The Global Leader in DDS

July 2011

System Architecture for Robust Integration

An introduction to Common Integration Patterns

Rick Warren - Director of Technology Solutions
Bill Antypas - Chief Consulting Engineer
Gordon Hunt - Chief Applications Engineer

www.rti.com

US HEADQUARTERS
Real-Time Innovations, Inc.
385 Moffett Park Drive
Sunnyvale, CA 94089
Tel: (408) 990-7400
www.rti.com
info@rti.com

Architecture Overview

Enterprises increasingly need to develop distributed systems in an agile manner, with minimal perturbation to end users and at lower costs. An important consideration in realizing these benefits is to break down expensive system stovepipes and to leverage common services and capabilities. Only a competitive marketplace based on interoperable standards with transparent governance can provide the agility, reuse, and cost control necessary. A vendor-specific or non-interoperable infrastructure cannot, regardless of whether the customer has access to its source code.

Interoperability requires that distributed services share a common understanding of the data on which they operate—the data's structure as well as its Qualities of Service (QoS, *i.e.* how it changes and how it's distributed). Using an agreed-upon message format (sometimes called an Interface Control Document, or ICD) is not sufficient, because if the relationship of message to data is not explicit, the integration infrastructure cannot govern the data. Instead, every application must take the job on itself—in a redundant, application-specific way. Applications become more brittle and harder to develop, and without a robust integration infrastructure, systems become closed stovepipes.

System architectures can be classified based on the level to which they govern their data.

- An **application-centric architecture** provides little or no governance. It is so called because each application is a world unto itself. Its state is implicit and not exposed. The operations that act on that state are specific to that application. As a result, applications cannot interoperate unless they are tightly coupled to each other. Each application must understand the others, so it is difficult to change them independently. Such architectures are therefore typically appropriate for monolithic distributed "systems" (really just single applications) under the tight control of an authority capable of evolving them all at once.

  Example implementation technology: CORBA

  Example scenario: Each object defines a unique interface, to which all of its clients are tightly coupled.

- A **data-centric architecture** provides strong governance over data. It is so called because it organizes the interactions among applications in terms of stateful data rather than in terms of operations to be performed. Data structure and QoS are explicit and discoverable. The operations that act on that state are uniform[1]. As a result, the integration infrastructure is able to enforce the data structure and QoS contracts on behalf of the applications, such that applications are not permitted to communicate malformed data or to change data in inappropriate ways. Applications are easier to develop, less dependent on each other, and more fault-tolerant. Such architectures are therefore

[1] These operations typically follow a pattern called "CRUD"—Create, Read, Update, and Delete—because most supporting technologies have parallels to these operations. In SQL [4], the operations are INSERT, SELECT, UPDATE, and DELETE. In HTTP, they are POST, GET, PUT, and DELETE. In DDS, they are WRITE, READ, DISPOSE, and UNREGISTER.

appropriate for distributed systems of any size, including systems of systems and those involving multiple teams.

Example implementation technologies: SQL databases [4] (data at rest only), RESTful web services [6] (data at rest only), and OMG DDS [2] (data in motion).

Example scenario: Two applications connect to a relational database. One changes a row in the database, identified by its key, and the other subsequently queries the updated value.

- In between these two, a **message-centric architecture** governs the mechanism of communication (*i.e.* the flow of messages) but not the state data to which that communication refers. State and/or operations may be exposed using application-specific message sets—for example, an ICD describing that a message with contents *X* updates a certain state that should be been established by a previous message with contents *Y*. The integration infrastructure is able to govern the flow of messages, ensuring that they flow where they are intended and that their contents are well formed; applications are therefore somewhat decoupled from one another. However, the infrastructure cannot determine whether messages have the appropriate impacts on system state, or govern the distribution of that state, or ensure that applications operate based on up-to-date and correct views of the broader system. As a result, integrations are typically point-to-point among constituent subsystems and tend to be brittle. Applications are responsible for maintaining their own state, which can lead to challenges if they fail and restart or need to be redeployed elsewhere on the network. Such architectures are appropriate for small to medium-sized distributed systems that have a limited number of known constituent subsystems and that can be upgraded all at once if necessary.

  Example implementation technologies: AMQP [1], Java Message Service (JMS) [3], WS-Notification [5]

  Example scenario: One application may expose a notification “mouse clicked” and another exposes an operation “create widget”. Both of these operations are expressed in terms of JMS messages. An Enterprise Service Bus (ESB) sits between them and sends a “create widget” message every time it receives a “mouse clicked” message.

Data-centric architecture is most broadly applicable, because it provides strong governance over the integration. However, the simpler the integration to be performed, and the more control that the integrating organization has over the constituent subsystems, the less serious the ramifications of a lack of governance. Consequently, for systems of modest complexity under a single authority, other approaches may yield acceptable results.

Integration Principles

System integrators have found that robust Open-Architecture integration requires interoperability at three levels:

1. **Byte Level**. The system elements must be able to exchange unstructured data. (Technologies that support application-centric architecture address interoperability up to this level.)
2. **Message Level**. The system elements must share a common "syntax" for their communication. (Technologies that support message-centric architecture address interoperability up to this level.)
3. **Data Level**. The system elements must relate the messages they exchange to explicit data objects that change in well-defined ways—they must share a common set of semantics. (Technologies that support data-centric architecture address interoperability up to this level.)

Data-centric architecture relates messages to data according to the following principles:

1. **The structure, changes, and motion of stateful data must be well defined.** "State" consists of the information that an application needs in order to interpret messages correctly. For example, suppose there is an announcement, "the score is four to three". What game is being played? Who are the players? Which one of them has four points and which three? The answers to these questions comprise the state that is necessary to understand the message. This is a specialization of the Service-Oriented Architecture principle of *standardized service contracts*; see [10].
2. **The contracts governing the structure, changes, and motion of stateful data must be discoverable.** This is the same as the Service-Oriented Architecture principle of *discoverable service contracts*; see [8].
3. **State must be managed by the infrastructure, and applications must be stateless.** This is the same as the Service-Oriented Architecture principle of *stateless applications* [9] as captured in the *state repository* pattern [7].
4. **State must be accessed and manipulated by a set of uniform operations.** Operations express attempts to change the state. This principle is shared with the REST Architecture; see [6].

The above principles allow applications and systems to interoperate at the level of an explicit data model. When a system's data model is explicit, it can be used at run time by applications to make dynamic decisions based upon the content of the data, increasing capability and operational agility. Further, interactions can be governed by infrastructure, reducing per-application costs and inter-application coupling. On the other hand, if the data model is implicit, decisions must be pre-determined, established, and enforced by static code prior to execution, decreasing agility and increasing vendor lock-in.

Summary

In traditional IT systems, a modest number of applications were developed by related teams within the same organization and managed by a single authority. These systems had short life cycles and could be evolved all at once if necessary. Consequently, message-centric approaches were sufficient. However, today's enterprises are increasingly being asked to address systems of systems that must be long-lived and incorporate subsystems that were not known *a priori* and for which "big bang" upgrades are impossible. In such systems, appropriate dissemination and synchronization of state are critical, and a data-centric approach can significantly improve agility and drive down total cost of ownership.

Appendix: Technology Evaluation

This section describes several technologies in terms of the architectural principles outlined in this document.

Principle	DDS	AMQP	Relational Database	WS-Notification
Interoperable Transport Protocol	**Yes** (DDS-RTPS/UDP)	**Yes** (TCP)	**No**	**Yes** (HTTP)
Interoperable Messaging Protocol	**Yes** (DDS-RTPS)	**Yes**	**No**	**Yes** (SOAP)
Standardized Contracts				
– Formal Type Definition Language	**Yes** (OMG IDL or W3C XSD)	**Yes** (AMQP-specific)	**Yes** (SQL)	**Yes** (W3C XSD)
– Operations	**Yes** (Uniform operations; portable API [2])	**Partial** (Formal message syntax; non-standard API)	**Yes** (Uniform operations; portable API [4])	**Partial** (Formal message syntax; non-standard API)

Principle	DDS	AMQP	Relational Database	WS-Notification
– Data Structure	**Yes**	**Partial** (Optional message format definitions, but unspecified association between message flow and format and between message and data)	**Yes**	**Partial** (Standard message formats, but messages have undefined relationship to data)
– Data Motion	**Yes**	**No**	**No**	**No**
– Data Changes	**Yes**	**No**	**No**	**No**
– Run-Time Contract Enforcement	**Yes**	**No**	**Yes**	**No**
State Repository, Stateless Applications	**Yes**	**No**	**Yes**	**No**
Discoverable Contracts	**Yes**	**No**	**Yes**	**Yes**

Note that architecture abstractions and technology implementations are related but independent. A system's architecture may be at a certain level while the technologies that implement it are at a lower level. In this case, the system builders will have to "make up the difference" themselves, leading to increased cost and risk. Consider the implications for interoperability, reliability, and system maintenance of such an approach vs. one based on more capable technologies.

Nevertheless, in systems of systems, it may be necessary to integrate a subsystem that has a given architecture (*e.g.* data-centric) with another subsystem that has a different architecture (*e.g.* message-centric). This can be done by means of a mediation service between the subsystems.

- As messages flow from the message-centric subsystem to the data-centric one, the mediation service collapses and correlates messages with one another to generate changes to the data objects to which they pertain.
- As data objects change in the data-centric subsystem, the mediation service generates the appropriate messages describing those changes in the message-centric subsystem.
- As messages flow from the message-centric subsystem to the data-centric one, the mediation service collapses and correlates messages with one another to generate changes to the data objects to which they pertain.
- As data objects change in the data-centric subsystem, the mediation service generates the appropriate messages describing those changes in the message-centric subsystem.

Resources

The following resources are referenced in this document.

Specifications

1. *Advanced Message Queuing Protocol* (AMQP), version 1-0r0. AMQP Working Group. http://www.amqp.org/confluence/display/AMQP/AMQP+Specification.
2. *Data Distribution Service* (DDS), version 1.2. Object Management Group (OMG), document number formal/2007-01-01. http://www.omg.org/spec/DDS/1.2/.
3. *Java Message Service* (JMS), version 1.1. Java Community Process (JCP), Java Specification Request (JSR) 914. http://www.jcp.org/en/jsr/detail?id=914.
4. *Structured Query Language* (SQL). International Organization for Standardization (ISO), document number ISO/IEC 9075-14:2008. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45499.
5. *Web Services Notification* (WSN), version 1.3. OASIS. http://www.oasis-open.org/committees/wsn/.

Additional Resources

6. REST architecture: http://en.wikipedia.org/wiki/Representational_State_Transfer
7. SOA Pattern: *State Repository*. http://soapatterns.org/state_repository.php
8. SOA Principle: *Service Discoverability*. http://www.soaprinciples.com/service_discoverability.php
9. SOA Principle: *Service Statelessness*. http://www.soaprinciples.com/service_statelessness.php
10. SOA Principle: *Standardized Service Contract*. http://www.soaprinciples.com/standardized_service_contract.php

