LARGE SYNOPTIC SURVEY TELESCOPE

Large Synoptic Survey Telescope (LSST)

Data Management Middleware Design

K.-T. Lim, G. Dubois-Felsmann, M. Johnson, M. Juric, and D. Petravick

LDM-152

Latest Revision: 2019-07-10

Abstract

The LSST middleware is designed to isolate scientific application pipelines and payloads, including the Alert Production, Data Release Production, Calibration Products Productions, and science user pipelines executed within the LSST Science Platform, from details of the underlying hardware and system software. It enables flexible reuse of the same code in multiple environments ranging from offline laptops to shared-memory multiprocessors to grid-accessed clusters, with a common I/O and logging model. It ensures that key scientific and deployment parameters controlling execution can be easily modified without changing code but also with full provenance to understand what environment and parameters were used to produce any dataset. It provides flexible, high-performance, low-overhead persistence and retrieval of datasets with data repositories and formats selected by external parameters rather than hard-coding.

Change Record

Version	Date	Description	Owner name
1.0	2011-07-25	Initial version based on pre-existing UML models and presentations	Kian-Tat Lim
2.0	2013-05-22	Updated based on experience from prototypes and Data Challenges.	Kian-Tat Lim
8	2013-10-04	Updated based on comments from Process Control Review, changed to current terminology	Kian-Tat Lim
9	2013-10-09	Further updates based on Process Control Review, formatting cleanup.	Kian-Tat Lim
10	2013-10-10	TCT	R Allsman
11.0	2017-07-05	Rewritten for Construction and Operations. Approved in RFC-358.	K-T Lim

Document curator: Kian-Tat Lim

Document source location: `https://github.com/lsst/LDM-152`

Contents

Data Management Middleware Design

1 Introduction

This document describes the baseline design of the LSST data access and processing middleware, including the following components:

- Data Butler Access Client
- Task Framework

The Data Butler Access Client provides a flexible interface for retrieving and persisting the LSST data products. The Task Framework defines how scientific algorithms are packaged into pipelines, including how they are configured, how they use the Data Butler to access data, and how they execute on multiple nodes or cores.

Common to all aspects of the middleware design is an emphasis on flexibility through the use of abstract, pluggable interfaces controlled by managed, user-modifiable parameters. In addition, the substantial computational and bandwidth requirements of the LSST Data Management System (DMS) force the designs to be conscious of performance, scalability, and fault tolerance.

Requirements for the DMS Middleware are defined in LDM-556.

Figure 1 illustrates how various parts of the middleware interact with each other.

2 Data Butler Access Client

This component is the framework by which applications retrieve datasets from and persist datasets to file and database storage. It provides a flexible way of identifying datasets, a pluggable mechanism for discovering and locating them, and a separate pluggable mechanism for reading and writing them.

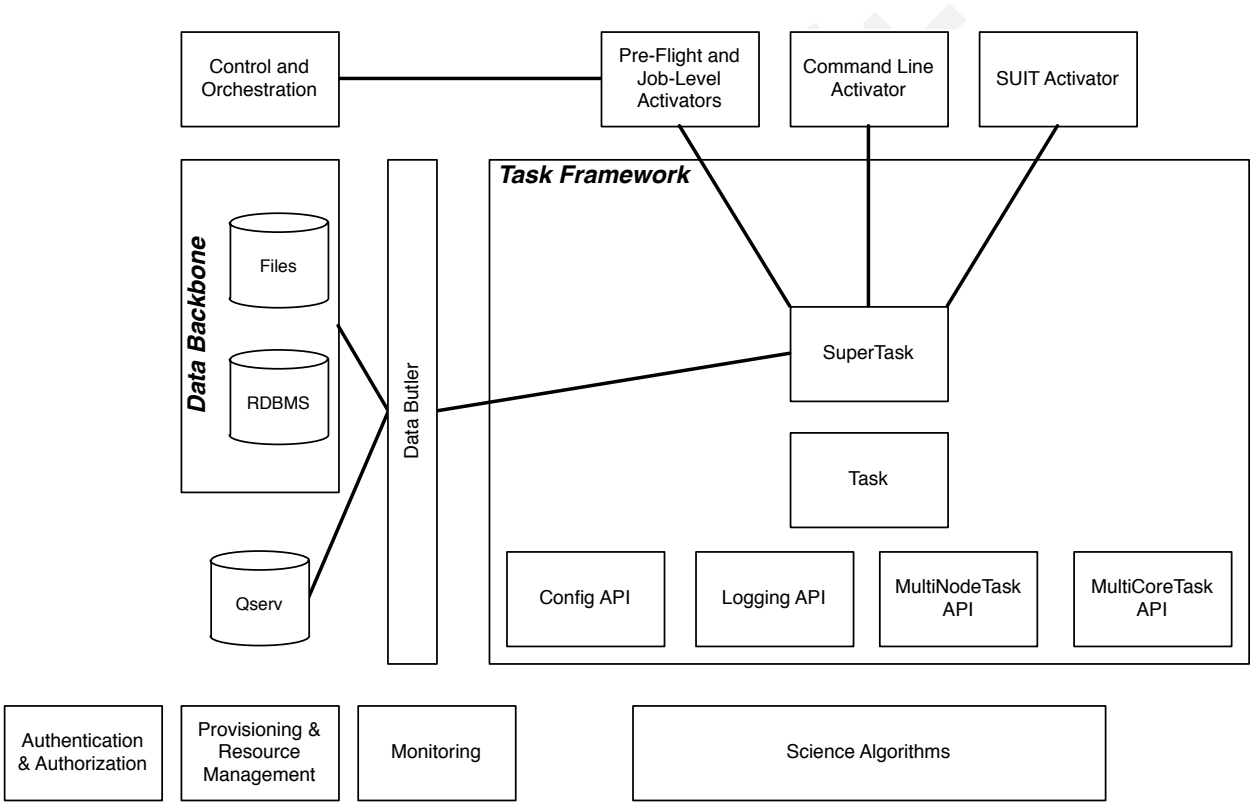

FIGURE 1: Data Management Middleware and Infrastructure

2.1 Key Requirements

The framework must provide persistence and retrieval capabilities to application code. Persistence is the mechanism by which application objects are written to files in some format or a database or a combination of both; retrieval is the mechanism by which data in files or a database or a combination of both is made available to application code in the form of an application object. Persistence and retrieval must be low-overhead, allowing efficient use of available bandwidth. The interface to the I/O layer must be usable by application developers. It is required to be flexible, allowing changes in file formats or even whether a given object is stored in a file or the database to be selected at runtime in a controlled manner. It must be possible to store image pixel data in a file while part or all of its metadata is stored in a different file or in a database table.

2.2 Baseline Design

The framework is designed to provide access to datasets. A dataset is a logical grouping of data that is persisted or retrieved as a unit, typically corresponding to a single programming object or a collection of objects. Datasets are identified by a set of key/value pairs along with a label for the type of data (e.g. processed visit image). Datasets may be persisted into multiple formats.

The framework is made up of two main components: a “Mapper” that manages camera-specific repositories of datasets and determines the logical location of an identified dataset and a “Butler” that performs persistence and retrieval for that dataset. In the baseline design, the Butler wraps the Mapper and provides the exposed interface; it is anticipated that future evolution will increasingly separate these two. Both components are implemented in Python.

The Butler and its included Mapper manage repositories of datasets which can be in files or in a database. Operations on datasets include get, put, list, and check for existence.

The Butler contains a pluggable set of serializers that handle persistence to and retrieval from serialization formats such as Python pickle files, task configuration override files (Python scripts), and FITS tables; separate plugins handle different types of storage such as filesystems, object stores, or SQL databases.

The Butler is initialized with zero or more read-only input repositories and one or more read-/write output repositories. When reading a dataset, the output repository is searched first; the "chained" input repositories are searched if the dataset is not found. When writing a dataset, the dataset always goes to the output repository, never to the chained inputs (unless the output is specified as being the same as an input). The set of input repositories is recorded for provenance purposes and for future uses of the output repository.

The Mapper translates from a dataset type name and one or more astronomically meaningful key/value dictionaries into a dataset location and storage. The location might be a pathname or URL for a file; it would include an SQL query for a database.

The Mapper provides flexibility at many levels. First, it allows the provided key/value dictionaries to be expanded using rules or database lookups. This can be used to map from a visit identifier to an exposure length, for example, or from a CCD name to an equivalent number. This facility is used to implement the "rendezvous" of raw data with its corresponding calibration data. Second, it allows the key/value pairs to be turned into a location string using a dataset type-dependent method. Typically, this will be performed by substitution into a dataset type-specific template. Third, the Mapper allows camera-specific and repository-specific overrides and extensions to the list of rules and templates, enabling per-camera and dynamic dataset type creation.

For LSST, the Mapper flexibility is used in several ways. For precursor data, image files can retain the names they were assigned in the upstream archive, with templates being used to compute the filename from the values in the dictionary. Metadata stored in a SQLite database within the repository allows more rapid listing of available datasets and expansion of partial key/value dictionaries. Calibration data is associated with images by observation timestamp using validity ranges stored in an auxiliary SQLite database. For LSST Data Release Production, the same mechanisms will be used to process data staged from the Data Backbone (DBB). Direct access by the Data Butler to the Data Backbone (e.g. from the LSST Science Platform) will use DBB metadata tables directly. The Mapper will produce logical file identifiers that the DBB converts to endpoint-local physical locations. For LSST Alert Production, the Mapper will be configured to point to raft images on the distributor nodes and locally-cached calibration and template images.

2.3 Alternatives Considered

Use of a full-fledged object-relational mapping system for output to a database was considered but determined to be too heavyweight and intrusive. Persistence from C++ was tried and found to be complex and unnecessary; Python persistence suffices since all control is in Python.

2.4 Implementation

A Python implementation of the design has been in place since DC3 prior to Final Design Review. This implementation has evolved substantially since then to simplify the pluggability of serialization and storage, to simplify the configuration of common Mapper subclasses, to allow and maintain more repository configuration information within the repository, to support multiple input and output repositories, to provide a configurable (rather than hard-coded) mechanism for retrieving and persisting composite datasets that are made up of more than one serialized dataset, to replace a custom configuration file format with standard YAML [4], and to provide support for caching persisted and retrieved objects in memory.

Since low-level serialization code is implemented in C++ or external libraries, I/O performance remains good.

3 Task Framework

The Task Framework enables the packaging of scientific algorithms into executable and reusable pipelines. It handles configuration, argument parsing, and interfacing with the I/O and inter-process communications mechanisms.

The Task Framework is a Python class library that provides a structure of standardized class entry points and conventions to organize low-level algorithms into potentially-reusable algorithmic components called Tasks. Sample Tasks might include dark frame subtraction, object detection, or object measurement. The Framework organizes tasks into basic pipelines called PipelineTasks. Sample PipelineTasks might include processing a single visit, building a coadd, or differencing a visit. The algorithmic code is written into (Pipeline)Tasks by overriding classes and providing implementation for standard entry points. The Task Framework

allows the pipelines to be constructed and run at the level of a single node or a group of tightly-synchronized nodes. It allows for sub-node parallelization: trivial parallelization of Task execution, as well as providing parallelization primitives for development of multi-core Tasks and synchronized multi-node Tasks.

The Task Framework serves as an interface layer between orchestration and the algorithmic code. It exposes a standard interface to Activators (command-line runners as well as the workflow component and automated QC systems), which use it to execute the code wrapped in Tasks. The Task Framework does not concern itself with fault-tolerant massively parallel execution of the pipelines over multiple (thousands) of nodes nor any staging of data that might be required; this is the concern of the orchestration and workflow middleware.

The Task Framework exposes to the workflow system the needs and capabilities of the underlying algorithmic code (i.e., the number of cores needed, expected memory-per-core, expected need for data). It may also receive from the orchestration layer information on how to optimally run the particular task (i.e., which level of intra-node parallelization is desired).

3.1 PipelineTask

A PipelineTask represents a unit of (generally transformational) work to be performed on data. Its primary responsibility is to provide the interface between Activators and Tasks. In doing so, it separates input and output from computation, making Tasks more reusable and enabling data movement and other optimizations within a distributed execution environment. The PipelineTask also exposes the kinds of data that it accepts and generates. For example, a coaddition PipelineTask might operate on a set of processed visit images and produce a patch of a coadded image. The specific data items to be processed are supplied through the Activator-PipelineTask interface. The goal of the design is that any PipelineTask can be run in any computational environment, from a laptop command line to the large-scale Data Release Production.

In general a PipelineTask receives the content of its inputs and produces its outputs by invoking the Data Butler.

The PipelineTask base class is a subclass of Task. This is so that PipelineTask can take advantage of the configuration mechanism for Tasks. The hierarchy of Tasks in a specific application therefore extends all the way up to the top-level PipelineTask, and each level is addressable

for configuration discovery and overrides.

Each PipelineTask implements a method that groups the input datasets into "quanta" that are the minimal units of work for an instance of the PipelineTask and notifies the Activator of the outputs to be produced from each such unit of work. It also implements a method to execute a computation on a single quantum of data, typically by retrieving the inputs from the Data Butler and executing the underlying Task, followed by persisting the outputs, again using the Data Butler.

Datasets, as with the Data Butler, are specified by a set of key/value pairs, typically obtained by performing a database query on metadata tables, along with a label for the type of data (e.g. processed visit image).

PipelineTasks also expose their processing requirements to their Activators, such as a need for multi-node communication or multi-core execution.

PipelineTask implementation is in the prototype stage. The previous design and implementation combined the Activator and PipelineTask functionality into a single class (`CmdLineTask`) that is now being replaced.

3.2 Activators

The Activator is responsible for providing a Butler instance for the PipelineTask's use. It is also responsible for instantiating the PipelineTask to be run and for providing necessary inputs to the configuration parameter mechanism (see section 3.4) for the PipelineTask. For example, the "command line Activator" identifies the PipelineTask to be run by name, locates and instantiates it, and provides for command-line overrides of config parameters of the PipelineTask. It also creates a Butler based on one or more provided or defaulted data repositories.

An Activator is responsible for arranging for the execution of a PipelineTask's execution method one or more times over a set of dataset specifiers. Via collaboration with the PipelineTask interfaces, the Activator is able to determine the parallelization and scatter-gather behavior that is permissible and/or required to implement the workflow defined by the PipelineTask.

The Activator therefore controls the input/output data access environment as well as the computational environment of the PipelineTask. It is the plugin that enables PipelineTask

portability and reuse.

Specific Activators that are part of the design include a command line Activator and a workflow Activator that can be used to determine the data needed and produced by a PipelineTask before its execution in order to configure data staging capabilities.

Activator implementation is in the prototype stage.

3.3 Task

Tasks are simply Python scripts with a common base class. Using Python enables Tasks to support complex control flows without developing a new control flow language. Tasks may hierarchically call sub-Tasks as part of their execution. Errors are reported through standard Python exception subclasses.

Tasks provide by default three facilities commonly used by all algorithmic code: configuration, metadata, and logging. The Task base class provides configuration facilities using the configuration framework. The Task configuration can include selection of sub-Tasks to be executed, allowing the pipeline to be reconfigured at runtime. The Task class allows Tasks to save metadata related to their processing, such as performance or data quality information, separate from their data products. Each Task provides a default logger instance associated with the Task's name and position within the Task/sub-Task hierarchy.

The basic Task implementation is complete and resides in the `pipe_base` package.

3.4 Configuration

The configuration component of the Pipeline Framework is a mechanism to specify parameters for applications and middleware in a consistent, managed way. The use of this component facilitates runtime reconfiguration of the entire system while still ensuring consistency and the maintenance of traceable provenance.

3.4.1 Key Requirements

Configurations must be able to contain parameters of various types, including at least strings, booleans, integers, and floating-point numbers. Ordered lists of each of these must also be supported. Each parameter must have a name. A hierarchical organization of names is required so that all parameters associated with a given component may be named and accessed as a group.

There must be a facility to specify legal and required parameters and their types and to use this information to ensure that invalid parameters are detected before code attempts to use them. Default values for parameters must be able to be specified; it must also be possible to override those default values, potentially multiple times (with the last override controlling).

Configurations and their parameters must be stored in a user-modifiable form. It is preferable for this form to be textual so that it is human-readable and modifiable using an ordinary text editor.

It must be possible to save sufficient information about a configuration to obtain the value of any of its parameters as seen by the application code.

3.4.2 Baseline Design

The initial design based on a custom text file format has been refined based on experimentation during the design and development phase.

Configurations are instances of a Python class. The class definition specifies the legal parameter names, their types, default values if any, minimum and maximum lengths for list values, and whether a parameter is required. It also mandates that a documentation string be provided for each parameter. Use of Python for defining configurations enables inheritance, the use of package imports to easily refer to configurations from other components, complex parameter validation, and the ability to define powerful new parameter types. Default values in configuration instances can be overridden by human-readable text files containing normal Python code, simplifying the specification of multiple similar parameters. Overrides can also be set using command line parameters. The Python base class maintains complete history information for every parameter, including its default and all overrides. The state of a configuration as used by the application code can be written out and optionally ingested into a

database for provenance purposes .A mechanism is provided to automatically translate between the Python configuration instance and a control object for C++ code.

3.4.3 Implementation

An implementation of the Python-based design in the `pex_config` package has been used since December 2011. It contains features such as selection of an algorithm by name from a registry, automatically pulling in the algorithm's configuration. Tools are provided to print out the history of any parameter.

3.5 Logging

The logging service is used by application and middleware code to record status, diagnostic, and debugging information about their execution.

3.5.1 Key Requirements

Log messages must be associated with component names organized hierarchically. Logging levels controlling which messages are produced must be configurable on a per-component level. There must be a way for messages that are not produced to not add significant overhead. Logs must be able to be written to local disk files to be gathered by a collection system. Metadata about a component's context, such as a description of the CCD being processed, must be able to be attached to a log message.

3.5.2 Baseline Design

Log objects are created in a parent/child hierarchy and associated with dotted-path names; each such Log and name has an importance threshold associated with it. Methods on the Log object are used to record log messages. One such method uses the C++ varargs functionality to avoid the overhead of formatting the message until it has been determined if the importance meets the threshold. Log messages can have additional key/value contextual metadata associated with them through a per-thread diagnostic context.

3.5.3 Implementation

The logging implementation in the `log` package is based on the Apache `log4cxx` package [5]. Use of an off-the-shelf package provides the required functionality, including relatively advanced features as thread-safety, with minimal support cost.

Wrappers were written to adapt the `log4cxx` classes to LSST needs, providing "syntactic sugar" as well as providing an interface that can be reimplemented in case `log4cxx` becomes deprecated in the future. The C++ classes and methods were also wrapped with `pybind11` to enable compatible access from Python.

`log4cxx` provides for significant configurability of logs, including their destinations, message formatting, and thresholds, both through code and through external configuration files in either XML or Java property format.

3.6 MultiNode API

The MultiNode API is used to isolate the applications code from the details of the underlying communications mechanism used to coordinate execution on and transfer data among a tightly-synchronized set of nodes.

3.6.1 Key Requirements

The MultiNode API must support at least point-to-point communication, global collection and aggregation of data from a parallel computation with distribution of the aggregate back to parallel processes, and data exchange from processes to “neighboring” processes using a defined geometry. It must be possible to send and receive objects, but transmission of complex data structures involving pointers is not required.

3.6.2 Baseline Design

The MultiNode API will be an abstract interface used by applications code implemented using two technologies: a message broker such as RabbitMQ [3] and MPI [1]. The former will typically be selected for general-purpose, low-volume communication, particularly when global publish/subscribe functionality is desired; the latter will be used for efficient, high-rate com-

munication. A PipelineTask will call the MultiNode API with a specification of its desired geometry in order to execute its algorithm in parallel. The algorithm will make explicit use of the MultiNode API to send data to and receive data from other instances of the task, including scatter/gather (or map/reduce) communication.

3.6.3 Prototype Implementation

A set of classes have been written that use a batch system and communications via `mpi4py` [6] to provide a pool of nodes that can be used in map/reduce fashion. Tasks making use of this package subclass the base classes provided by this package and call the API explicitly to map Task and function execution across data distributed over the node pool. This API is used by driver scripts that perform single frame processing, calibration frame processing, coaddition, and multiband measurement.

3.7 MultiCore API

The MultiCore API is used to isolate the applications code from the details of the underlying threading mechanism used to coordinate execution on muliple cores on one node.

Use of this API will be necessary to take advantage of current and future processors that contain large numbers of cores but limited amounts of memory per core.

This API has not yet been designed. Given contemporary limitations of threading at the Python level, it is anticipated that it will be implemented only at the C++ level and will likely use an existing API such as OpenMP [2].

4 References

[1] MPI Documents, URL `http://mpi-forum.org/docs/`

[2] OpenMP, URL `http://www.openmp.org/`

[3] RabbitMQ – Messaging that just works, URL `https://www.rabbitmq.com/`

[4] The Official YAML Web Site, URL http://yaml.org/

[5] Apache log4cxx, URL https://logging.apache.org/log4cxx/latest_stable/

[6] MPI for Python, URL http://mpi4py.readthedocs.io/en/stable/

[7] **[LDM-556]**, Dubois-Felsmann, G., 2017, *Data Management Middleware Requirements*, LDM-556, URL https://ls.st/LDM-556