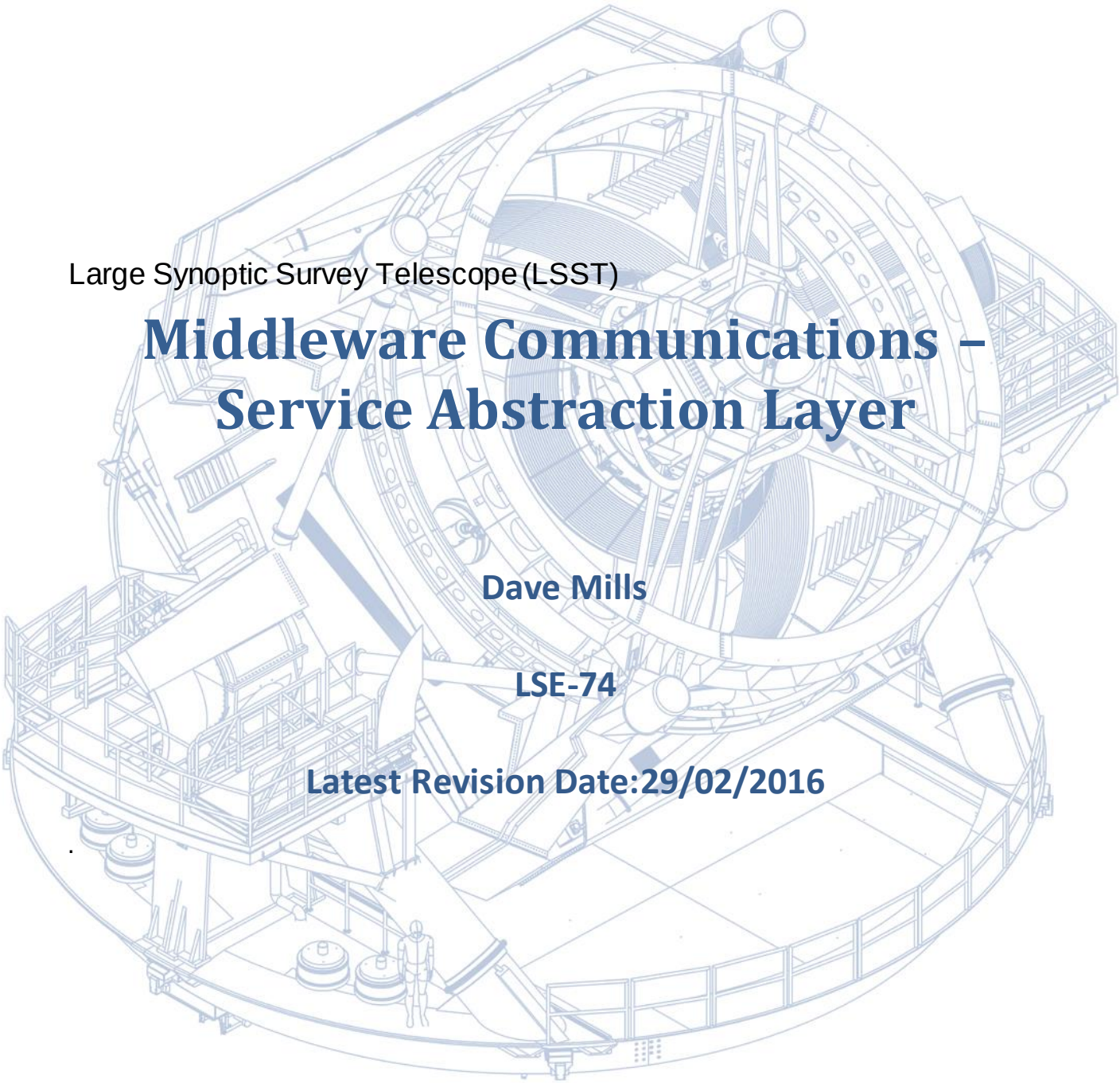
LARGE SYNOPTIC SURVEY TELESCOPE



Large Synoptic Survey Telescope (LSST)

Middleware Communications – Service Abstraction Layer

Dave Mills

LSE-74

Latest Revision Date:29/02/2016

.

Change Record

Version	Date	Description	Owner name
1.0	12/13/2012	Initial Draft	D. Mills
2.0	01/20/2016	Split into separate documents Moved subsystem specific details into non controlled document	D. Mills
2.1	02/29/2016	Incorporated feedback	D. Mills

Contents

Change Record 2
1.0 SCOPE 5
1.1 Document Overview 5
1.2 Glossary 5
1.3 Applicable Documents 6
2.0 CONCEPT OF OPERATIONS 6
2.1 System Overview 6
2.2 SAL (Service abstraction layer) 10
2.4 OMG DDS 15
2.6 OMG RTPS wire protocol 18
2.7 Security and Integrity 18
3.0 Detailed Interface Design 20
3.1 Commanding 20
3.1.1 Interface Processing Time 21
3.1.2 Message Metadata 21
3.2 Telemetry Archiving 21
3.3 Events 21
3.4 Communication Methods 22
3.4.1 Initiation: DDS discovery 22
3.4.2 Flow Control: DDS topics 23
3.5 Security Requirements 24
3.5.1 Message timestamps 24
3.5.2 Software versioning checksums 24
4.0 Qualification methods 24
4.1 System dictionary 24
4.2 Command definition database 25

4.3 Telemetry DataStream Definition database 25
4.4 Code generation 25
4.5 Testing and simulation 26
5.0 XML Schema 26
5.1 Interface Definition Examples 31

1.0 SCOPE

This Document provides an introduction to the design of the communications middleware software. It is intended as supplementary support material for the definitive requirements documents LSE-70 and LSE-209.

1.1 Document Overview

This document describes the middleware (software stack) used for subsystem communications. The publish-subscribe architecture is used as the basis. Open standards (OMG Data Distribution Service) describe the low level protocols and API's. The OpenSplice DDS implementation is being used to provide this framework.

An additional "Service Abstraction Layer" provides application developers with a simplified interface to the DDS facilities. It also facilitates the logging of all subsystem telemetry and command history to the Engineering and Facility Database (EFD).

Tools are used to automatically generate communications code from Telemetry, Command , and Event definitions, which are described using "Interface definition Language".

The IDL is generated from XML files describing the interface on a per-subsystem basis.

A System Dictionary describes the syntax and naming schemes used in the XML, thus establishing system-wide consistency. Details of the XML schema can be found in Section 5.

The detailed descriptions of all the Telemetry Streams and Commands, are listed in per-subsystem appendices.

1.2 Glossary

DDS - Data Distribution Service
EA - Enterprise Architect
EFD - Engineering and Facility Database
ICD – Interface Control Document
IDL - Interface Definition Language
LSST - Large Synoptic Survey Telescope

OCS – Observatory Control System
ODBC - Open Database Connectivity
OMG - Object Management Group
QoS - Quality of Service
RTPS - Real Time Publish Subscribe
SAL - Service Abstraction Layer

SQL - Standard Query Language
XML - eXtensible Markup Language

1.3 Applicable Documents

DataStream Definitions Document - DataStream Prototypes 1.7 (Document-11528)
Definition of subsystems - LSST Project WBS Dictionary (Document-985)
Documentation standards - LSST DM UML Modeling Conventions (Document-469)
Messaging standards - OMG DDS 1.1 (Document-2233)
TCS Software Component Interface (LTS-307)
TCS Communication Protocol Interface (LTS-306)
SAL User Guide (Document-16629)
Middleware Service Abstraction Layer API - (Document-3692)
LSST C++ Programming Style Guidelines (Document-3046)
Vendor documentation - OpenSplice manuals (Collection-2791)

Security policies - http://dev.lsstcorp.org/trac/attachment/wiki/Security/Security Policy documents.zip

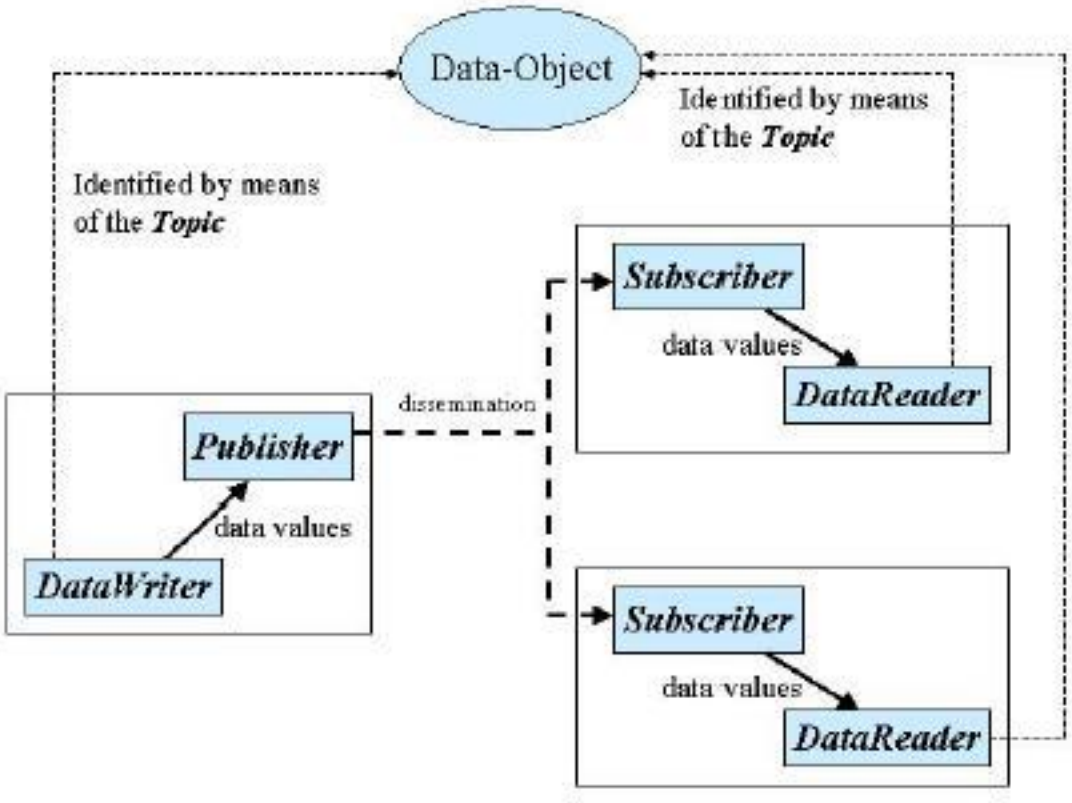
2.0 CONCEPT OF OPERATIONS

2.1 System Overview

The publish-subscribe communications model provides a more efficient model for broad data distribution over a network than point-to-point, client-server, and distributed object models. Rather than each node directly addressing other nodes to exchange data, publish-subscribe provides a communications layer that delivers data transparently from the nodes publishing the data to the nodes subscribing to the data. Publishers send events to the communications layer, which in turn, passes the events to subscribers. In this way, a single event published by a publisher can be sent to multiple subscribers. Events are categorized by topic, and subscribers specify the topics they are interested in. Only events that match a subscriber's topic are sent to that subscriber. The service permits selection of a number of quality-of-service criteria to allow applications to choose the appropriate trade-off between reliability and performance.

The Publish-Subscribe model can also be configured to support the concept of a Command/Action/Response model, in that the transmission of commands is decoupled from the action that executes that command. A command will return immediately; the action begins in a separate thread. When an application receives a command, it validates the attributes associated with that command and immediately accepts or rejects the command. If the command is accepted, the application then initiates an independent internal action to meet the conditions

imposed by the command. Once those conditions have been met, an acknowedgement is posted signifying the successful completion of the action (or the unsuccessful completion if the condition not be met). In this figure, callbacks are implemented using the event features of the publish-subscribe model.



DataReader's and DataWriter's

Information flows with the aid of the following constructs: Publisher and DataWriter on the sending side, Subscriber, and DataReader on the receiving side.

A Publisher is an object responsible for data distribution. It may publish data of different data types. A DataWriter acts as a typed interface to a publisher. The DataWriter is the object the application must use to communicate to a publisher the existence and value of data-objects of a given type. When data-object values have been communicated to the publisher through the appropriate data-writer, it is the publisher's responsibility to perform the distribution (the publisher will do this according to its own QoS, or the QoS attached to the corresponding data-writer). A publication is defined by the association of a data-writer to a publisher. This

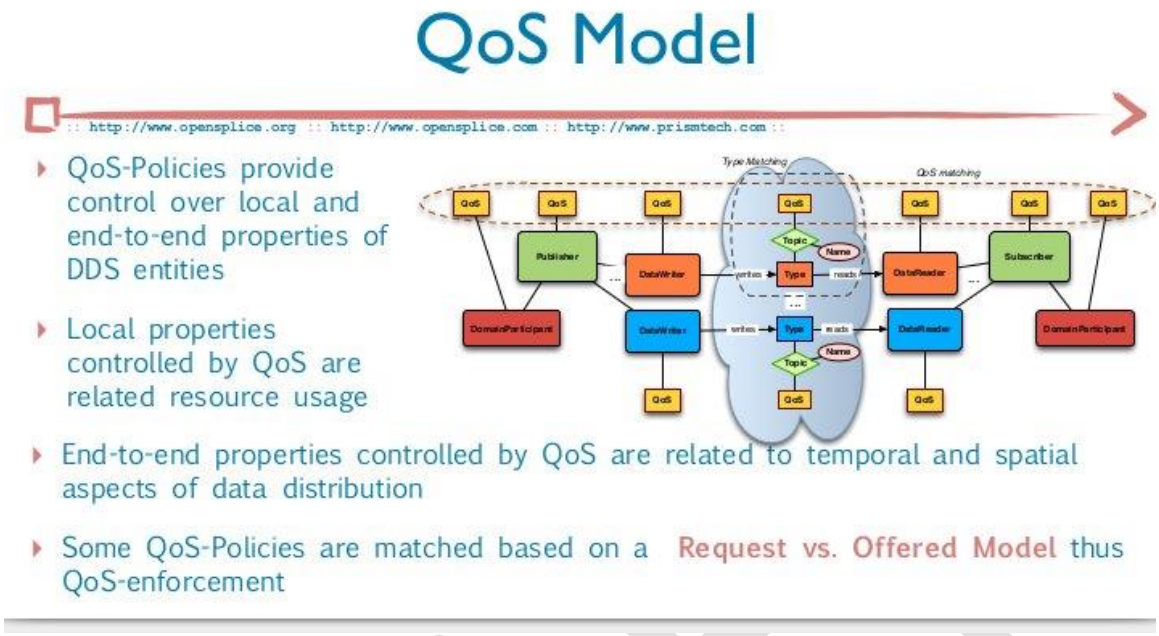
association expresses the intent of the application to publish the data described by the data-writer in the context provided by the publisher.

A Subscriber is an object responsible for receiving published data and making it available (according to the Subscribers QoS) to the receiving application. It may receive and dispatch data of different specified types. To access the received data, the application must use a typed DataReader attached to the subscriber. Thus, a subscription is defined by the association of a data-reader with a subscriber. This association expresses the intent of the application to subscribe to the data described by the data-reader in the context provided by the subscriber.

Topic objects conceptually fit between publications and subscriptions. Publications must be known in such a way that subscriptions can refer to them unambiguously. A Topic is meant to fulfill that purpose: it associates a name (unique in the domain), a data-type, and QoS related to the data itself. In addition to the topic QoS, the QoS of the DataWriter associated with that Topic and the QoS of the Publisher associated to the DataWriter control the behavior on the publisher's side, while the corresponding Topic, DataReader, and Subscriber QoS control the behavior on the subscriber's side.

When an application wishes to publish data of a given type, it must create a Publisher (or reuse an already created one) and a DataWriter with all the characteristics of the desired publication. Similarly, when an application wishes to receive data, it must create a Subscriber (or reuse an already created one) and a DataReader to define the subscription.

QoS (Quality of Service) is a general concept that is used to specify the behavior of a service. Programming service behavior by means of QoS settings offers the advantage that the application developer only indicates what is wanted rather than how this QoS should be achieved. Generally speaking, QoS is comprised of several QoS policies. Each QoS policy is then an independent description that associates a name with a value. Describing QoS by means of a list of independent QoS policies gives rise to more flexibility. QoS settings control multiple aspects of the communications e.g. latency budgets, read and write queues, access to history. The DDS detects and flags incompatible QoS settings.



OpenSplice DDS QoS Model

This specification is designed to allow a clear separation between the publish and the subscribe sides, so that an application process that only participates as a publisher can embed just what strictly relates to publication. Similarly, an application process that participates only as a subscriber can embed only what strictly relates to subscription.

QoS Policy	Applicability	RxO	Modifiable	
DURABILITY	T, DR, DW	Y	N	Data Availability
DURABILITY SERVICE	T, DW	N	N	
LIFESPAN	T, DW	N/A	Y	
HISTORY	T, DR, DW	N	N	
PRESENTATION	P, S	Y	N	Data Delivery
RELIABILITY	T, DR, DW	Y	N	
PARTITION	P, S	N	Y	
DESTINATION ORDER	T, DR, DW	Y	N	
OWNERSHIP	T, DR, DW	Y	N	
OWNERSHIP STRENGTH	DW	N/A	Y	
DEADLINE	T, DR, DW	Y	Y	Data Timeliness
LATENCY BUDGET	T, DR, DW	Y	Y	
TRANSPORT PRIORITY	T, DW	N/A	Y	

QoS Policy	Applicability	RxO	Modifiable	
TIME BASED FILTER	DR	N/A	Y	Resources
RESOURCE LIMITS	T, DR, DW	N	N	
ENTITY FACTORY				Configuration
USER DATA	DP, DR, DW	N	Y	
TOPIC DATA	T	N	Y	
GROUP DATA	P, S	N	Y	
LIVELINESS	T, DR, DW	Y	N	
WRITER DATA LIFECYCLE	DW	N/A	Y	Lifecycle
READER DATA LIFECYCLE	DR	N/A	Y	

OpenSplice DDS QoS Policies

Underlying any data-centric publish subscribe system is a data model. This model defines the global data space and specifies how Publishers and Subscribers refer to portions of this space. The data-model can be as simple as a set of unrelated data structures, each identified by a topic and a type. The topic provides an identifier that uniquely identifies some data items within the global data space. The type provides structural information needed to tell the middleware how to manipulate the data and also allows the middleware to provide a level of type safety. However, the target applications often require a higher-level data model that allows expression of aggregation and coherence relationships among data elements.

The OCS Middleware builds upon multiple layers of software. It is recommended that the highest level (SAL API) methods be utilized whenever possible.

The access levels are:

- SAL (Service abstraction layer)
- OMG DDS (Data Distribution Service)
- OMG RTPS (Real-time Publish Subscribe) - not used directly at present

Each subsequent layer provides more detailed access to the low-level details of the configuration and control of the Topic definitions and/or tuning of real-time behavior.

2.2 SAL (Service abstraction layer)

The SAL provides the highest level of access to the Middleware functionality.

Transparent access to Telemetry, Event and Command objects residing on any subsystem is provided via means of automatic memory mapping of the underlying data objects.

The lower level objects are managed using an implementation of the OMG's DDS.

The currently selected implementation is OpenSplice DDS which has the advantage of open-source licensing (LGPL) , and a community supported version. However, the existence of the SAL permits flexibility in migrating to other DDS solutions if required.

The SAL provides direct access to only a small subset of the total functionality provided by the DDS, reducing both the amount of code required, and its complexity, as seen by the application programmer.

The OMG DDS standard is an evolving entity. It is expected that the prototype C++ API's referenced below, will be replaced by agreed OMG ISOCPP compliant standards of equivalent functionality.

The SAL framework is designed to make this, and other similar transitions, transparent to the application level developers.

2.2.1 SDK Installation

The SAL SDK should be installed on a modern (x86_64) Linux computer. The current baseline recommended configuration is 64-bit CentOS 7.0.

The following packages should also be installed prior to working with the SDK (use either the rpm or yum package managers for CentOS, and apt-get , dpkg, or synaptic for Debian based systems). Appropriate rpms can be found in the rpms subdirectory of the unpacked SDK.

- g++
- make
- ncurses-libs
- xterm
- xorg-x11-fonts-misc
- java-openjdk-devel
- boost-python
- boost-python-devel
- maven
- python-devel
- swig
- tk-devel

The distribution includes dedicated versions of the following packages

- OpenSplice

All the services are built upon a framework of OpenSplice DDS. Code may be autogenerated for a variety of compiled and scripting languages, as well as template documentation, and components appropriate for ingest by other software engineering tools.
A minimum of 800Mb of disk space is required, and at least 1Gb is recommended in order to leave some space for building the test programs.

The default OpenSplice configuration requires that certain firewall rules are added.

For iptables : this can be done (as root) with the following commands

/etc/init.d/iptables stop

or by editing the

/etc/sysconfig/iptables

to add the following lines

-A INPUT -p udp -m udp --dport 250:251 -j ACCEPT

-A INPUT -p udp -m udp --dport 7400:7411 -j ACCEPT
-A OUTPUT -p udp -m udp --dport 250:251 -j ACCEPT
-A OUTPUT -p udp -m udp --dport 7400:7411 -j ACCEPT

The iptables service should then be restarted
/etc/init.d/iptables restart

For firewalld : this can be done (as root) with the following commands
First, run the following command to find the default zone:
firewall-cmd --get-default-zone

Next, issue the following commands:
firewall-cmd --zone=public --add-port=250-251/udp --permanent
firewall-cmd --zone=public --add-port=7400-7411/udp --permanent
firewall-cmd --reload

Replace *public* with whatever the default zone says, if it is different.

Unpack the SAL tar archive in a location of choice (/opt is recommended), e.g. (in a terminal)

cd /opt
tar xzf [location-of-sdk-archive]/salSDK-x.y.z_x86_64.tgz

(where x.y.z is the current release identifier)

and then add the SDK setup command.

source /opt/setup.env

to your bash login profile.

If you chose to install the SDK in a location other than /opt, then you will need to edit the first line of the setup.env script to reflect the actual location.
e.g.
LSST_SDK_INSTALL=/home/saltester

2.2.1 Interfaces - C++

The C++ interface utilizes the OpenSplice C++ PSM as generated by the ddsgen tool. Of the possible wrapper-codes, the "standalone" variant is used.

2.2.2 Interfaces - Java

The Java interface utilizes the OpenSplice Java PSM as generated by the ddsgen tool. Of the possible wrapper-codes, the 'standalone" variant is used.

2.2.3 Interfaces - LabVIEW

The SAL LabVIEW interface provides per-subsystem and per-DataStream specific objects to facilitate application level publishing of all telemetry.

The SAL also provides automatic version and temporal consistency checking and appropriate feedback to the application level code.

2.2.4 Interfaces - Python

The SAL provides 2 C++ based Python wrappers. The first is based on Boost::Python, and the second on the SWIG suite. Each provides a Python loadable shared library implementing all the SAL C++ API calls, as well as wrappers for the data transfer object instantiation.

2.2.5 Interfaces - Engineering and Facility Database

The Engineering and Facility Database (EFD) is a dedicated cluster which records the complete Command, Event, and Telemetry history for the duration of the LSST survey. It exists as two independent identical copies, one at the summit, and another at the base. All data mediated by the SAL communications middleware is automatically saved in SQL database tables on the EFD clusters.

The SAL provides automatically generated instantiation, population and query code for all the defined Command, Event, and Telemetry datatypes, for all subsystems.

2.3 SAL Tools

A combination of methods are provided to facilitate data definition, command definition, and associated generation of code and documentation.

2.3.1 Salgenerator

The Salgenerator tool and associated SDK provide a simple (command line interface) method of interacting with all the tools included in the SAL.

Invocation with no arguments will result in display of the on-line help.

```
SAL generator tool - Usage:
      salgenerator subsystem flag(s)
where flag(s) may be
      validate - check the XML Telemetry/Command/LogEvent definitions
               sal      - generate SAL wrappers for cpp, java, isocpp,
python
               lib      - generate shared library
      compile  - compile a c++ module
      java     - generate JNI interface
      tcl      - generate tcl interface
      html     - generate web form interfaces
      labview  - generate LabVIEW low-level interface
      maven    - generate a maven repository


      db       - generate telemetry database table
                     for db extra arguments required are
            db start-time end-time interval
               where the times are formatted like "2008-11-12 16:20:01"
                     and the interval is in seconds

      shmem  - generate shared memory interface
      sim      - generate simulation configuration
      icd      - generate ICD document
      link     - link a SAL program
      verbose  - be more verbose ;-)
```

A comprehensive installation and user guide is available in LTS-???.

2.4 OMG DDS

The OMG Data-Distribution Service (DDS) is a specification for publish-subscribe data-distribution systems. The purpose of the specification is to provide a common application-level interface that clearly defines the data-distribution service. The specification describes the service using UML, providing a platform-independent model that can then be mapped into a variety of concrete platforms and programming languages.

The goal of the DDS specification is to facilitate the efficient distribution of data in a distributed system. Participants using DDS can read and write data efficiently and naturally with a typed interface. Underneath, the DDS middleware will distribute the data so that each reading participant can access the most-current values. In effect, the service creates a global data space that any participant can read and write. It also creates a name space to allow participants to find and share objects.

DDS targets real-time systems; the API and QoS are chosen to balance predictable behavior and implementation efficiency/performance.

2.4.1 Interfaces - C/C++

See the OpenSplice DDS C++ API documentation for details.

2.4.2 Interfaces - Java

See the OpenSplice DDS Java API documentation for details.

2.5 DDS Tools

2.5.1 Code generation

The DDS standard provides a source code generation tool, the IDL Pre-Processor (idlpp) which can generate DSS interface code for a variety of language/environment combinations. We use the "standalone C++", and "standalone Java" variants. The salgenerator makes use of this tool to generate the basic Topic type support code for all defined Command, Event, and Telemetry datatypes.

2.5.2 Message Translation

OpenSplice Gateway

The OpenSplice Gateway provides support for message translation between a large number of middleware protocols.

By leveraging the Apache Camel integration framework and its support for over 80 connectors, the OpenSplice Gateway is ideal for integrating DDS-interoperable applications with proprietary as well as standards-based messaging technologies, such as JMS and AMQP, as well as user applications leveraging Web standards such

as W3C Web Services, REST and HTML5 WebSockets (See http://camel.apache.org for details). The current SAL does not make use of the Gateway package, but it will the preferred mechanism for message translation if future requirements necessitate.

2.5.3 Debug

Opensplice Tuner

The OpenSplice Tuner is a deployment tool within PrismTech's OpenSplice DDS suite. This tool offers total control over a deployed OpenSplice based DDS-system from any local or remote platform that supports the Java language.

The Java based OpenSplice Tuner tool aids the design, implementation, test and maintenance of OpenSplice based distributed systems (the OpenSplice Tuner is available both as a 'standalone' Java-program as well as an Eclipse plug-in for the Productivity tool suite).

The OpenSplice Tuner's features target all lifecycle stages of distributed system development and can be summarized as:

- Design: During the design phase, once the information model is established (i.e. topics are defined and 'registered' in a runtime environment, which can be both a host-environment as well as a target-environment), the Tuner allows creation of publishers/writers and subscribers/readers on the fly to experiment and validate how this data should be treated by the middleware regarding persistence, durability, latency, etc.
- Implementation: During the implementation phase, where actual application-level processing and distribution of this information is developed, the OpenSplice Tuner allows injection of test input-data by creating publishers and writers 'on the fly' as well as validating the responses by creating subscribers and readers for any produced topics.
- Test: during the test phase, the total system can be monitored by inspection of data (by making 'snapshots' of writer- and reader-history caches) and behavior of readers & writers (statistics, like how long data has resided in the reader's cache before it was read) as well as monitoring of the data-distribution behavior (memory-usage, transport-latencies).
- Maintenance: Maximum flexibility for planned and 'ad-hoc' maintenance is offered by allowing the Tuner tool (which can be executed on any JAVA enabled platform without the need of OpenSplice to be installed) to remotely connect via the web-based SOAP protocol to any 'reachable' OpenSplice system around the world (as long as an HTTP-connection can be established with the OpenSplice computing-nodes of that system). Using such a dynamic-connection, critical data may be logged and data-sets may be 'injected' into the system to be maintained (such as new settings which can be automatically 'persisted' using the QoS features as offered by the 'persistence-profile supported by OpenSplice).

OpenSplice Tester

This Java based tool is designed with the systems integrator in mind and offers an intuitive set of features to aid his task, offering both local operation (where the tool is running on a deployed DDS-system) as well as remote operation (where the tool is connect over SOAP to a remotely deployed DDS-system).

The main features of the OpenSplice Tester are:

- Automated testing of DDS-based systems
  - Dynamic discovery of DDS entities
  - Domain-Specific scripting Language (DSL) for test scenario's
- Batch execution of regression tests
  - Debugging of distributed DDS system
  - One-click definition of a monitoring-time-line
  - Analysis/comparison of topics/instances & samples
  - Virtual topic-attributes to dramatically ease analysis
  - System-browser of DDS entities (app's/readers/writers)
  - Connectivity and QoS-conflict monitoring/detection
  - Statistics-monitoring of applications and services
- Integrated IDE
  - Syntax highlighting editor, script-executor and Sample Logger
  - One-click relations between script, logs and timeline
  - Optional integration of message-interfaces with DDS interactions

DDS touchstone

DDS Touchstone is a scenario-driven Open Source benchmarking framework for evaluating the performance of OMG DDS compliant implementations. It is an open-source project.

Its main characteristics are:

- A deployment, language, OS and vendor neutral benchmarking suite
- Measures and outputs roundtrip latencies and throughput numbers
- Supports creation of many different kinds of DDS scenarios
- Supports recording and replaying scenarios

DDS Benchmark Environment

The DDS Benchmarking Environment may be used to test the DDS implementation. The DBE consists of scripts to automate test running on a distributed network. The DBE is the kernel of what may later become a unified testing architecture for any type of middleware.

This DBE is freely available from the Real-Time DDS Examination & Evaluation Project (RT-DEEP)

2.6 OMG RTPS wire protocol

The OpenSplice DDS can utilize the OMG standard Real-time-Publish-Subscribe wire protocol. This enables interoperability with other vendors implementations.

The RTPS mode is the default for all SAL OpenSplice installations.

The specification defines the message formats, interpretation, and usage scenarios that underlie all messages exchanged by applications that use the RTPS protocol.

See the RTPS Specification Document for details.

2.7 Security and Integrity

2.7.1 General policies

Refer to http://dev.lsstcorp.org/trac/attachment/wiki/Security/Security Policy documents.zip

2.7.2 Firewall

A firewall's basic task is to regulate the flow of traffic between computer networks of different trust levels. Typical examples are the Internet which is a zone with no trust and an internal network which is a zone of higher trust. A zone with an intermediate trust level, situated between the Internet and a trusted internal network, is often referred to as a perimeter network or Demilitarized zone (DMZ).

2.7.3 Packet filtering

Packet filters act by inspecting the packets which represent the basic unit of data transfer between computers on the Internet. If a packet matches the packet filter's set of rules, the packet filter will drop (silently discard) the packet, or reject it (discard it, and send error responses to the source).

This type of packet filtering pays no attention to whether a packet is part of an existing stream of traffic (it stores no information on connection state). Instead, it filters each packet based only on information contained in the packet itself (most commonly using a combination of the packet's source and destination address, its protocol, and, for TCP and UDP traffic, which comprises most internet communication, the port number).

Because TCP and UDP traffic by convention uses well known ports for particular types of traffic, a stateless packet filter can distinguish between, and thus control, those types of traffic (such as web browsing, remote printing, email transmission,

file transfer), unless the machines on each side of the packet filter are both using the same non-standard ports. Second Generation firewalls do not simply examine the contents of each packet on an individual basis without regard to their placement within the packet series as their predecessors had done, rather they compare some key parts of the trusted database packets. This technology is generally referred to as a 'stateful firewall' as it maintains records of all connections passing through the firewall, and is able to determine whether a packet is the start of a new connection, or part of an existing connection. Though there is still a set of static rules in such a firewall, the state of a connection can in itself be one of the criteria which trigger specific rules.

This type of firewall can help prevent attacks which exploit existing connections, or certain Denial-of-service attacks, including the SYN flood which sends improper sequences of packets to consume resources on systems behind a firewall.

2.7.4 Private subnet

Firewalls often have network address translation (NAT) functionality, and the hosts protected behind a firewall commonly have addresses in the private address range, as defined in RFC 1918. Firewalls often have such functionality to hide the true address of protected hosts. Originally, the NAT function was developed to address the limited amount of IPv4 routable addresses that could be used or assigned to companies or individuals as well as reduce both the amount and therefore cost of obtaining enough public addresses for every computer in an organization. Hiding the addresses of protected devices has become an increasingly important defense against network reconnaissance.

2.7.5 DDS domains

The domain is the basic construct used to bind individual applications together for communication. A distributed application can elect to use a single domain for all its data-centric communications.

All Data Writers and Data Readers with like data types will communicate within this domain. DDS also has the capability to support multiple domains, thus providing developers a system that can scale with system needs or segregate based on different data types. When a specific data instance is published on one domain, it will not be received by subscribers residing on any other domains. The normal operating mode for LSST will be to use a single domain

Multiple domains provide effective data isolation. This allows the clean separation of otherwise identical components. For example it may be necessary to operate both the on-telescope rotator, and the backup rotator simultaneously, therefore utilizing a second “test” domain for the latter.

3.0 Detailed Interface Design

A comprehensive UML model detailing the Interface Requirements is available in LSE-70 and LSE-209. The following sections are intended to supply a more verbose english language description of the major elements.

3.1 Commanding

There are two basic classes of commands used:

Lifecycle commands: commands used by OCS to control the lifecycle characteristics of applications

Functional commands: commands that implement the specific functional characteristics of a subsystem components.

Functional operation is based on the Command/Action/Response model that isolates the transmission of the command from the resulting action that is performed. When an application receives a command, it validates any parameter associated with that command and immediately accepts or rejects the command. If the command is accepted, the application then initiates an independent internal action to meet the conditions imposed by the command. Once those conditions have been met, an event is posted signifying the successful completion of the action (or the unsuccessful completion if the conditions cannot be met).

Commands return immediately but the actions that are initiated as a result of a command may take some time to complete. When the action completes, an action status acknowledgment is posted that includes the completion status of that action.

If a command is accepted by the subsystem it causes an independent action to begin. A response to the command is returned immediately. The action begins matching the current configuration to the new demand configuration. When the configurations match (i.e., the subsystem has performed the input operations) the commandee signals the successful end of the action. If the commands cannot be matched (whether by hardware failure, external stop command, timeout, or some other fault) the commandee signals the unsuccessful end of the action.

The important features of the command/action/response model are:

- Commands are never blocked. As soon as one command is started, another one can be issued. The behavior of the controller when two or more commands are started can be configured on a per subsystem basis. Commands which take a significant amount of time to process may indicated by the subsystem indicating a busy substate (of the enabled state).
- The actions are performed using one or more separate threads. They can be tuned for priority, number of simultaneous actions, critical resources, or any other parameters.

- Action completions produce events that tell the state of the current configuration. Actions push the lifecycle of the command through to completion.
- Responses may be monitored by any other subsystems.

Generic subsystem control state commands

All subsystems support the lifecycle commands as defined in LSE-209. These are used to initiate transitions in the subsystem state machine. Each transition is recorded by the generation of an Event by the subsystem.

3.1.1 Interface Processing Time

Timing constraints are requirements on the middleware layer. It remains incumbent on the application level code to be "ready" to accept and process the messages in a similarly timely fashion.

3.1.2 Message Metadata

Every stream includes items for consistency checking and performance monitoring support.

3.2 Telemetry Archiving

Telemetry data issued via the middleware is received by the computer system(s) of the Facility database(s), and archived.

3.3 Events

Any application may post Events and/or subscribe to Events posted elsewhere. The Event service is robust and high performance. An Event consists of a topic and a severity. The topic is used to identify publishers to subscribers. The severity may be used as a filter by subscribers.

The Event service has the following general properties: An Event topic represents a many to many mapping: Events may be posted to the topic from more than one source and received by zero or more targets. (Typically, however, most topics will have a single source.)

Events posted by a single source into an Event topic are received by all targets in the same order as they were posted.

Delivery of Events to one subscriber cannot be blocked by the actions of another subscriber.

Events are automatically tagged with the source and a timestamp.

3.4 Communication Methods

3.4.1 Initiation: DDS discovery

Discovery is the process by which domain participants find out about each other's entities. Each participant maintains a database on other participants in the domain, and their entities. This happens automatically behind the scenes (anonymous publish-subscribe).

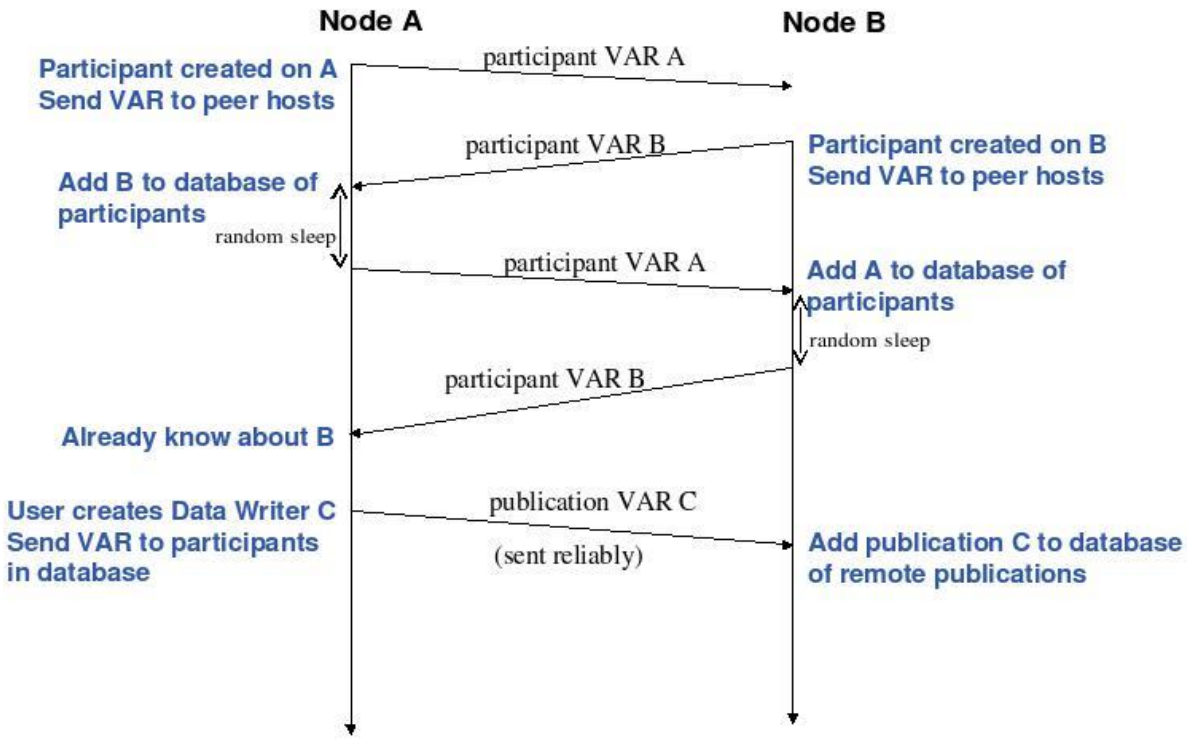
Participant discovery has the following properties:

- Discovery does not cross domain boundaries
- Dynamic discovery
- Participants must refresh their presence in the domain or will be aged out of database, this functionality is provided by the DDS
- QoS changes are propagated to remote participants
- Discovery has two consecutive phases
- Participant discovery phase
- Participants discover each other
- Best-effort communication
- Endpoint discovery phase
- Participants exchange information about their datawriter and datareader entities
- Reliable communication
- Steady state traffic to maintain liveliness of participants
- Participants periodically announce their presence using RTPS VAR message
- Contains participant GUID, transport locators, QoS
- Initially sent to all participants in initial peers list, then sent periodically to all discovered participants
- Sent using best-effort

DataWriter/DataReader discovery has the following properties:

- Send out pub/sub RTPS VAR (entity attributes) message to every new participant
- NACK for pub/sub info if not received from a known participant
- Send out changes/additions/deletions to each participant
- Uses reliable communication between participants

- Data Distribution Service matches up local and remote entities to establish communication paths



Discovery is implemented using DDS entities known as Built-in Data Writers and Built-in Data Readers

- Uses same infrastructure as user defined Data Writers/Data Readers
- Participant data is sent best effort
- Publication/subscription data is sent reliably

Three Built-in topics (keyed):

- DCPSParticipant
- DCPSPublication
- DCPSSubscription

Each participant on the same host and in the same domain requires a unique participant index

For given domain, participant index determines port numbers used by the participant

3.4.2 Flow Control: DDS topics

Topics provide the basic connection point between publishers and subscribers. The Topic of a given publisher on one node must match the Topic of an associated

subscriber on any other node. If the Topics do not match, communication will not take place.

A Topic is comprised of a Topic Name and a Topic Type. The Topic Name is a string that uniquely identifies the Topic within a domain. The Topic Type is the definition of the data contained within the Topic. Topics must be uniquely defined within any one particular domain. Two Topics with different Topic Names but the same Topic Type definition would be considered two different Topics within the DDS infrastructure.

3.5 Security Requirements

3.5.1 Message timestamps

Message integrity is enhanced by the inclusion of egress-time and arrival time (local system clocks) field in every topic (command, notification, and telemetry). The SAL software automatically performs validation to ensure early detection of clock slew or other time related problems. The format used is PTP TAI time. Utility routines are provided to translate between this form and human readable UTC time and date forms. The SAL API provides access to the current TAI time, and the ability to issue a callback at a future TAI time (within kernel scheduling constraints).

The TAI time will also be propagated within subsystems (eg Camera, Mount) using appropriate high precision timing hardware.

3.5.2 Software versioning checksums

Communications consistency and security is supported by the inclusion of CRC checksum fields in every topic definition (command, notification, and telemetry). The SAL software automatically checks that the publisher and subscribers are running code generated using identical (at the source code level) topic definitions. This prevents problems associated with maintaining consistent inter-subsystem interfaces across a widely distributed software development infrastructure.

4.0 Qualification methods

4.1 System dictionary

A system wide dictionary of all subsystems, devices, actions and states is maintained. All the interactions between subsystems are automatically checked to verify that only objects defined in the dictionary can be used or referenced. The System Dictionary contents forge a link between the communications interfaces as described in the subsystem ICD's, and the code generation facilities of the SAL. This is used to ensure consistency all the way from ICD through implementation in code, and finally to the historical recording of all activity in the EFD.

4.2 Command definition database

A database of permissible commands is maintained on a per subsystem basis. The database references the system dictionary and contains 1 record per command. The XML format definitions are maintained in a Stash repository and updated upon agreement between T&S and the subsystem engineers.

Each command is constrained in terms of target subsystems, parameter ranges, maximum frequency, timeout, pause/hold potential, and failure severity.

The database is used to automatically generate application level code to perform all command level interactions. This code is thus guaranteed to be consistent system wide.

4.3 Telemetry DataStream Definition database

All telemetry DataStream definitions are stored in a database. Each definition is automatically verified for compliance with the system dictionary. Telemetry items are detailed in terms of type, size, frequency, units, and value ranges. Any item with a physical correlate has an SI unit associated with it.

The database is used to automatically generate application level code to perform all DataStream topic references. This code is thus guaranteed to be consistent system wide. The XML format definitions are maintained in a Stash repository and updated upon agreement between T&S and the subsystem engineers (subject to approval of the new ICD by the CCB).

Events are generated using the same database, and comprise a special category of topics which may assigned the highest priority (ALERT category) if appropriate.

Another special category of Events is associated with "State Transitions". Each subsystem is required to publish an appropriate Event on every major transition. These Events may optionally include minor substate information.

Subsystems are also required to publish the details of any applied "Configuration" as soon as it is complete. Thus ensuring that all the details of all possible configurations are also captured in the XML definitions , and also recorded in the EFD.

4.4 Code generation

The primary implementation of the software interface described in this document is automatically generated. The Service Abstraction layer (SAL) provides a standardized wrapper for the low-level OMG DDS functionality which provides the transport layer.

The permissible commands, DataStream contents, and issuable alerts are all defined by the controls system database and their nomenclature is controlled by the system dictionary. The XML format definitions of all the SAL objects is maintained in a Stash repository and updated upon agreement between T&S and the subsystem engineers.

All inter-subsystem messages formats are auto generated. Low level data transfers include versioning checksums based on the source level record definition.

4.5 Testing and simulation

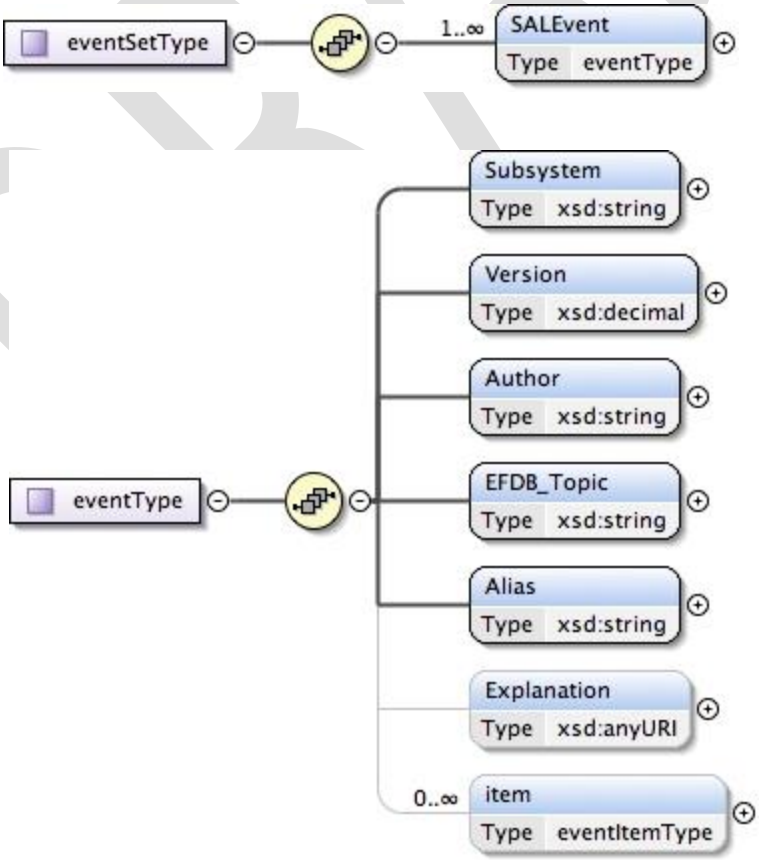
Test servers and clients are generated which implement the full set of commands, Telemetry, and Events as defined by the SAL object XML definitions. Tests may be configured for a variable number of servers/clients and automatically monitored to ensure compliance with bandwidth and latency requirements. All test results are archived to the facility database for future examination.

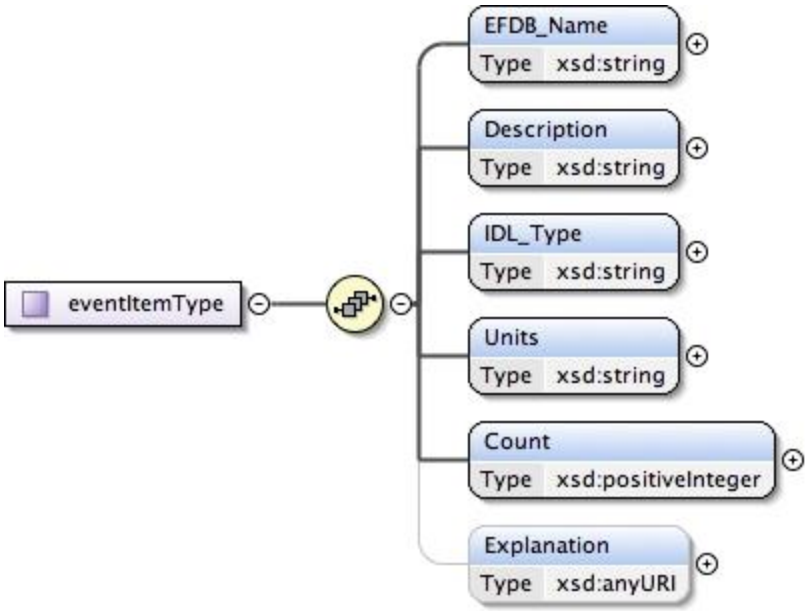
5.0 XML Schema

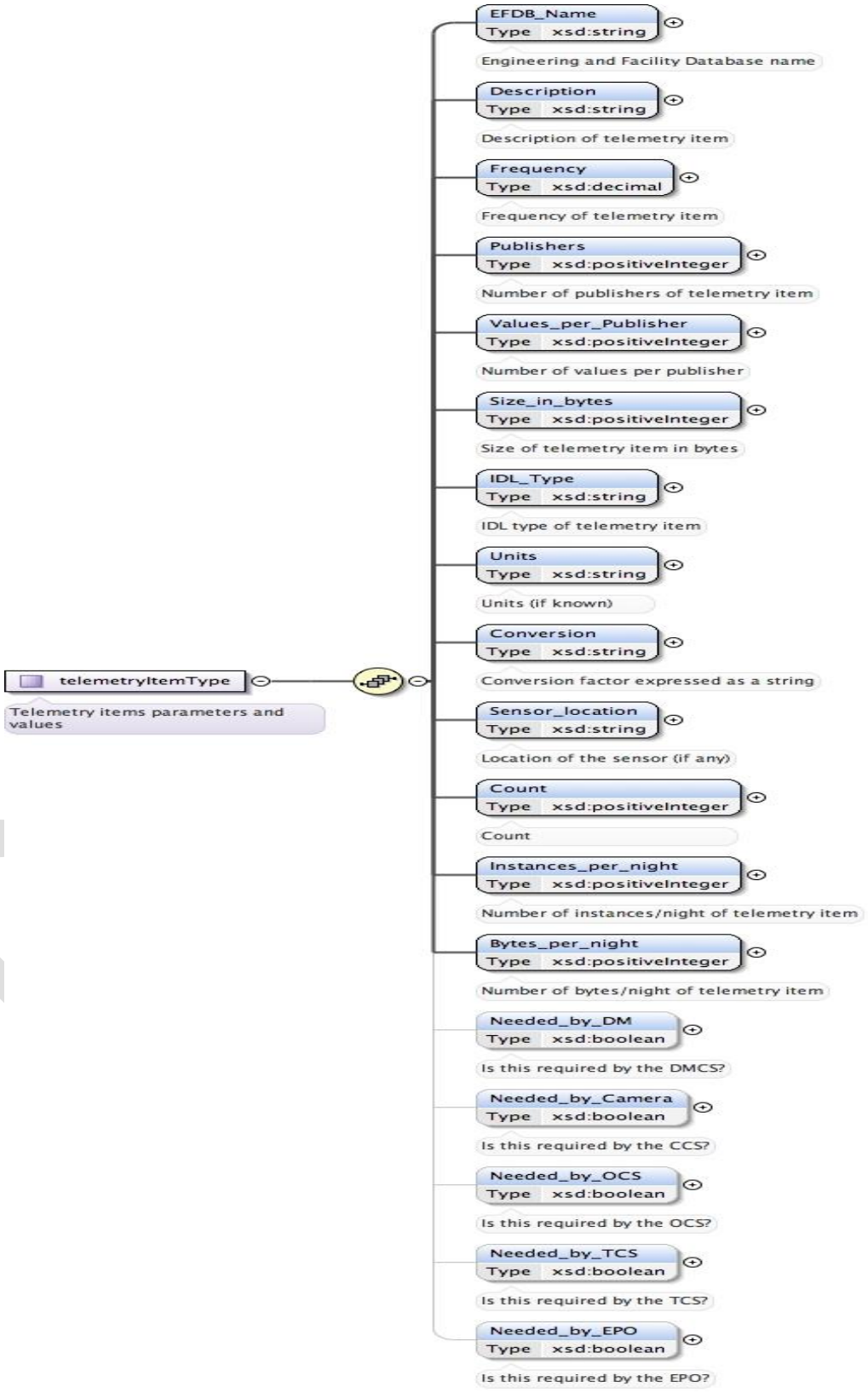
The definitive versions of the schema can be found at

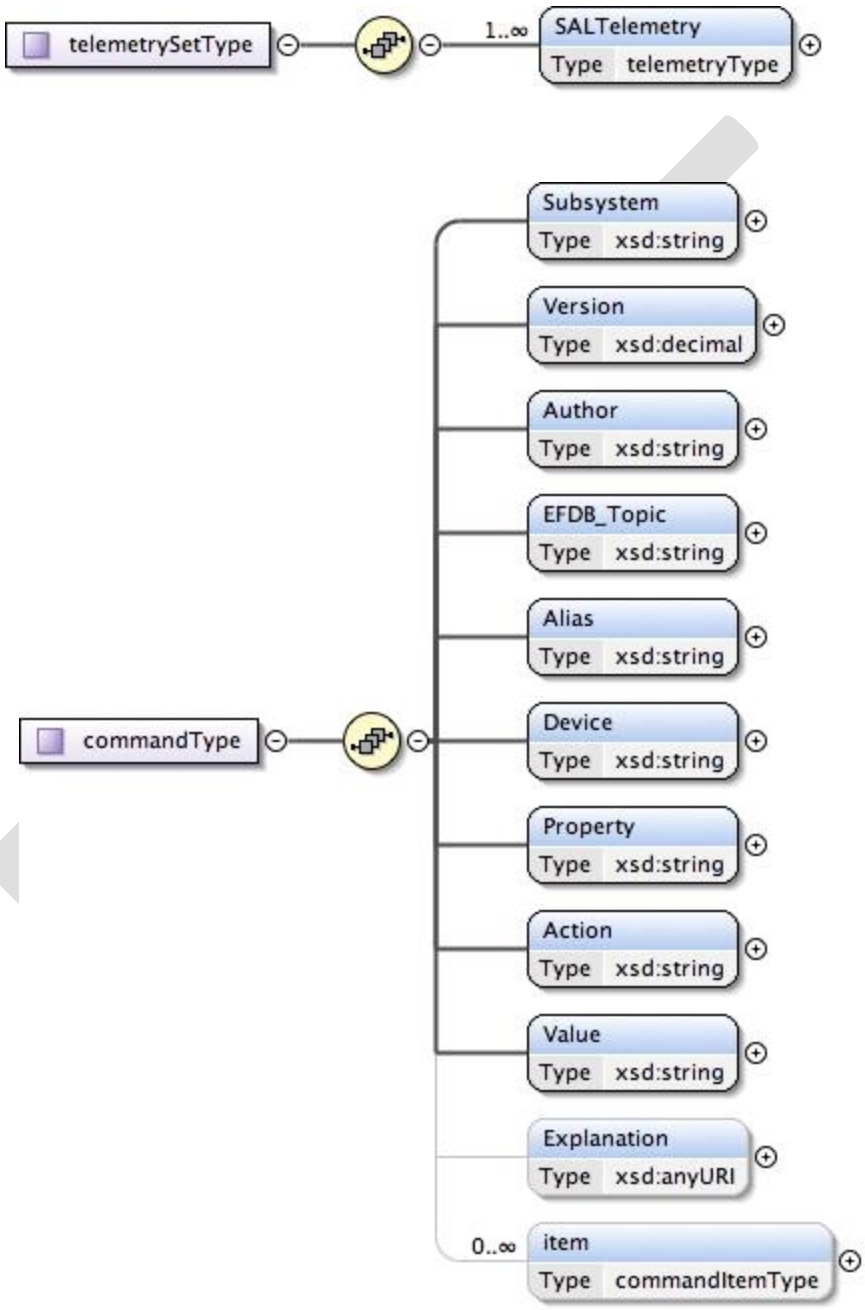
https://project.lsst.org/ts/[name-of-schema].xsd, where [name-of-schema] is of the form SAL[Object]Set.xsd, and [Object] is one of Command, Event, or Telemetry.

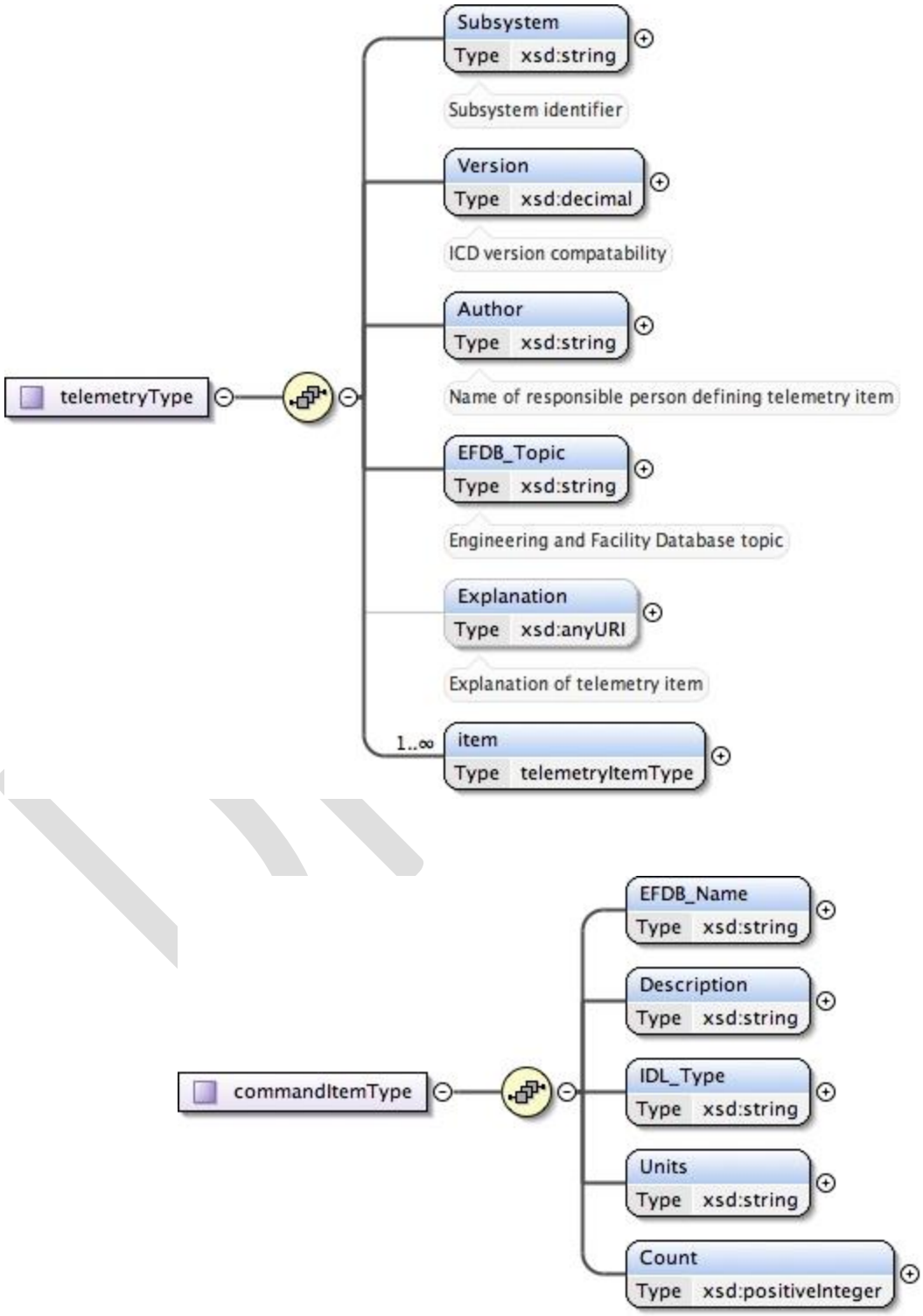
The definitive versions of the current definitions for each subsystem can be found at http://stash.lsstcorp.org/projects/TS/repos/ts_xml/browse

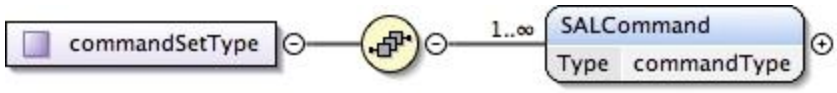



eventItemType
EFDB_Name
Type xsd:string
Description
Type xsd:string
IDL_Type
Type xsd:string
Units
Type xsd:string
Count
Type xsd:positiveInteger
Explanation
Type xsd:anyURI


telemetryItemType
Telemetry items parameters and values
EFDB_Name
Type xsd:string
Engineering and Facility Database name
Description
Type xsd:string
Description of telemetry item
Frequency
Type xsd:decimal
Frequency of telemetry item
Publishers
Type xsd:positiveInteger
Number of publishers of telemetry item
Values_per_Publisher
Type xsd:positiveInteger
Number of values per publisher
Size_in_bytes
Type xsd:positiveInteger
Size of telemetry item in bytes
IDL_Type
Type xsd:string
IDL type of telemetry item
Units
Type xsd:string
Units (if known)
Conversion
Type xsd:string
Conversion factor expressed as a string
Sensor_location
Type xsd:string
Location of the sensor (if any)
Count
Type xsd:positiveInteger
Count
Instances_per_night
Type xsd:positiveInteger
Number of instances/night of telemetry item
Bytes_per_night
Type xsd:positiveInteger
Number of bytes/night of telemetry item
Needed_by_DM
Type xsd:boolean
Is this required by the DMCS?
Needed_by_Camera
Type xsd:boolean
Is this required by the CCS?
Needed_by_OCS
Type xsd:boolean
Is this required by the OCS?
Needed_by_TCS
Type xsd:boolean
Is this required by the TCS?
Needed_by_EPO
Type xsd:boolean
Is this required by the EPO?


telemetrySetType
1..∞
SALTelemetry
Type telemetryType
commandType
Subsystem
Type xsd:string
Version
Type xsd:decimal
Author
Type xsd:string
EFDB_Topic
Type xsd:string
Alias
Type xsd:string
Device
Type xsd:string
Property
Type xsd:string
Action
Type xsd:string
Value
Type xsd:string
Explanation
Type xsd:anyURI
0..∞
item
Type commandItemType


telemetryType
Subsystem
Type xsd:string
Subsystem identifier
Version
Type xsd:decimal
ICD version compatability
Author
Type xsd:string
Name of responsible person defining telemetry item
EFDB_Topic
Type xsd:string
Engineering and Facility Database topic
Explanation
Type xsd:anyURI
Explanation of telemetry item
1..∞
item
Type telemetryItemType
commandItemType
EFDB_Name
Type xsd:string
Description
Type xsd:string
IDL_Type
Type xsd:string
Units
Type xsd:string
Count
Type xsd:positiveInteger



5.1 Interface Definition Examples

5.1.1 Telemetry Definition

A very simple XML schema is used to define a telemetry topic.
The topic is the smallest unit of information which can be exchanged using the SAL mechanisms.

The following Reserved words may NOT be used in names and will flag an error at the validation phase (once the SAL System Dictionary is finalized, the item names will also be validated for compliance with the dictionary).

Reserved words : *bstract any attribute boolean case char component const consumes context custom default double emits enum eventtype exception factory false finder fixed float getraises home import in inout interface local long module multiple native object octet oneway out primarykey private provides public publishes raises readonly sequence setraises short string struct supports switch true truncatable typedef typeid typeprefix union unsigned uses valuebase valuetype void wchar wstring*
e.g.

```
<SALTelemetry>
<Subsystem>hexapod</Subsystem>
<Version>2.5</Version>
<Author>A Developer</Author>
<EFDB_Topic>hexapod_LimitSensors</EFDB_Topic>
    <item>
       <EFDB_Name>liftoff</EFDB_Name>
       <Description>State of lift-off sensors</Description>
       <Frequency>0.054</Frequency>
       <IDL_Type>short</IDL_Type>
       <Count>18</Count>
    </item>
    <item>
       <EFDB_Name>limit</EFDB_Name>
       <Description></Description>
       <Frequency>0.054</Frequency>
       <IDL_Type>short</IDL_Type>
       <Count>18</Count>
    </item>
```

</SALTelemetry>

Notes : <Frequency> is advisory and used for network bandwidth and database sizing estimates. Default is a minimum of once per exposure.

5.1.2 Command Definition

The process of defining supported commands is similar to Telemetry using XML. The command aliases correspond to the ones listed in the relevant subsystem ICD. e.g.

```
<SALCommand>
 <Subsystem>hexapod</Subsystem>
 <Version>2.5</Version>
 <Author>salgenerator</Author>
 <EFDB_Topic>hexapod_command_configureAcceleration</EFDB_Topic>
 <Alias>configureAcceleration</Alias>
 <Device>drive</Device>
 <Property>acceleration</Property>
<Explanation>http://sal.lsst.org/SAL/Commands/hexapod_command_configureAcceleration.html
</Explanation>
   <item>
    <EFDB_Name>xmin</EFDB_Name>
    <Description>Minimum x accelleration </Description>
    <IDL_Type>double</IDL_Type>
    <Count>1</Count>
   </item>
   <item>
    <EFDB_Name>xmax</EFDB_Name>
    <Description>Maximum x accelleration </Description>
    <IDL_Type>double</IDL_Type>
    <Count>1</Count>
   </item>
   <item>
    <EFDB_Name>ymin</EFDB_Name>
    <Description> Minimum y accelleration</Description>
    <IDL_Type>double</IDL_Type>
    <Count>1</Count>
   </item>
   <item>
    <EFDB_Name>ymax</EFDB_Name>
    <Description>Maximum y accelleration </Description>
    <IDL_Type>double</IDL_Type>
    <Count>1</Count>
   </item>
</SALCommand>
```

5.1.3 Log Event Definition

Events are defined in a similar fashion to commands. e.g

The Log Event aliases are as defined in the relevant ICD.

e.g.

```
<SALEvent>
 <Subsystem>hexapod</Subsystem>
 <Version>2.4</Version>
 <Author>salgenerator</Author>
 <EFDB_Topic>hexapod_logevent_limit</EFDB_Topic>
 <Alias>limit</Alias>
 <Explanation>http://sal.lsst.org/SAL/Events/hexapod_logevent_limit.html</Explanation>
   <item>
    <EFDB_Name>priority</EFDB_Name>
    <Description>Severity of the event</Description>
    <IDL_Type>long</IDL_Type>
    <Count>1</Count>
   </item>
   <item>
    <EFDB_Name>axis</EFDB_Name>
    <Description>nName of axis </Description>
    <IDL_Type>string</IDL_Type>
    <Count>1</Count>
   </item>
   <item>
    <EFDB_Name>limit</EFDB_Name>
    <Description>Type of limit </Description>
    <IDL_Type>string</IDL_Type>
    <Count>1</Count>
   </item>
   <item>
    <EFDB_Name>type</EFDB_Name>
    <Description>Condition code </Description>
    <IDL_Type>string</IDL_Type>
    <Count>1</Count>
   </item>
</SALEvent>
```