeline Launching

Pipeline launching consists of creating the `PipelineInstance` and `PipelineInstanceNode` objects, binding the latest versions of all parameter sets referenced by the trigger to these objects, locking the configuration entities for the data accountability record, and launching the jobs for the first module in the pipeline. These steps are described in more detail below.

Lock pipeline definition. The first step in launching a new instance is to lock the associated pipeline definition to prevent subsequent modifications to the version of the definition used by the new instance. This preserves the state of the pipeline definition for purposes of maintaining the data accountability record for the new instance. If the operator makes subsequent changes to the definition, those changes will be made to a new, unlocked version of the definition.

Bind parameters. Triggers are mapped to parameter sets by name only—to a specific version of the parameter sets. This is referred to as a 'soft reference'. When a trigger is fired, a new pipeline instance is created and the pipeline instance is bound to the latest version of each parameter set. This is known as a 'hard reference'. A hard reference between a pipeline instance and specific versions of parameter sets serves two purposes: First, it guarantees that all tasks that make up the pipeline instance will have a consistent view of the parameters, even if the operator changes the parameters while the pipeline is running. Secondly, it provides a data accountability record of the parameter values that were used for a given pipeline run. When parameter sets are bound to the new pipeline instance, the pipeline parameters are bound to the `PipelineInstance` object and module parameters are bound to the appropriate `PipelineInstanceNode` object.

Create instance nodes. Once the parameters at the pipeline level are bound, the instance nodes are created. There is one instance node per module in the pipeline definition. For each instance node, the associated module definition is locked and the parameters defined at the module level in the trigger are bound to the instance node.

Launch first module. Finally, the first module of the pipeline is launched. The logic of launching a module is the same for the first module launched by a trigger as it is for later modules in the pipeline that are launched by a worker as part of the transition logic between modules.

4.2 Module Launching and the Unit of Work

Modules are launched by triggers when initiating a new pipeline instance or by worker machines when executing the transition logic for synchronized transitions between modules. **Figure 5** illustrates this process.

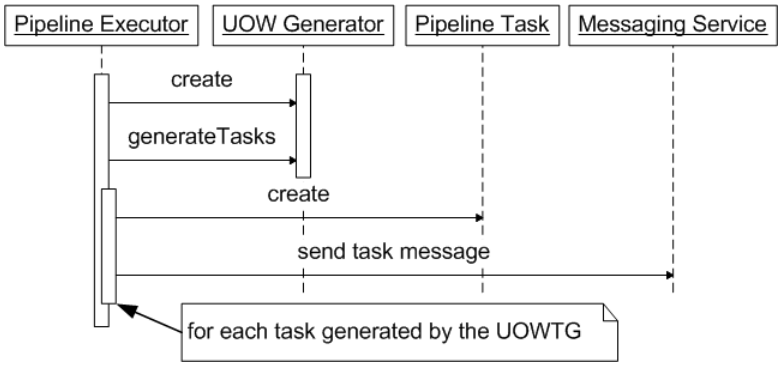

Figure 5: Launching a New Pipeline Instance Node

The decision of how to break up the work for a module is delegated to the unit of work generator. The unit of work generator is a Java class that implements the `UnitOfWorkGenerator` interface. The operator sets the fully-qualified name of this class in the configuration for the pipeline definition. This approach allows the creation and configuration of arbitrary unit of work generators without code changes to the framework.

When the pipeline executor launches a module, it first instantiates a new instance of the unit of work generator using the name of the Java class defined in the pipeline definition for the module. Next, the executor combines the parameters that are bound to the pipeline instance with the parameters that are bound to the instance node, and passes the combined set to the `generate` method of `UnitOfWorkGenerator`. This allows the parameters to control the behavior of the task generator. Passing the combined set allows the operator to set task generator parameters at the pipeline level or the module level.

The generator returns a list of objects that implement the `UnitOfWorkDescriptor` interface. This interface contains a single method, `briefState`, which returns a String containing a brief description of the unit of work for display in the console. The actual fields that describe the unit of work are contained only in the implementing class and are not used by the framework. They are simply stored and passed to the pipeline modules where the pipeline modules can use them to determine the dimensions of the data set to be retrieved from the data store.

The pipeline executor creates a `PipelineJob` object for each `UnitOfWorkDescriptor` object returned by the task generator. The pipeline executor attaches the `UnitOfWorkDescriptor` object to the `PipelineJob` and persists it using the `BeanWrapper` mechanism described above.

Finally, the pipeline executor creates a `WorkerTaskRequest` object that contains the pipeline job identifier, and this object is enqueued for the worker machines using JMS.

For unsynchronized transitions between modules, the pipeline executor will copy the unit of work descriptor from the previous module instead of calling the unit of work generator to generate new descriptors.

4.3 Use of JMS for Distributing Jobs

Pipeline jobs are load-balanced across the pool of available workers using JMS. One key aspect of JMS is that it decouples senders from receivers. Instead of sending messages to a specific receiver, messages are sent to a named topic or queue. The JMS broker then forwards those messages to receivers that have registered an interest in those

topics/queues. JMS supports two messaging paradigms; queues (point-to-point messaging) and topics (broadcast, or publish-subscribe messaging). A JMS queue is used to distribute pipeline jobs to workers. When a new pipeline instance is launched, a corresponding queue is created. This JMS queue has a name of the form worker-task-request-N, where N is the pipeline instance identifier. A JMS message announcing the creation of the task queue is then broadcast to all workers using the '`pipeline-events`' topic, which all pipeline workers subscribe to. The announcement tells the workers to start listening for new messages from the new task queue.

Distributing the jobs evenly to the available pool of workers is the responsibility of the JMS broker. The broker distributes the jobs to workers in a round-robin fashion, but jobs are allocated to a worker only when that worker signals that it is ready to start processing a new job. The worker does this by attempting to dequeue a message from the queue. Because jobs are not pre-allocated to workers, workers can be added to or removed from the cluster at any time and new worker machines will immediately start processing pending jobs in the queue.

Priority can be configured at the pipeline instance level, so multiple instances can be running at the same time, but jobs from higher priority instances will get processing priority.

4.4 Job Execution

To execute a job, control must be passed to the custom code implemented for the pipeline module, and certain meta-data managed by the framework must be made available to that code. These meta-data include the unit of work descriptor associated with the job and the parameters configured in the trigger that launched the instance.

Instantiating and Invoking the `PipelineModule` object

Job processing is performed by creating a new instance of the implementing class for the pipeline module (specified by the `implementingClass` field in the `PipelineModuleDefinition` database object). This class must extend the `PipelineModule` abstract class. This class defines an abstract method called `processJob`, which is used to invoke the module. This method is defined as follows:

```
public abstract void processJob(PipelineInstance pipelineInstance, PipelineJob
pipelineJob) throws PipelineException;
```

Access to the unit of work descriptor and the parameters is provided by the `PipelineJob` object via the following methods. If the `processJob` throws an exception, the framework considers the job to have failed, and updates the `PipelineJob` meta-data in the database accordingly. The framework does not execute the transition logic (see below) in this case.

```
public <T extends Parameters> T getParameters(Class<T> parametersClass);
```

This method allows a pipeline module to retrieve the instance of the parameters for the specified parameter class. This method will retrieve the parameters regardless of whether they were bound at the module level or at the pipeline level in the trigger so that the application code does not need to be aware of the distinction.

```
public BeanWrapper<UnitOfWorkDescriptor> getUowDescriptor();
```

This method provides access to the unit of work descriptor for the job.

4.5 Pipeline Transition Logic

When the operator launches a new pipeline instance, the pipeline executor only creates `PipelineJob` objects for the first module. Worker machines create jobs for subsequent modules as part of the transition logic execution that takes place immediately following successful job processing in the worker process.

Transitions between pipeline modules can be synchronous or asynchronous. The operator specifies the type of transition in the configuration of the pipeline definition. For synchronized transitions, all jobs must be complete for the current module before jobs for the next module can be created. For unsynchronized transitions, as each job completes for the current module, a new job will be created for the next module using the same unit of work descriptor as the current job. This type of transition requires that both modules use the same unit of work type (i.e., they must be configured to use the same `UnitOfWorkGenerator`). Transitions between modules of different types must be synchronized.

Figure 6 shows an example of both types of transition. The *Kepler* photometric analysis (PA) and pre-search data conditioning (PDC) modules both use the same unit of work generator and therefore support asynchronous transition. As each PA job completes, a PDC job with the same unit of work descriptor will be started even if there are pending PA

tasks for other units of work. The transiting planet search (TPS) module, however, uses a different unit of work type than PDC, so the transition must be synchronous. All PDC tasks must be complete before any TPS tasks start.

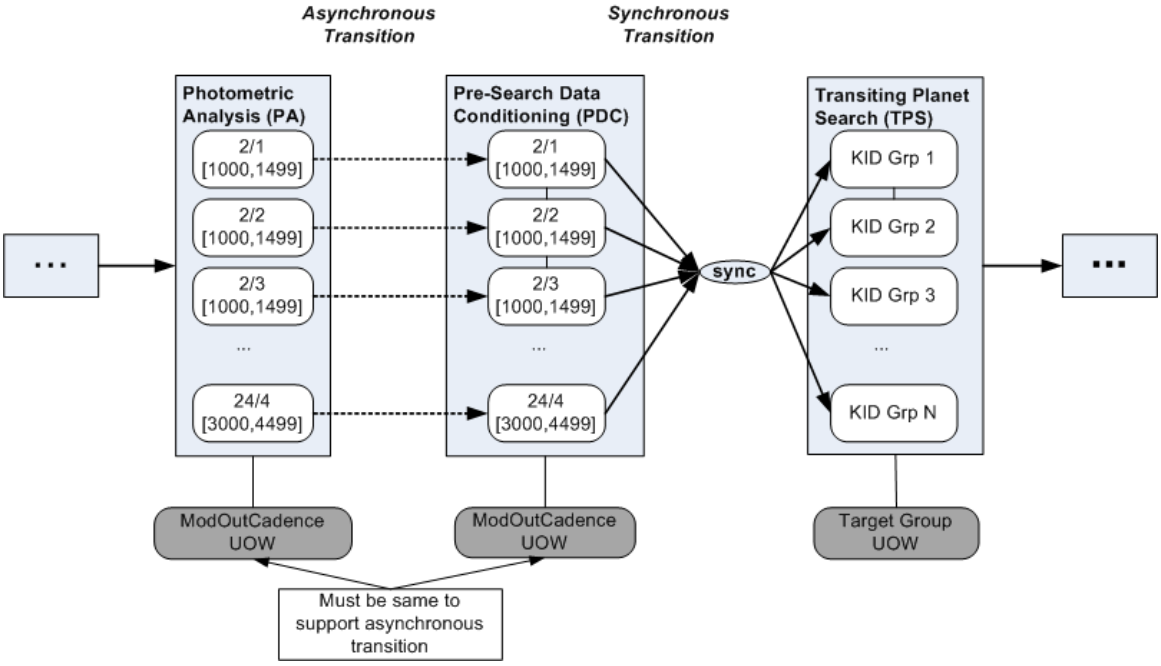

Figure 6: Synchronous vs. Asynchronous Transitions

The pipeline instance node keeps track of the number of jobs associated with the module (grouped by job state). The first step in the transition logic is to update these metrics. Since workers running on different nodes may attempt to update these metrics at the same time, a locking mechanism is needed to serialize the access to the critical block of code that retrieves the old values of the metrics, updates them, and stores the updated values back to the database. Java synchronization is insufficient since we need to synchronize multiple workers running in different Java virtual machines on different hosts. Instead, we lock the row in the database that contains the metrics using SQL '`SELECT FOR UPDATE`' syntax. This approach locks the row until the transaction is committed and blocks other workers from reading the row until the lock is released, ensuring that the *read-increment-update* sequence is performed atomically. This lock is held for a very short period of time, so the performance impact is negligible.

4.6 Failure Recovery

Since pipeline job failures are an inevitable part of pipeline operations, the framework provides a mechanism that allows the operator to restart failed jobs after the problem that caused the failure is corrected. The operator can rerun (via the graphical console) all jobs in the `ERROR` state for a particular pipeline instance. Similarly, the operator can re-run individual failed jobs.

When a failed job re-runs, the pipeline executor resets the state to `SUBMITTED`, and updates the job state counts (decrement failed count, increment submitted count). Then, the pipeline executor places a new `WorkerTaskRequest` object in the JMS queue for the pipeline instance using the messaging service.

There are certain scenarios where a job is interrupted, but the worker doesn't get a chance to update the job state to `ERROR`. These include power failures, hardware failures, or software crashes. To allow recovery from these states, workers perform a check when they initialize. This check consists of changing the state to `ERROR` for all jobs in the `PROCESSING` state that are currently assigned to the worker doing the check. Since the worker is still initializing, these jobs cannot truly be `PROCESSING` and must have been previously interrupted. Putting the jobs in the `ERROR` state allows the operator to re-run them if desired. The operator can also manually force a job into the `ERROR` state (for example, if the worker machine goes permanently offline).

In order to preserve the integrity of the data accountability record, changes made to the pipeline configuration, parameter library, or model metadata library after the instance was launched will not be seen by the jobs of the instance, even if the

changes were made before re-running the failed jobs. If fixing the problem that caused the error involves changing one of these elements, a new pipeline instance will need to be launched.

5. GRAPHICAL CONSOLE

The graphical console is a Java/Swing application that allows the operator to view and edit various elements of the pipeline configuration, launch new pipelines, and monitor running pipelines. Pipeline configuration, including parameters, can be done entirely using the GUI (figure 7), or via an XML import using either the GUI or a command-line tool.

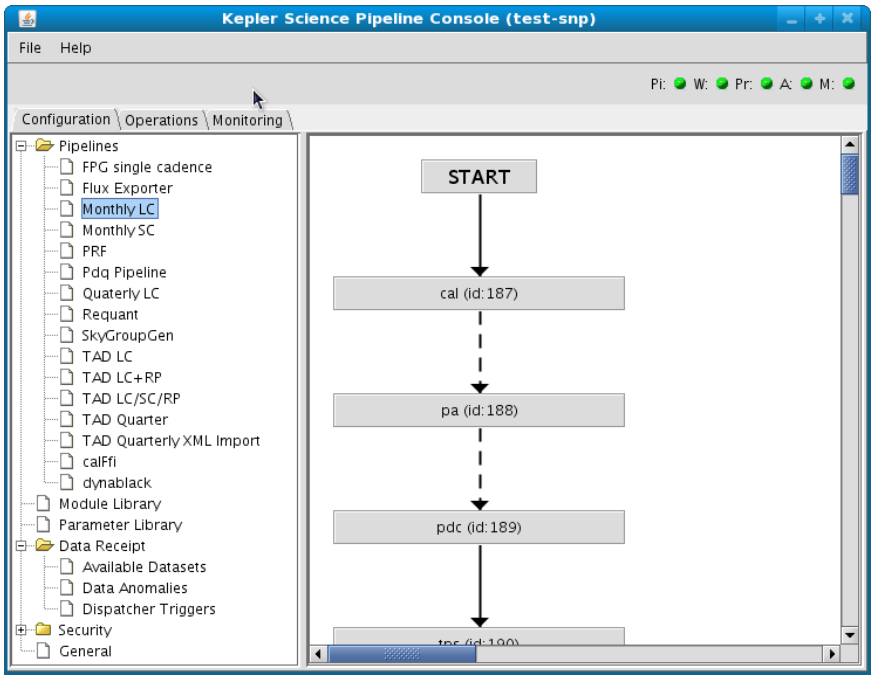

Figure 7: Graphical Console: Pipeline Configuration

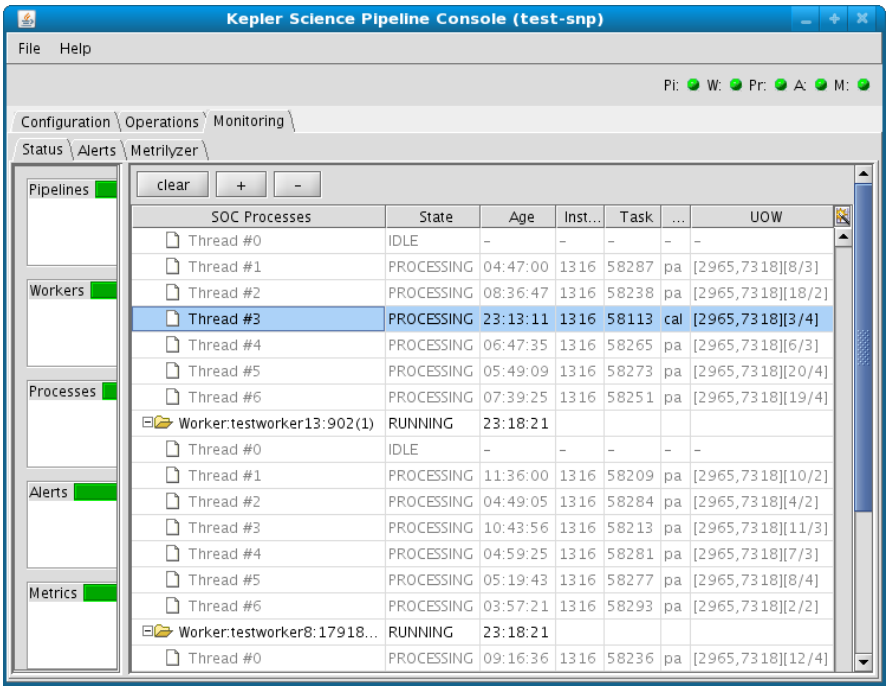

Figure 8: Graphical Console: Cluster Status Monitor

ACKNOWLEDGMENTS

Funding for this work has been provided by the NASA Science Mission Directorate (SMD).

REFERENCES

[1] Middour, C., et al., “Kepler Science Operations Center architecture,” Proc. SPIE, this volume (2010).
[2] Hall, J., et al., “Kepler Science Operations Processes, Procedures, and Tools,” Proc. SPIE, this volume (2010).
[3] Koch, D.G., et al., “Kepler Mission design, realized photometric performance, and early science”, Astrophysical Journal Letters, 713, L79-L86 (2010).
[4] Quintana, E.V., et al., “Pixel-level calibration in the Kepler Science Operations Center pipeline,” Proc. SPIE, this volume (2010).
[5] Twicken, J.D., et al., “Photometric analysis in the Kepler Science Operations Center pipeline,” Proc. SPIE, this volume (2010).
[6] Twicken, J.D., et al., “Presearch data conditioning in the Kepler Science Operations Center pipeline,” Proc. SPIE, this volume (2010).
[7] Jenkins, J.M., et al., “Transiting planet search in the Kepler pipeline,” Proc. SPIE, this volume (2010).
[8] Wu, H., et al., “Data validation in the Kepler Science Operations Center pipeline,” Proc. SPIE, this volume (2010).
[9] Klaus, T.C., et al., “Kepler Science Operations Center pipeline framework extensions,” Proc. SPIE, this volume (2010).
[10] Sun Developer Network, “Java Message Service Specification,” http://java.sun.com/products/jms/
[11] Hibernate, “Hibernate,” http://www.hibernate.org/
[12] Apache, “Apache Commons,” http://commons.apache.org/
[13] JBoss, “JBoss Transactions," http://www.jboss.org/jbosstm
[14] HSQLDB, “HyperSQL Database,” http://hsqldb.org/
[15] Apache, “ActiveMQ,” http://activemq.apache.org/