EXPERIENCES IN MESSAGING MIDDLEWARE FOR HIGH-LEVEL CONTROL APPLICATIONS*

Nanbor Wang†, Svetlana Shasharina, James Matykiewicz, and Rooparani Pundaleeka
Tech-X Corporation, Boulder, CO 80303, U.S.A.

Abstract

Existing high-level applications in accelerator control and modeling systems leverage many different languages, tools and frameworks that do not interoperate with one another. As a result, the accelerator control community is moving toward the proven Service-Oriented Architecture (SOA) approach to address the interoperability challenges among heterogeneous high-level application modules. Such SOA approach enables developers to package various control subsystems and activities into “services” with well-defined “interfaces” and make leveraging heterogeneous high-level applications via flexible composition possible. Examples of such applications include presentation panel clients based on Control System Studio (CSS) and middle-layer applications such as model/data servers.

This paper presents our experiences in developing a demonstrative high-level application environment using emerging messaging middleware standards. In particular, we utilize new features in EPICS v4 and other emerging standards such as Data Distribution Service (DDS) and Extensible Type Interface by the Object Management Group. We first briefly review examples we developed previously. We then present our current effort in integrating DDS into such a SOA environment for control systems. Specifically, we illustrate how we are integrating DDS into CSS and develop a Python DDS mapping.

BACKGROUND AND INTRODUCTION

Accelerator control systems (ACS) coordinate the interactions among control hardware, data acquisition instruments, logging and data storage devices, and operator's interface. High-level accelerator control applications encompass activities such as operator control panels, tune measurement, orbit control, parameter save/restore, feedback, optic optimization, and parameter scanning that allow physicists and operators to control and reason accelerator behaviors in physically meaningful abstractions. There exist many tools and frameworks to help bring modern software engineering practices to the development and integration of lower-level hard real-time controls and the high-level soft real-time applications with great success.

Emerging Trends and Challenges

Control systems are often built on top of a set of existing tools and platforms that suit the needs of their target platforms. For examples, the EPICS [1] toolkit provides a standard for low-level controller architecture and a set of interoperable tools and engineering applications to assist control system developments. Depending on the scale of the target accelerator, high-level applications are often developed as a monolithic Graphical User Interface (GUI), a simple script, or a library routine. As in the case of generalized ACS control environments, many tools and frameworks such as Unified Accelerator Language (UAL) and Matlab Middle Layer Toolkit (MMLT), are available to assist the integration and interaction among high-level applications and device controllers (built, *e.g.*, using EPICS.)

All the different development environments and tools do not generally interoperate with one another. This is not a major issue for small- or medium-sized accelerators. However, such *ad hoc* approach no longer scales for modern large-scale accelerator facility such as the new NSLS-II, Project X, and the Intensity Frontier. Several Design Reports have called for a separate “service tier/middle layer” to provide devices and functional abstractions as units of integration.

SOA: Service-Oriented Architecture

SOA [2,3] has gained widespread acceptance in the business/enterprise software world as it has shown to facilitate the integration and composition of disparate software services across enterprises and businesses boundaries. Applying SOA principles in ACS is a promising approach in isolating and managing the complexity. In fact, many existing accelerator control systems have already adopted many SOA guidelines and principles. To address the needs and challenges of next-generation, large-scale accelerator control systems, we are enhancing the SOA environment for next-generation large-scale accelerator control systems to manage the complexity and contain the cost of developing future accelerator control systems and upgrading existing ones.

Message-Oriented Middleware

Traditionally, a SOA is often constructed on top of point-to-point request-reply, client-server communication middleware technologies. Middleware serves as the standard communication bus among different service. Examples of well-established SOA middleware include SOAP-based and RESTful Web Services, CORBA and Java RMI. However, there are still limitations with these point-to-point, request-reply, RPC-styled communication model as they impose scalability issues as the size and complexity of accelerator control systems grow.

The emerging DDS [4] is a new class of Message-Oriented Middleware (MOM) standard specified by the

*Work partially supported by US Department of Energy under contracts #DE-FG02-08ER85043 and #DE-SC-0000842, and Tech-X Corporation.
†nanbor@txcorp.com

Object Management Group (OMG). DDS complements RPC-styled client-server middleware and address their many limitations. DDS is a natural extension to many existing accelerator control frameworks. Furthermore, because DDS inherently supports many quality-of-service (QoS) policies such as message priority, rate, reliability, and deadline, that are necessary for mission-critical applications, DDS is a natural selection to act as the alternative service-bus in a SOA for control systems.

Figure 1 illustrates a SOA for high-level accelerator environment where applications exchange data using DDS as the common standard service bus. As shown in the figure, client applications can readily act as gateways to another enterprise service bus such as Web Services.

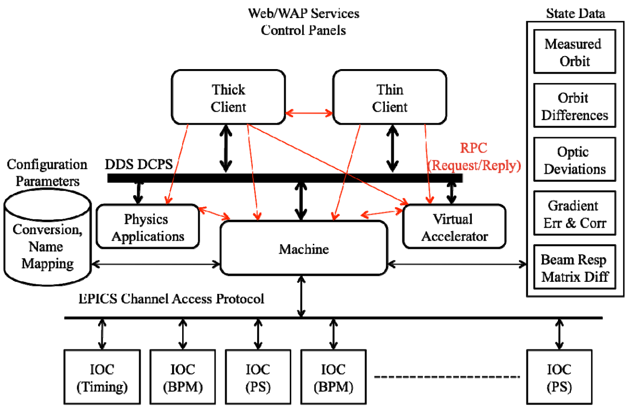

Figure 1: SOA for high-level applications over DDS.

DESIGNING DDS APPLICATION THROUGH PERFORMANCE TESTING

Because there are many ways to configure a DDS system, to better emulate various operation scenarios easily, we developed a DDS performance test suite and performed benchmarking tests with it. With the test framework, users will be able to develop various DDS runtime scenarios and experiment with various QoS policy combinations and evaluate their effect to the overall system performance. We design the test suite to be portable so that ACS developers can evaluate different DDS implementations easily.

Our performance test suite is built on top of the generic benchmarking application similar to the open-source Touchstone performance tool. Touchstone's benchmarking application provide the mechanism to instantiate test components such as transceiver and transponder for latency test, via special DDS messages for a set of control topics. Users can design and instantiate tests of different scales and with different combinations of QoS policies easily. Such approach allows users to create scenario-based performance evaluations easily.

Accessing the performance of messaging systems and overall applications provide critical information to help making key design decisions such as:

- Selection of DDS implementations: Different DDS implementations make different tradeoffs and adopt different implementation strategies to realize the movement of data from publishers to subscribers according to the QoS policies specified by all the entities involved. Some implementations also make such strategies configurable. Therefore, certain implementations or configurations perform better under certain operation environment while others ay scale better.
- Assisting in DDS configuration: DDS supports a rich set of QoS policies. Configuring a system using different sets of policies can affect the overall performance in different ways. For example, setting priority on one data stream can affect the overall behaviors of other data streams. Being able to perform tests to observe system behaviors at similar scales can provide essential guidance on design strategies.

EXAMPLE MIDDLE-LAYER SERVERS

To demonstrate the three-tier high-level application architecture, we implemented a general-purpose web-based optimization service with the help from our collaborator at Brookhaven National Lab (BNL). The service allows users to perform Twiss calculation and lattice optimization over the web.

Figure 2 illustrates the overall architecture of such a general-purpose web-based optimization service. This simple service allows accelerator physicists to submit lattice and optimization files to the service and returns the visual results of twiss calculations. This example provides

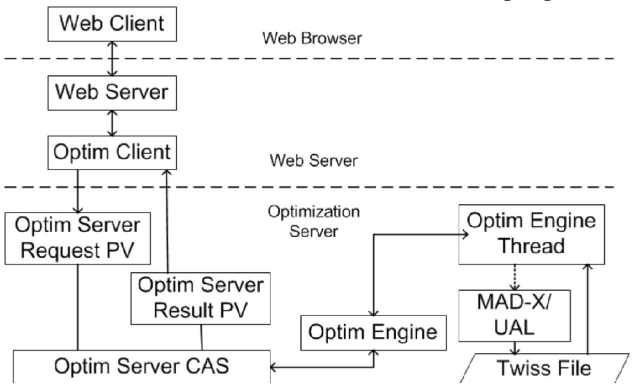

Figure 2: Example architecture of a middle-layer server.

an example "Software as a Service (SaaS)" prototype. It demonstrates the core idea on SOA using DDS and lays down the foundation for further, more complicated services. We implemented the prototype service using both MAD-X and UAL as the underlying compute engine. For messaging between the Web Server and the Optimization Server, we used both DDS and the experimental EPICS-DDS [5].

INTEGRATING DDS WITH THE CONTROL SYSTEM STUDIO

Fermilab's Intensity Frontier examplifies another usage scenario for a dual-message-bus SOA. There, they employ both DDS and EPICS in the control system. Subsystems that require real-time responses are built on top of EPICS. Higher-level applications use DDS instead as the messaging mechanism.

There is a need to aggregate information and control various systems running on either EPICS or DDS together. Therefore, we are developing plug-ins for publishing and subscribing DDS topics from within the Control System Studio (CSS) [6]. The prototype plug-ins are modeled after org.csstudio.platform.libs.epics and its UI plug-ins. Another com.txcorp.soaac.css.dds.pv plug-in provides methods for publishing and subscribing topic-specific variable. Using our prototype, CSS widgets can subscribe or publish to variables as DDS topics using the familiar URI syntax such as "dds://topicname".

Although our prototype plug-ins demonstrate that DDS can be integrated into the CSS environment seamlessly, there are several limitations in our current design that we are working to improve:

1. Currently, we model the DDS topic structure closely after the EPICS PV data structure. We are working to relax this restriction so that a widget can subscribe to arbitrary variable within a DDS topic, e.g., "dds://device/part#control_point".
2. For a CCS application to subscribe to certain topics, their type-specific implementation must be generated and compiled into a .jar file, which then must be loaded into the application. We are working on adding support for a CSS application to read in topic structure and QoS definitions as XML files during runtime. This enhancement will eliminate the needs to compile and load application specific code into the CSS library.

SUPPORT PYTHON-DDS MAPPING (PYDDS)

Python language is a very versatile dynamic, object-oriented language that has gained great popularity among sceintists. Many scientists have expressed interests in interacting with DDS-based systems directly from within their Python codes. We have previously experimented with using DDS in Python. As illustrated in Fig. 3, our previous implementation of Python DDS support wraps all the topic-specific C/C++ codes generated by DDS tools into Python using SWIG or Boost.Python.

Although this approach serves the purpose, it is not compatible with Python's dynamic programming style. In particular, extra-steps outside of Python are required to generate the topic-specifc python mapping. Furthermore, when topic structures change, these wrappers have to be regenerated and repackaged in order for the actual Python application to use them. All these limitations interrupt the natural workflow of programmers and are not compatible the the dynamic language nature of Python.

In order to address these limitations, we are developing a new Python DDS (pyDDS) implementation. As shown in Fig. 4, we have moved the generation of topic-specific codes into Python. These topic-specific codes in-turn interace with generic DDS services. Using this approach, applications can source in topic structure definitions at

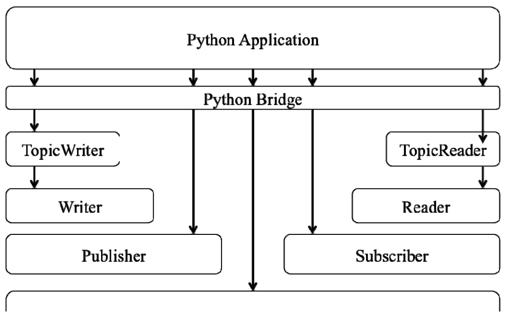

Figure 3: A straightforward Python DDS wraps generated topic-specific codes.

run time and generate the topic-specific readers and writers on the fly as pure Python classes. By eliminating the need to generate and load topic-specific wrappings in separate steps, the new pyDDS library is more compatible with dynamic programming nature of Python and also more easily accepted by Python developers.

The following listings demonstrate how to interact with a DDS data topic from within Python:

```
# First thing to do to use pydds
import pydds;
```

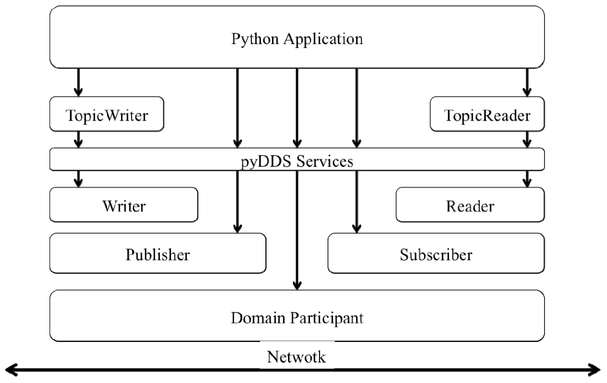

Figure 4: The new pyDDS implementation takes advantage of Python's dynamic language features.

To Join a Data Domain:

```
# Uers defines the dataspace runtime
# and pass it in to various other
# operations that need it.
 myDataspace =
   pydds.connect_dataspace
    ("Domain name", "Partition name")
```

To Manipulate QoS Policies:

```
myQoS = pydds.create_qos()
myQoS.set_reliable (3000000)
myQoS.set_transient()
myQoS.set_keep_last (3)
```

To Create Topic Reders/Writers:

```
# Creating/Finding a topic in global
# data space. Last argument specifies
# the URI to the topic definitions
helloTopic = pydds.createTopic
 ("TopicName",
  myDataspace,
  myQoS,
  file:///HelloWorld.idl#HelloTopic)

# Now create reader/writer objects
helloReader =
 helloTopic.create_reader(readerQoS)
helloWriter =
 helloTopic.create_writer(writerQoS)
```

To Write and Read Data Samples:

```
# Creating a sample
helloSample =
 helloTopic.create_sample
  (message = "John Smith",
   repeat = 3)

# Publishing the sample
status =
 helloWriter.write (helloSample)

# Simple read/take
[samples, infos] = helloReader.read()
sys.stdout write(samples[0].message)
```

We plan to model listener-based callback data read interface after the Twist or Trellis libraries. Furthermore, we would also like to take advantage of the emerging eXtensible Type Standard which is being ratified by the OMG [7,8].

CONCLUDING REMARKS

This paper describes how a dual-service-bus SOA provides an ideal development environment that promotes modulization of high-level application components and facilitates dynamic composition of high-level applications. In the context of the high-level application environment, it means flexibility in selecting and connecting the most appropriate modeling algorithms and programs. To support such use case, we are bringing the next generation data-centric publish-subscribe middleware called DDS to this environment by investigating various usage scenarios and programming patterns of DDS in ACS. Furthermore, we are developing tools and libraries to enable ACS developers to leverage the new technologies. We developed a DDS performance test suite and used it to benchmark and evaluate various open source and commercial DDS implementations. We also implemented a set of example Web-based high-level application on top of open source DDS implementations.

Other than demonstrating how a ACS can utilizes DDS middleware technologies, we are also developing tools to help facilitate the adoption of DDS. We implemented a set of prototype plug-ins for Control System Studio to monitor and emit DDS data stream. We also developed a prototype Python mapping for DDS to enable Python applications to participating in DDS data exchange. We are in the process of hardening these prototypes and enhance their robustness and usability.

ACKNOWLEDGMENTS

The authors would like to thank Nikolay Malitsky of Brookhaven National Lab for his help in formulating the middle-layer server examples. Similarly, we wish to thank Jim Kolwalkoski, Marc Paterno, Kurt Biery, and Erik Gottschalk of Fermilab for discussing the needs and requirements of their projects.

REFERENCES

[1] L. Dalesio *et al.*, "The Experimental Physics and Industrial Control System Architecture," ICALEPCS'93, Berlin, Germany, October 1993, http://www.aps.anl.gov/epics/
[2] Dirk Krafzig and Karl Banke and Dirk Slama. Enterprise SOA – Service-Oriented Architecture Best Practices. Prentice Hall, 2005.
[3] Eric Newcomer and Greg Lomow. "Understanding SOA with Web Services," Addison Wesley, New Jersey, 2005.
[4] OMG, "Data Distribution Service for Real-time Systems, Version 1.2," formal/07-01-01, http://www.omg.org/cgi-bin/doc?formal/07-01-01
[5] N. Malitsky *et. al.*, "Prototype of a DDS-based High-Level Accelerator Application Environment," ICALPCS09, Kobe, Japan, Oct 2009.
[6] Desy, SNS, BNL, "Control System Studio," http:// http://css.desy.de/content/index_eng.html
[7] OMG, "Extensible and Dynamic Topic Types for DDS," http://www.omg.org/spec/DDS-XTypes
[8] N. Malitsky *et. al.*, "DDS XType-based Machine Server," ICALEPCS'11, Grenoble, France, Oct. 2011.