Data-Oriented Architecture:

A Loosely-Coupled Real-Time SOA

Rajive Joshi, Ph.D.

rajive.joshi@rti.com
408-200-4754

Real-Time Innovations, Inc.

3975 Freedom Circle
Santa Clara, CA 94054

2007 August

"Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming. (See Brooks p.102)."

- Rob Pike, Notes on Programming in C, 1989

Abstract

As more devices and systems get woven into the fabric of our networked world, the scale and the complexity of integration is growing at a rapid pace. Our existing methodologies and training for system software design, rooted in principles of object-oriented design, that worked superbly for small scale systems begin to break down as we discover operational limits which requires frequent and unintended redesigns in programs year over year. Fundamentally, object-oriented thinking leads us to think in terms of tightly-coupled interactions that include strong state assumptions. Large scale distributed systems are often a mix of subsystems created by independent parties, often using different middleware technologies, with misaligned interfaces. Integrating such sub-systems using object-oriented thinking poses some fundamental challenges:

(1) it is brittle to incremental and independent development, where interfaces can change without notice;

(2) there is often an "impedance mis-match" between sub-systems in the quantity and the quality of information that must be exchanged between the two sides;

(3) there is a real need to dynamically adapt in real-time to network topology reconfigurations and failures;

(4) scalability, performance, and up-time cannot always be compromised in this dynamic environment .

A different paradigm is needed in order to address these new challenges in a systematic manner.

As the scale of the integration and complexity grows, the only unifying common denominators between disparate sub-systems (generally numbering more than two) are:

(1) the data they produce and consume;

(2) the services they use and offer.

In order to scale, system software architecture must be organized around a common "shared information model" that spans multiple systems. This *leads us to the principle of "data-oriented" design: expose the data and hide the code.*

In this paper, we will discuss the principles of data-oriented thinking, and discuss why it offers an appropriate paradigm to address large scale system integration.

We discuss the critical role played by the middleware infrastructure in applying data-oriented design, and describe a generic data-oriented integration architecture based on the data distribution service (DDS) middleware standard. We analyze popular architectural styles including data flow architecture, event driven architecture, and service oriented architecture from this perspective and establish that they can be viewed as specializations of the generic data-oriented architecture.

Finally we illustrate how the data-oriented integration architecture was used to rapidly develop a working demonstration of a real-time package tracking system-of-systems, in a short time frame. The information model is described once. The tool-chain is used to transform and manipulate the shared data model across disparate implementation technologies.

Table of Contents

1 Introduction

The growing popularity of cheap and widespread data collection “edge” devices and the easy access to communication networks (both wired and wireless) is weaving in more devices and systems into the fabric of our daily lives. As computation and storage costs continue to drop faster than network costs, the trend is to move data and computation locally, using data distribution technology to move data between the nodes as and when needed. As a result, the quantity of data, the scale of its distribution and the complexity integration is growing at a rapid pace.

The demands on the next generation of distributed systems and systems-of-systems include being able to support dynamically changing environments and configurations, being constantly available, and being instantly responsive, while integrating data across many platforms and disparate systems.

How does one systematically approach the design of such systems and systems-of-systems? What are the common unifying traits that can be exploited by architects to build systems that can integrate with other independently systems, and yet preserve the flexibility to evolve incrementally? How does one build systems that can be self aware and self-healing, and dynamically adapt to changes in their environment? Can this be done on the scale of the Internet, and yet be optimized for the best performance that can be supported by the underlying hardware and platform infrastructure? Can this be done without magnifying the ongoing operational and administrative costs?

These and related topics are the subject of this paper.

1.1 A Real-Time System-of-Systems Scenario

Let us consider the air traffic control example of Figure 1. It involves a variety of disparate systems that must seamless operate as a whole.

On the *“edge”* is a real-time avionics system inside the aircraft, which may communicate with a control-tower. The data flowing in this system is typically at high rates, and time-critical. Violating timing constraints could result in the failure of the aircraft or jeopardize life or safety.

The control tower is yet another independent real-time system, monitoring various aircraft in the region, coordinating their traffic flow, and generating alarms to highlight unusual conditions. The data flowing in this system is still time-sensitive for proper local and wide-are system operation, albeit it may a bit more tolerant of occasional delays.

In our simplified example, the control tower communicates with the airport *Enterprise* Information System (EIS). The enterprise information system keeps track of historical information, flight status, and so on and may communicate with multiple control towers and other enterprise information systems. The enterprise system is not in the time-critical path, and therefore can be much more tolerant of delays on arrival of data; the data rates are also much lower compared to those in a control-tower or an aircraft; however, the volume of data is far greater because of the shear magnitude of the scaling factors involved (numbers of airplanes, numbers of passengers, historical data, and so on). This enterprise system is responsible for synthesizing a composite "*dashboard*" view, such as passenger information, flight arrival and departure status, and overall operational health of the airport, from the real-time data originating from the "edge" systems (aircraft).

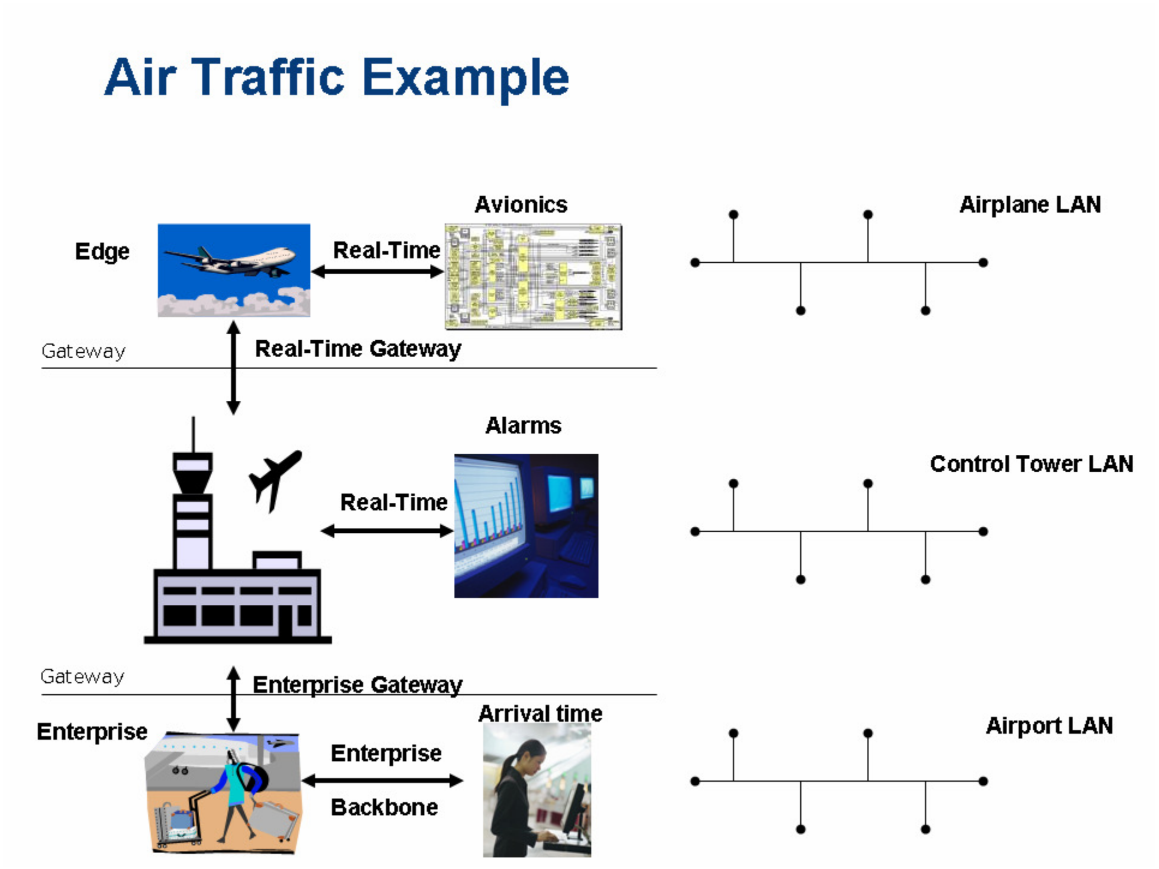


Figure 1 The next generation of distributed systems is dynamic in nature, and must meet the demands of constant availability, instant responsiveness, reliability, safety, and integrity. They require integration of many platforms, systems, and their data.

This "*real-time edge-to-enterprise integration*" scenario is representative of many other application domains including the US Department of Defense's (DoD) vision of net-centric operations and the global-information-grid (GiG), new initiatives on intelligent transportation systems (ITS), telematics, ground traffic control, homeland security, medical systems, instrumentation, industrial automation, supply chain, simulation, unmanned vehicles, robotics, and systems engineering, to name a few. Indeed, parallels to this "multi-tiered" edge to enterprise example can be easily seen around us---for example, instead of "airplane, control tower, airport" it could be "cell phone, cell tower, cell base station", or "power generation systems, power grid centers, power distribution centers".

In business, many forces, including the rise of outsourcing, the need for business agility, and the trend towards “on-demand” business, are leading to scenarios where independently developed systems must be quickly adapted and re-adapted to realize new business capabilities.

The new emerging class of distributed applications is an integrated *“system-of-systems” (SoS)*, that brings together multiple independent systems to provide new capabilities. In general, the different classes of systems form a continuum; however for the purposes of this paper we broadly classify systems into three categories, below.

1. *Edge* systems. These are systems that touch the “real-world” to perform various functions, including sensing, and actuation. Often, these systems must operate under “real-time” constraints---i.e. their correct operation depends on meeting critical design requirements and timing constraints for various functions. The timescale is at machine-level, sometimes in the order of microsecond resolution. Examples of edge systems include instrumentation, robots, radars, communication equipment etc.

2. *Enterprise* systems. These are the “information technology” (IT) systems that traditionally include functions such as high-level user interaction, decision support, storage and retrieval of historical data. Often these systems provide the executive-level operational “dashboards” that integrate data from edge systems. Usually, these systems have “soft” real-time constraints, and the timescale is at a “human” response level in the order of seconds, minutes, hours, days, etc. Examples of enterprise systems include application servers, packaged applications, web-servers etc.

3. *Systems-of-Systems (SoS).* These are distributed systems composed of many Edge, and/or Enterprise systems, including other SoS. Successful examples of SoS include the “World-Wide-Web (WWW)”, and “electronic mail”, that run on the public Internet. The US DoD’s GiG is envisioned as a SoS that integrates various DoD assets over the DoD’s private internet.

SoS are loosely coupled, with many independent entry points, under independent control domains, that effectively inter-operate to realize multiple objectives.

1.2 Key Challenges

SoS must effectively deal with various issues, including (1) crossing trust boundaries, where each system is controlled and managed independently, and involves social, political and business considerations; (2) managing quantitative and qualitative differences in the data exchange and performance; for example an “edge” system often carries time-critical data at high rates, some of which must eventually trickle into an “enterprise” system; (3) operating across disparate technology stacks, design paradigms, and life-cycles of the different systems.

Next, we examine some of the key technical challenges in building successful SoS, and leave the non-technical aspects for discussion elsewhere.

1. *Incremental and Independent Development* arising from the fact that systems are generally developed and evolved independently.

2. *Impedance Mismatch* arising from the non-functional differences in the information exchange between systems – both in the quantity and the quality of the data exchange.

3. *Dynamic Real-Time Adaptation* arising from the fact that the environments can change dynamically, and it is not practical to have a centralized administrator or coordinator.

4. *Scalability and Performance*, arising from the need to support larger SoS as more resources are introduced, with minimal overhead.

1.2.1 Incremental and Independent Development

Each system can be under an independent domain of control---sometimes both from an operational as well as management perspective. Different systems may be developed independently. In the “systems engineering” world, it is common development practice to switch a system or a sub-system transparently from a “simulated” version to an independently developed “real” version and vice-versa. Furthermore, deployed systems continuously undergo incremental evolution and development as capabilities are added or extended.

By their very nature, the systems under consideration are *loosely-coupled*---minimal assumptions can be made about the interface between two interacting sub-systems. The integration should be robust to independent changes in either side of an interface. Ideally, changes in one side should not force changes on the other side. This implies that the interface should contain only the invariants that describe the interaction between the two-systems. Since, behavior is implemented by each independent system; the interface between them must *not include any system specific state or behavior* (Figure 2). The remaining invariant is the *information exchange* between the two systems.

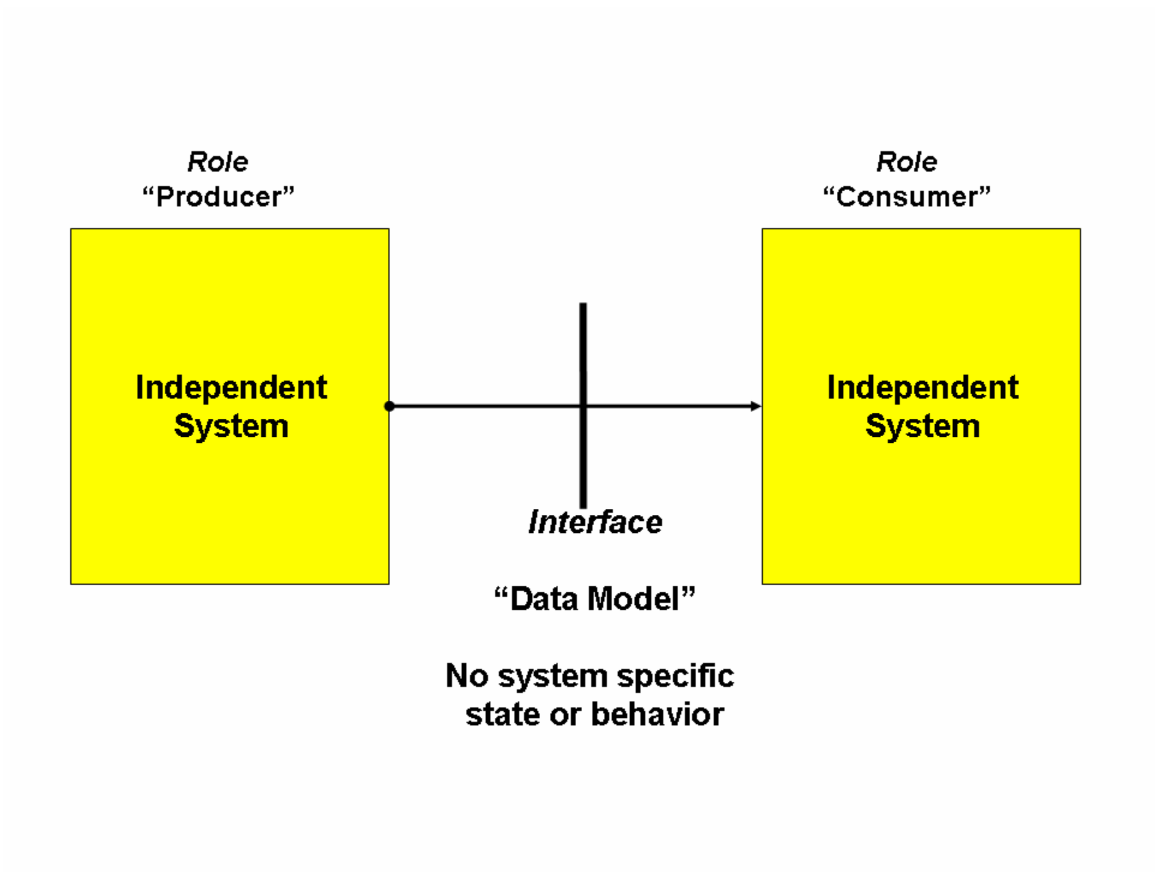


Figure 2 The interface between "loosely-coupled" independent systems should not include any system-specific behavior or state. It can include the "data model" of the information exchanged between the systems, and the role played by a system.

An *information exchange* can be described in terms of

1. the information exchange "data model"
2. the roles of "producer" and "consumer" participating in the information exchange

Thus, when dealing with loosely coupled systems, a system’s interface can be described in terms of the “data model” and the “role” (producer or consumer) the system plays in the information exchange. Additional assumptions can break the loose coupling.

1.2.2 Impedance Mismatch

The systems on either side of an interface (Figure 2) may differ in the qualitative aspects of their behavior, including differences in data volumes, rates, timing constraints, and so on.

We use the term “*impedance mismatch*” to connote all the *non-functional* differences in the information exchange between two systems. Figure 3 illustrates the impedance mismatch between edge and enterprise systems in the current state of affairs.

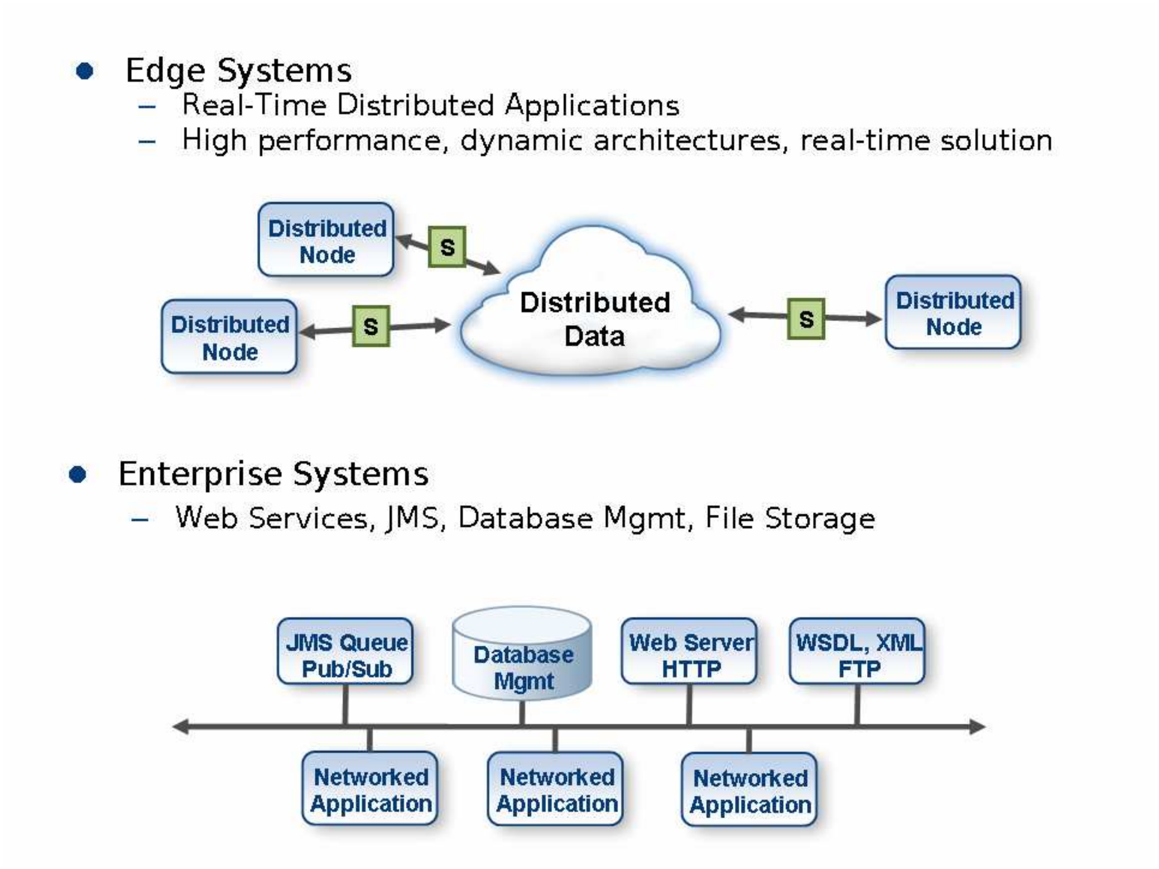


Figure 3 Impedance mismatch between today's edge and enterprise systems arises from a number of factors, including differences in data rates, timing constraints, life-cycle of data, persistency, and the differing technology stacks.

The non-functional aspects of a system's interface (data model and role, Figure 2) can be conceptually captured as a *quality-of-service (QoS)* aspect associated with the system's role. A producer may "offer" some QoS, while a consumer may "request" some QoS for an interface. A producer and consumer can participate in information exchange using that interface *if-and-only-if* their QoS are "*compatible*". This approach allows us to conceptually model and deal with the impedance mismatch between independent systems.

1.2.3 Dynamic Real-Time Adaptation

The independently managed systems can appear and disappear asynchronously, as they are started, shutdown, rebooted, or reconfigured. The environment can change dynamically, causing systems to react differently.

Physical communication links between systems may go down or may be unreliable---such failures may be indistinguishable from system failures at the other end of the communication link [Waldo 1994].

In general, it is not possible or not practical to have a centralized administrator or coordinator of the various systems, especially at the granularity of asynchronous changes that may occur dynamically. Thus, each system must detect and react to dynamic changes as they occur. The responsibility of detecting dynamic changes in connectivity is best delegated to the *information exchange infrastructure* provided by the computing environment

The information exchange infrastructure typically includes the network transport hardware, the computing hardware, the operating system, and the communications middleware, on top of which an application runs.

Ideally, the information exchange infrastructure would be "*self-aware"* in the sense of being able to detect and inform the systems when changes occur in their connectivity with other systems.

1.2.4 Scalability and Performance

Scalability refers to the ability to handle proportionally more load, as more resources are added. Scalability of the "information-exchange" infrastructure thus refers to the ability to take advantage of underlying hardware and networking resources, and the ability to support larger SoS as more physical resources are added. From an component developer's perspective, this translates to the ability of the information exchange infrastructure to naturally scale from systems comprising of one-producer and one-consumer pair, to one-to-many, many-to-one, and many-to-many producers and consumers.. A related notion is that of "administrative scalability" of the infrastructure---how the infrastructure administrative load increases as the number of components increases. Ideally, this should be at a minimum.

Performance refers to the ability to support information exchange with minimal overhead. Performance considerations typically include latency, jitter, throughput, and processor loading.

Scalability and performance are practical concerns that determine the suitability and the justification of a SoS for business objectives in the first place. They depend heavily on the implementation of the underlying information exchange infrastructure.

1.3 Solution Dimensions

The key technical challenges in building SoS can be tackled among the following dimensions.

1. *Design Paradigm.* Traditional "object-oriented" design works well for *"tightly-coupled*" systems where strong state assumptions about the interaction are acceptable. Indeed it is possible to avoid state assumptions with object-orientation as well; however that is not the general practice or guidance, and state assumptions invariably do trickle into the interface specification. For loosely-coupled systems, a new "data-oriented" design paradigm (see the Section "The Data-Oriented Design Paradigm") provides a more suitable framework to address the challenges of (a) incremental and independent development; (b) impedance mismatch.

2. *Middleware Infrastructure*. It refers to the information exchange infrastructure. As noted during the discussion the technical challenges, the middleware infrastructure that take on important responsibilities, and provide key capabilities that can ease and facilitate the construction of SoS and distributed systems in general. The middleware infrastructure is the place to address the challenges of (a) dynamic real-time adaptation; and (b) scalability and performance. It can also facilitate integration by directly supporting a design paradigm for loosely-coupled systems, and thus aid in addressing (c) incremental and independent development; (b) impedance mismatch. We discuss this in the Section "The Role of the Middleware Infrastructure".

3. *Application Architecture*. It refers to the overall architecture of a SoS to realize business objectives. It encompasses the overall interaction patterns, common data models, QoS, and application specific semantics. Several approaches are prevalent for the application architecture, including client-server architecture (CSA), event-driven architecture (EDA), and so on. The application architecture is anchored to the design paradigm, and utilizes the underlying middleware infrastructure. Application architecture is the place to address the challenges of (a) impedance mismatch; (b) dynamic real-time

adaptation at an integrated SoS level. We discuss this in the Section ”Application Architecture: Applying the Data-Oriented Paradigm”, and finally illustrate a concrete SoS integration example that ties all the concepts together.

The rest of this paper discusses each of these solution dimensions in depth.

2 The Data-Oriented Design Paradigm

System design, in general, may be viewed as a collection of interconnected *components*. Depending on the context and granularity of scale, a component may be, for example, an entire system (in a SoS), or an application, or a process, or a library module. Let us examine the underlying design principles that can provide us the theoretical foundation for dealing with the challenges introduced in Section “Key Challenges”.

2.1 The Practice of Distributed System Design

When we examine the practice of system design, we can identify two lines primary schools of thought, distinguished along the lines of “tight-coupling” vs. “loose-coupling” of components. These have been applied to the design of both local and distributed processing systems.

2.1.1.1 Tightly-coupled system design is promoted by a familiar paradigm

Tight-coupling refers to making strong assumptions about the interface of interconnected components. Changes in one component’s interface typically have a significant impact on the components that interact with it.

Our existing methodologies and training for system software design, rooted in principles of object-oriented design, lead to tightly-coupled interactions. In the object-oriented approach, it is common practice to make strong assumptions about the interaction state in the interface; the paradigm encourages the notion of “distributed state” across the interacting objects. This works superbly on the small scale for “local processing” where the components live in a share a *common “logical” address space*, the communication latency is negligible, memory can be accessed by shared pointers or references, and there is no “real” indeterminism about how much of a computation completed when failures occur or when operations are invoked---however these assumptions are not valid for distributed system where components do not share a single logical address space [Waldo 1994].

Since the object-oriented programming paradigm is so widely popular and successful for local processing, it is only natural that we apply it for distributed processing as well. The idea is to wash away the differences between local and distributed processing, and treat the system as a logical whole around a logical distributed shared memory model. The defining mantra for this approach is captured by Sun Microsystems tag line: “The network is the computer”.

This path has lead to various endeavors, including the work on distributed operating systems, distributed shared memory, remote-procedure calls (or RPC), common request-broker architecture (CORBA), enterprise java beans (EJB), simple-object access protocol (SOAP), and approaches for clustering and load-balancing to scale the performance of the “logically” centralized computing model across distributed nodes. It is now well established in the literature that it is a mistake to wash away the differences between remote and local objects [Waldo 1994], and at best this approach only works for small and tightly managed environments.

Indeed, object-oriented design is an appropriate paradigm for components that are intrinsically tightly coupled, and allows one to take advantage of the performance optimizations afforded by direct memory access, lower latencies, simple failure modes, and centrally managed concurrency and resources.

Object-oriented system design has been used with success in tightly controlled and managed distributed environments. However, tightly-coupled solutions are more difficult to modify since changes made in one place cause changes to be made somewhere else. At the best they require a re-test of the entire system or system-of-systems to make certain nothing was broken. They do not robustly scale up when the assumptions of centralized control and management no longer hold [Waldo 1994], as is the case with the new generation of distributed systems (eg SoS).

2.1.2 Loosely-coupled system design requires a paradigm shift

Our existing object-oriented methodologies and training for system software design that worked superbly for local or centralized processing begin to break down for distributed or decentralized systems.

When components live in different address spaces, the differences in communication latency and memory access among components become significant. Working with components in multiple address spaces also introduces "true" indeterminism in failure modes [Waldo 1994], and in the invocation of concurrent operations [Lamport 1978]. Such systems must deal with partial failures, arising from failure of independent components and/or communication links; in general the failure of a component is indistinguishable from the failure of its connecting communication links. In such systems, there is no single point of resource allocation, synchronization, or failure recovery. Unlike local process, a distributed system may simply not be in a consistent state after a failure.

The "fallacies of distributed computing" [Van Den Hoogen 2004], summarized below, capture the key assumptions that break down (but still are often made by architects) when building distributed systems.

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

A different paradigm is needed to build distributed systems, which by nature, are "loosely-coupled". This has been clearly noted in the literature [Waldo 1994].

> *"Programming a distributed application will require thinking about the problem in a different way than before it was thought about when the solution was a non-distributed application.*
> *:*
> *One consequence of the view espoused here is that it is a mistake to attempt to construct a system that is "objects all the way down" if one understands the goal as a distributed system constructed of the same kind of objects all the way down."*
>
> \- *Waldo, Wyant, Wollrath, Kendall, 1994*

An appropriate paradigm for building “loosely-coupled” systems can be found in the principles of “*data-oriented programming”* [Kuznetsov], and discussed in the following Section “
Data-Oriented Programming”. It is based on the observation that the “data model” is the only invariant (if any) in a loosely coupled system (Figure 2), and should be exposed as a first-class citizen----in other words “data” is primary, and the operations on the data are secondary! Since a common “logical” address space cannot be assumed, the information exchange between components is based on a *message passing* interaction paradigm.

When applied to distributed systems, the defining mantra for a loosely-coupled approach may be captured by: “*The computer is the network*”, which acknowledges the fact that distributed computing must indeed be cognizant of the network and the potential for communication failures. The world-wide-web and electronic-mail are good examples. Other examples are seen in the availability of various message-oriented middleware (MOMs), Java Message Service (JMS), data-centric publish-subscribe middleware (DDS) to support the implementation of loosely-coupled distributed systems.

Loosely-coupled system design approaches can be applied to the design of local or centralized applications as well, to achieve a more scalable architecture, modularize complexity, promote a more robust design, support outsourcing activities, merger and acquisition activities, integrate third-party components, and so on. For centralized systems, this requires extra effort and discipline, but can be well worth the rewards. Examples of such usage are increasingly common in the design of application servers (AS), enterprise service bus (ESB) [Richards 2006], Java Business Integration (JBI) [Richards 2006]; although these are not necessarily examples of data-oriented programming.

Thus, loosely-coupled system design approaches work well for both localized centralized processing, and distributed decentralized processing, including SoS. Loosely-coupled design can lead to more robust systems that are amenable to more complex scenarios and interactions.

2.1.3 Reality Check

Figure 4 shows a classification of popular technologies with respect to the coupling assumptions they make about the components involved. Tightly-coupled approaches for distributed systems and loosely-coupled approaches for building more complex centralized applications are quite popular. Our focus is on building loosely-coupled distributed systems, an emerging area of research and technology.

Technology Map	***Tight-coupling***	***Loose-coupling***
Local or Centralized Processing	Application Programs	Application Servers (AS) Enterprise Service Bus (ESB) Java Business Integration (JBI)
Distributed or Decentralized Processing	Remote Procedure Calls (RPC) Common-Object Request Broker Architecture (CORBA) Enterprise Java Beans (EJB) Simple Object Access Protocol (SOAP) Distributed Shared Memory (DSM) Distributed Databases Distributed Operating Systems Clustering Load Balancing	Data Distribution Service (DDS) Messaging-Oriented Middleware (MOM) Java Message Service (JMS) World-Wide Web (WWW) Electronic Mail (e-Mail) Representational State Transfer (REST) US DoD's Global Information Grid (GIG)

Figure 4 Classification of popular technologies for centralized and decentralized processing with respect to coupling assumptions.

2.2 Data-Oriented Programming

The "data-oriented programming" (DOP) [Kuznetsov] paradigm refers to a collection of existing design principles for building interoperable software and integrating disparate systems. It is complementary to "object-oriented programming" (OOP), often practiced using Java or C++. DOP is like OOP in the sense of being a conceptual framework that is above any specific programming language or implementation technologies.

DOP provides a solid foundation for constructing loosely-coupled systems, and can be seen as the theoretical basis for much of the recent work on service-oriented architecture (SOA) and web-services.

2.2.1 Data-Oriented Programming Principles

DOP is based on the following principles, elucidated by Kuznetsov .

1. Expose the data and meta-data
2. Hide the code
3. Separate data and code, or data-handling and application-logic.
4. Generate data-handling code from interfaces

An updated version of Kuznetsov’s DOP principles is discussed below. We add the notion of QoS (see Section “Impedance Mismatch”), not found in Kuznetsov’s original version.

1. **Expose the data and meta-data.** Meta-data refers to the information about the data structure itself, including the name, type, multiplicity, and organization of the fields. DOP exposes the data and meta-data as first-class citizens, and uses them as the primary means of interconnecting heterogeneous systems.

    a. The data is the primary means for hooking up components. This DOP principle is the exact opposite of the “encapsulation” design principle in OOP, which has practical outcome of all data fields being hidden behind accessor methods of an object.

    b. In addition to the data itself, DOP relies on exposing meta-data that includes a *formal machine readable format description (FMRFD),* so that the data is *self-describing.* For example, a FMRFD meta-data may be embedded with the data as in XML's self-describing tags [W3C XML], or may be in ASN.1 BER type fields [Kaliski 1993] specified alongside the data. Examples of FMRFD include WSDL [W3C WSDL], DTD [W3C XML], XML Schema [W3C XMLSchema], XML InfoSet [W3C XMLInfoSet], IDL [OMG 2002], ASN.1 [Kaliski 1993], and other schema languages.

    c. Optionally, the exposed meta-data can include formal quality of service (QoS) attributes that adorn the formal data format description, and capture the non-structural aspects of the data exchange.

   d. We observed in Figure 2, the data model is the primary “invariant” in a network of interconnected components. Changes in the exchanged messages can be tolerated while preserving connectivity as long as the associated *meta-data (FMRFD and QoS)* is updated. In addition, the meta-data can be extended to support evolutionary changes to the data model, without “breaking” existing interactions.

2. **Hide the code.** DOP hides the code, and relies on message passing to model coarse-grain interactions between components, with minimal state assumptions.

   a. DOP hides any code and direct references to code. For example, the meta-data should not contain programming language constructs. As we observed in Figure 2, the interface cannot assume any system-specific state or behavior; thus avoiding code in the interface is critical for loosely coupled systems.

   b. DOP models the interaction between two components as sending a message from one to the other with an *operation code*. Unlike OOP, DOP avoids tight coupling that results when one component invokes an operation on another.

   c. DOP encourages coarse-grained interactions between systems and strives to minimize state assumptions. As noted in Figure 2, state assumptions introduce coupling, which runs counter to the notion of loosely-coupled systems.

   d. A key difference from “remote-procedure call” (RPC) model is that there is *no context* or implied state or coupling across messages. A message is self contained, and includes the entire context. This makes DOP well suited to the needs of loosely-coupled systems.

3. **Separate data and code, or data-handling and application-logic.** DOP requires clean separation of the declarative meta-data and data communication, from the implementation of operations on the data.

   a. Sparation of data handling and application-logic code is necessary for DOP to be effective for loosely-coupled systems. A great anti-pattern for this principle is the traditional C++ header file, with its inline methods, and data members; data structures can be mixed in with data manipulation logic.

   b. The responsibility of data-handling can be delegated to the middleware infrastructure, so that the application logic can focus on the processing the data.

4. **Generate data-handling code from interfaces**. A DOP interface is defined by the meta-data (FMRFD and QoS), and must be expressed formally. DOP explicitly forbids the hand-coding of data parsers, translators or output systems; instead it requires that the data-manipulation code be machine generated from the meta-data in a form suitable for the implementation platform.The meta-data must contain all of the information required to encode and decode the data in a given format.

   a. The meta-data must contain all the QoS needed to capture the deal with the impedance mismatch and the implied “tuning” (see Section “Impedance Mismatch”).

   b. Systems using completely different implementation platforms must be able to generate code for encoding and decoding the data format.

   c. A middleware infrastructure can provide the tools for generating data-handling code specific to the implementation programming language and runtime environment, and mechanisms for tuning the impedance mismatch.

2.2.2 DOP and OOP

Figure 5 summarizes the principles of DOP from an OOP perspective.

Data-Oriented Programming (DOP)	Object-Oriented Programming (OOP)
Expose the data and meta-data	Hide the data (encapsulation)
Hide the code	Expose methods – code
Separate data & code	Intermix data & code
Send only messages	Mobile code
Must agree on data encoding, mapping system, QoS semantics	Must agree on code runtime system
Messages are primary	API / Object model is primary
Meta-data = Data Model/Schema + QoS	
Strict separation of parser, validator, transformer, impedance tuner, and logic	Combined processing, no restrictions
Change ➔ change declarative meta-data	Change ➔ read & change code
Loosely-coupled	Tightly-coupled

Figure 5 Comparison of data-oriented programming (DOP) and object-oriented programming (OOP) principles.

DOP and OOP can be viewed as complementary approaches aimed at solving different concerns. OOP promotes tight-coupling, while DOP promotes loose coupling.

Those familiar with the web-services community debate between Simple Object Access Protocol (SOAP) vs. Representational State Transfer (REST) [Prescod] will notice that fundamentally, the argument is really about choosing between an OOP vs. DOP paradigm for constructing web-services.

In practice, for example on a SoS, OOP might be used for fine grained tightly-coupled application components; whereas DOP may be used to interconnect the loosely coupled components.

2.2.3 DOP and Application Architecture

DOP can be seen as the foundation of several distributed application architecture approaches including service-oriented architecture (SOA) [Wikipedia SOA], event-driven architecture (EDA) [Wikipedia EDA], and contract-first design [Skonnard 2005].

DOP principles 1 and 2 can be construed to provide the definition of a "service contract" used in SOA community. The principles of SOA, namely (1) expose the services; (2) hide the objects, are consistent with DOP. The Web Services Description Language (WSDL) is an appropriate formalism for a DOP interface.

As can be seen from the above discussion, DOP provides the framework to addresses two of the key issues in system design (a) incremental and independent development (Section "Incremental and Independent Development"); (b) impedance mismatch (Section "Impedance Mismatch"). Thus, DOP is well suited to solving integration challenges, in a scalable and agile manner.

We should note that it always possible to tighten up loosely-coupled software; however it is not possible to loosen up tightly coupled software. Thus, loose-coupling and messaging are more general idioms than tight-coupling and object method invocation; the latter can be built on top of the former.

It is important to note that a design paradigm by itself does not result in well-designed systems. Like OOP, DOP can also be mis-used and abused. DOP provides the principles and guidance for building loosely-coupled systems. However, design is fundamentally a human activity; paradigms and tools can only facilitate the process.

2.3 An Example

For illustration purposes, let us consider a very simple example from both an OOP and a DOP viewpoint.

Let us consider the simple task of "registering a sale". The participants involved in this task are: a `customer`, a `store`, and the `item` sold.
s

From an OOP viewpoint, we would treat `customer`, `store`, and `item` objects as opaque, and we might model the task as follows.

- *`item.register_sale(store, customer)`*
- *`store.register_sale(item, customer)`*
- *`customer.register_sale(item, store)`*

A component would have to rely on the specific behavior of all the three object implementations to complete the sale transaction. However, if the behavior of a `customer`, `store`, or `item` changes---for example an operation used by the implementation of `register_sale` method is removed or its pre/post conditions or signature altered, the `register_sale` method would fail. Thus, this approach is *brittle* to changes in the participants, as it relies on the fact that their behavior will not change over time.

From a DOP viewpoint, we would define `customer`, `store`, and `item`, as publicly exposed data, and formally describe their structure as public meta-data. We might define a message called `register_sale` that operates on the `customer`, `store`, and `item` data to accomplish the task.

- *`register_sale(customer, store, item)`*

The consumer (or provider) of this message would have all the information necessary to execute the producer's (or requestor) request, and its implementation is no longer tied to the behavior of the `customer, store, item` objects. If the definition of `customer`, `store`, or `item` changes, the associated meta-data is be updated to inform the consumer of those changes, so that the data processing in the application logic can be adjusted accordingly. Thus, this approach is *robust* to the changes in the data structure, as well as the behavior of the participants.

3 The Role of the Middleware Infrastructure

Middleware refers to the software layer that sits between the application-logic and the underlying operating-system (if any).
Network middleware is responsible interconnecting components in a distributed system.

The network middleware infrastructure plays a critical role in the application of the DOP paradigm. Let's examine the role and the array of middleware choices available to use, and determine the ones most suitable for building loosely-coupled systems and SoS.

3.1 Middleware Infrastructure Requirements

As we noted in Figure 4, there are several middleware choices available for building distributed systems. The key requirement is that the middleware infrastructure be amenable to DOP principles; which means that the middleware infrastructure:

- Provides a mechanism for formally specifying meta-data (data-models and QoS) (principle 1);
- Does not force a common context or state or coupling across messages, and encourages state-less interactions (principle 2);
- Provides a total separation of data-handling from application logic, and does not impose any constraints on application logic (principle 3);
- Provides an inter-operability protocol for message representation and exchange "on-the-wire", so that independent implementations can meaningfully communicate (principle 3).
- Provides an abstract programming model, so that application specific data-handling code can be maintained separately from application logic in any desired target programming language (principle 4);

Also, considering the additional needs of dynamic real-time adaptation, scalability, and performance in loosely-coupled systems and SoS, the following capabilities can go a long way in enabling a clean and effective application architecture.

- Ability to dynamically specify and (re)configure the data flows;
- Ability to describe delivery requirements per data flow;
- Ability to specify and control middleware resources such as queues and buffering;
- Resiliency to individual node or participant failures; and
- Performance and scalability with respect to number of nodes, components, and data flows.

In addition, it is always desirable that a middleware technology be an open standard and not a proprietary technology, so that once can benefit from the competitive market forces, without incurring the costs of vendor lock-in.

3.2 Middleware Infrastructure Choices

One can quickly walk through the list of currently available middleware infrastructure technologies (some are listed in Figure 4) and narrow down the list of suitable technologies. It is not practical to undertake a detailed examination of all the popular middleware technologies in the scope of this paper; that will remain a subject to be tackled elsewhere. However, we briefly summarize the key findings.

We observe that the middleware technologies that support the *anonymous publish-subscribe* paradigm (see Section "Data-Centric Publish-Subscribe Middleware Infrastructure") have the best potential for meeting the requirements in the previous section. Dominant publish-subscribe middleware standards include: Data Distribution Service (DDS) [OMG 2006], Java Message Service (JMS) [JMS], High Level Architecture (HLA) [HLA], CORBA Notification Service [OMG 2004]. An overview of DDS is presented in [Joshi 2006b]. A detailed comparison of DDS and JMS is presented in [Joshi 2006a]. A detailed comparison of DDS and HLA is presented in [Joshi 2003].

Upon an in-depth examination of these middleware standards, DDS [Joshi 2006b] is the only middleware standard that meets and exceeds the requirements of the previous section for loosely-coupled systems and SoS. The key reasons include (1) DDS is the only publish-subscribe middleware standard that specifies both an API specification and an interoperability wire-protocol; (2) DDS supports a request/offered paradigm central to the design of loosely coupled systems; (3) DDS provides mechanisms for applications to be self-aware by informing them of network topology changes; (4) DDS does not make any assumptions about the state at the other components; (5) DDS requires that the data model be specified formally in a programming language neutral manner.

Specifically, JMS [Joshi 2006a] and message-oriented middleware (MOM) does introduce coupling via the use of a message "destination". Destinations are a point of tight-coupling for the following reasons: (1) in most implementations, destinations must be configured explicitly before they can be used; (2) a destination is a shared resource, accessed by producers and consumers; (3) a destination is specified when sending a message; (4) destinations can become communication bottleneck and a single point of failure. Additional reasons why JMS falls short of the requirements of DOP can be found in [Joshi 2006a].

Lets us examine the suitability of DDS for the needs of loosely-coupled systems and DOP.

3.3 Data-Centric Publish-Subscribe Middleware Infrastructure

DDS [OMG 2006, Joshi 2006b] is an abstract programming model specification and an inter-operability wire-protocol, that defines a *“data-centric”* publish-subscribe model for connecting anonymous information producers with information consumers.
A standard language independent programming model (or abstract API specification) ensures that application source code is portable across different implementations, and can be precisely defined for current and future programming languages. A standard wire-protocol ensures that applications written to different vendor implementations of the standard API will inter-operate because they all use the same low-level hand-shaking protocol on the network.

A distributed application is composed of “*participants*”, each potentially running in a separate address space, possibly on different computers. A participant may simultaneously publish and subscribe to typed data-flows identified by names called “*topics*". The DDS APIs allows typed “*data readers*” and “*data writers*” to present type-safe programming interfaces to the application. A typical DDS application architecture can be represented as software “*data-bus*" shown in Figure 6.

Data-Centric Publish-Subscribe Middleware

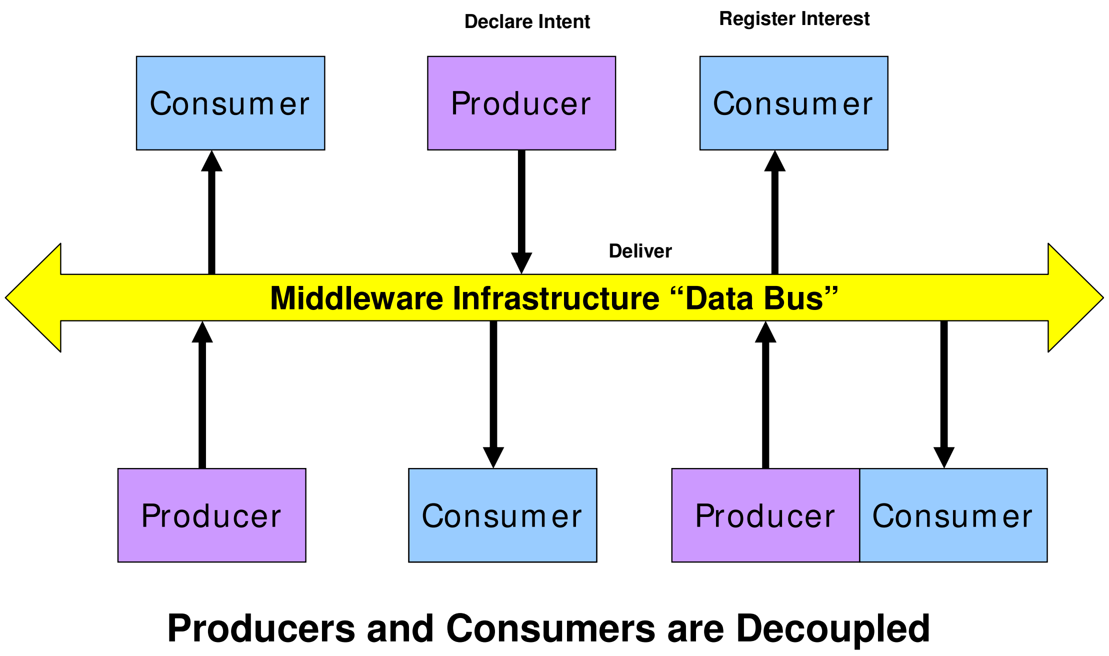


Figure 6 DDS data-centric publish-subscribe middleware decouples data producers from data consumers in location, space, platform, and multiplicity.

A producer declares the intent to produce data on a topic by creating a data writer for it; a consumer registers interest in a topic by creating a data reader for it. The middleware infrastructure manages these declarations, and automatically establishes direct (peer-to-peer) data-flows between data writers and data readers matching a topic. Thus, an application programmer can essentially ignore the complexity of the data flow; each node gets the data it needs from the bus, or puts the data it updates on the bus. The bus logically represents a "*shared data space*" that any participant can read and write via data readers and data writers.

As a result, the communications are decoupled in space (participants can be anywhere), time (delivery may be immediately after publication or later), flow (delivery QoS can be precisely controlled), platform (participants can be on different implementation platforms, and written in different languages), and multiplicity (there can be multiple data writers and data readers of a topic).

DDS has several unique aspects that make it especially suited for loosely-coupled systems and DOP. These include:

- The ability to define a type system for topics on the data bus. A type formally specifies the data-model or schema that describes the data on a topic. This directly supports DOP (principle 1).

- The ability to formally associate QoS with each data writer and data reader. DDS formally defines semantics of QoS parameters along with a partial-ordering on their values. A producer can "offer" a certain QoS, while a consumer can "request" a certain QoS. The middleware infrastructure establishes direct data flow if-and-only-if the requested QoS is "compatible" (as defined by the partial ordering) with what is offered. This gives us a mechanism to formally address the impedance mismatch challenge (see Section "Impedance Mismatch") and directly supports DOP (principle 1).

- Mechanisms for the application to detect and specify what should happen when QoS assertions are violated. This is consistent with DOP (principle 3).

- Mechanism to "*discover*" when data readers and data writers for a topic are created or deleted, when a direct data flow is established, and when a direct data flow could not be established due to incompatible QoS. This gives us a framework to address the challenge of dynamic real-time adaptation (see Section "Dynamic Real-Time Adaptation").

- No assumptions about the reliability of the network transport. The applications can deal with reliability, and connection management using the relevant QoS policies.

- Data is physically transmitted on the network if-and-only-if there are data readers requesting a QoS compatible with a data writer's offered QoS. Thus, DDS uses the network conservatively. In general, the DDS APIs and wire-protocol strives to deliver the highest possible performance (Section "Scalability and Performance") for the available resources and requested capabilities.

- To increase scalability, topics may contain multiple independent data channels identified by "keys". This allows a consumer to subscribe to many, possibly thousands, of similar data channels with a single subscription. When the data arrives, the middleware infrastructure can sort it by the key and manage it on behalf of the application logic, to enable efficient processing. This is one way in which DDS addresses the challenge of scalability and performance (see Section “Scalability and Performance”).

Keys are also a mechanism to specify a relational data model to the middleware infrastructure; the middleware can manage the relational data model on behalf of the application. This is consistent with DOP (principle 3).

In typical DDS usage, the data-model or FMRFD of types is expressed in a data description language such as IDL [OMG 2002]. A middleware implementation infrastructure specific code-generator (Figure 7) is used to generate the type representation in the target programming language and type specific facades for data readers and data writers, for use by the application in the target programming language. The generated code completely encapsulates the data-handling, while isolating the application logic as required by DOP (principles 3 & 4). QoS can be formally associated with the data readers and data writers by the application logic, in a manner consistent with DOP (principle 1).

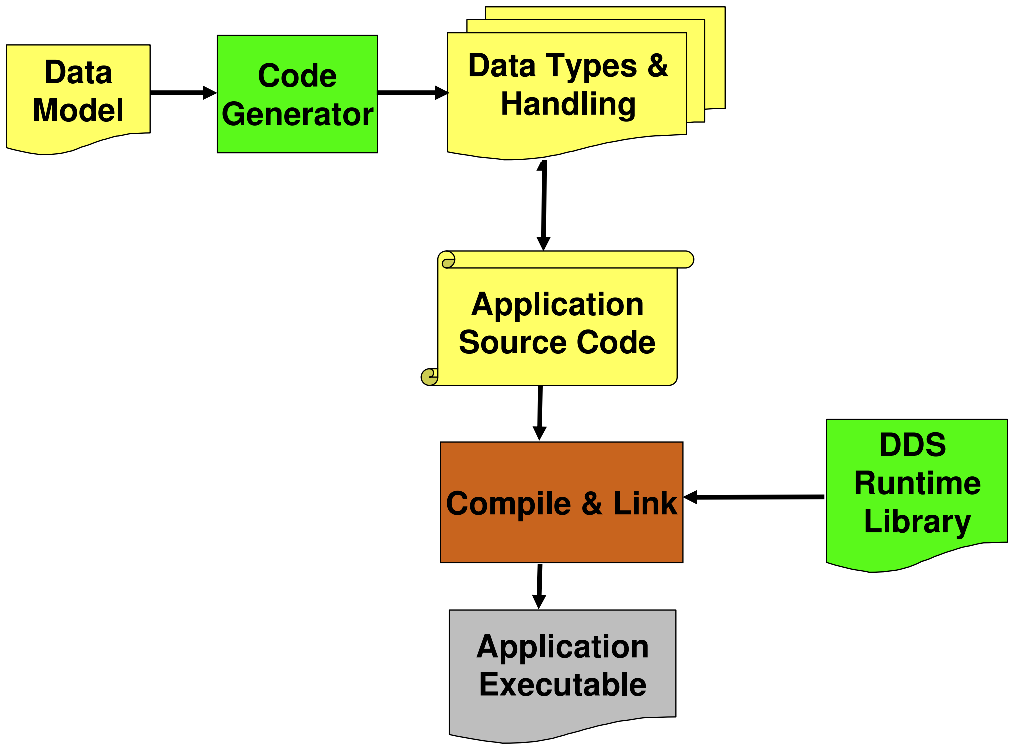


Figure 7 Typical DDS application workflow.

Unlike other publish-subscribe middleware infrastructure, DDS has a very strong emphasis on data modeling (as noted above). Besides supporting basic messaging and events, there is support for formally defined types, relational data modeling; data lifecycle management; content based filtering; and data transformations directly in the middleware infrastructure. For these reasons, DDS is often referred to as a "*data-centric*" publish-subscribe middleware.

The best DDS implementations [RTI DDS] do not impose any centralized configuration requirements on the applications. In particular, the applications don't require information about the other participating applications, including their existence or location. A DDS implementation can automatically handle all aspects of data exchange, without requiring any intervention at the application level.

The data-centric nature of DDS, and the fact that there is no intrinsic point of coupling between DDS applications (for example a shared resource) is the reason why DDS is especially well suited for implementing a very broad class of applications, using DOP principles.

4 Application Architecture: Applying the Data-Oriented Paradigm

“Application” refers to the domain specific portion of a system, to realize some purpose. The application-layer sits above the middleware-layer, and contains logic specific to the purpose of the system. “Architecture” refers to the organization of a system’s components, their inter-connections, properties, and relationships with one another.
Thus, “application architecture” refers to the organization of application-logic components and their relationships, built on top of a middleware infrastructure. It encompasses the overall interaction patterns, common data-models, QoS, and application specific semantics.

In this section, we examine the implications of the DOP paradigm and the supporting middleware infrastructure on the application architecture, and relate them to currently popular application architectural styles. We describe some emerging categories of “of-the-shelf” application components to address the emerging needs of SoS, especially edge to enterprise integration (see Section “A Real-Time System-of-Systems Scenario”).

Finally, we concretely illustrate the application architecture for a realistic edge to enterprise integration scenario that was implemented using the DOP principles in a relatively short amount of time, using commercially available “off-the-shelf” middleware infrastructure and application components.

4.1 Application Architecture Styles

An application architecture style defines a pattern for organizing and developing application components to achieve certain objectives. The successful use of an architectural style anchors on the underlying design paradigm used for component development, and supported by the middleware infrastructure. An architectural style may be realized in software using a tightly-coupled design approach or a loosely-coupled design approach.

The DOP paradigm and the DDS middleware infrastructure are extremely versatile in their ability to support multiple architectural styles, and realize loosely-coupled software implementations. The generic application architecture addressing enabled by DOP and DDS can be thought of as "wiring diagram", shown in Figure 8. A topic represents a "virtual wire" managed by the DDS middleware infrastructure. A topic may be viewed as organized into "records" or "data-objects" that are managed by the middleware. A component uses a data reader to access a data-object on a topic, and a data writer to update a data-object on a topic.

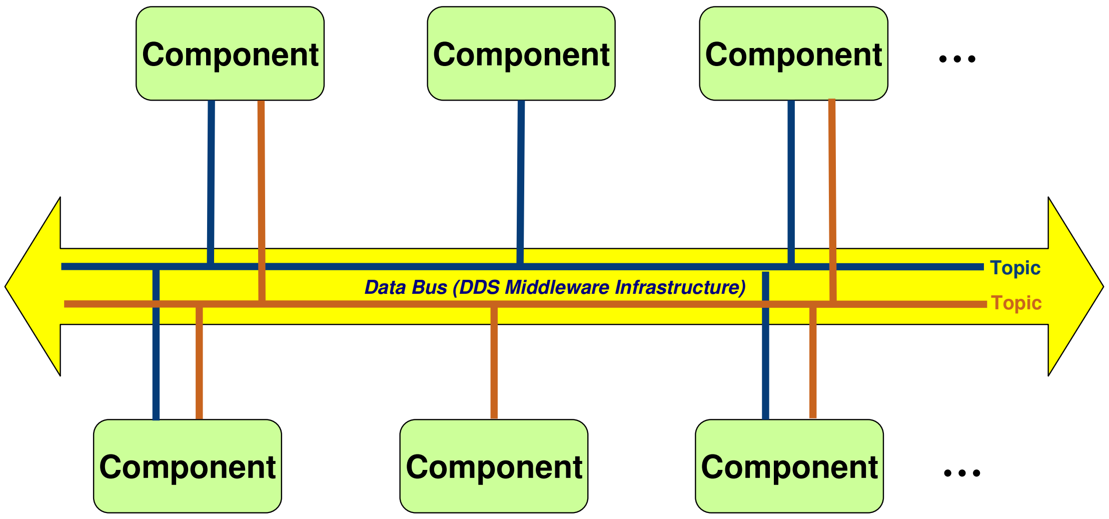


Figure 8 Data-oriented integration architecture, for developing loosely coupled applications. Components can be added and removed independently, without any knowledge of other components. Data readers and data writers of topics (data-flows) can be created, used, and deleted independently by a component, without requiring any centralized configuration or changes. Direct data paths are automatically established between data readers and data writers of a topic, and managed by the middleware infrastructure.

This generic architecture meets the needs of a wide range of applications, which may be characterized as follows.

1. Components are loosely-coupled;
2. Interactions between components are data-centric and not object-centric; often these can be viewed as “dataflows” that may carry information about identifiable data-objects;
3. Data is critical because of large volumes, or predictable delivery requirements, or the dynamic nature of the entities;
4. Computation is time sensitive and may be critically dependent on the predictable delivery of data;
5. Storage is local.

The predominant styles found in the Edge and Enterprise systems, and SoS including data flow architecture, event driven architecture, service-oriented architecture can be realized as specializations of this generic architecture. It is important to note that the above architectural styles can also be realized as tightly-coupled software---in fact that is typically the case in current practice. Our aim here is to illustrate how these can be realized as *loosely-coupled* software, utilizing the DOP design paradigm and the DDS middleware infrastructure.

4.1.1 Data Flow Architecture (DFA)

The data flow architectural (DFA) style is most common in sensor based edge systems and control systems. Sensors are data producer components that feed data into processing components. Controller components consume the data inputs, and produce data outputs for actuators or other components. Thus, the components naturally form an acyclic directed data flow graph. This architectural style can be viewed as a special case of the generic data-oriented integration architecture, as shown in Figure 9.

Data Flow Architecture

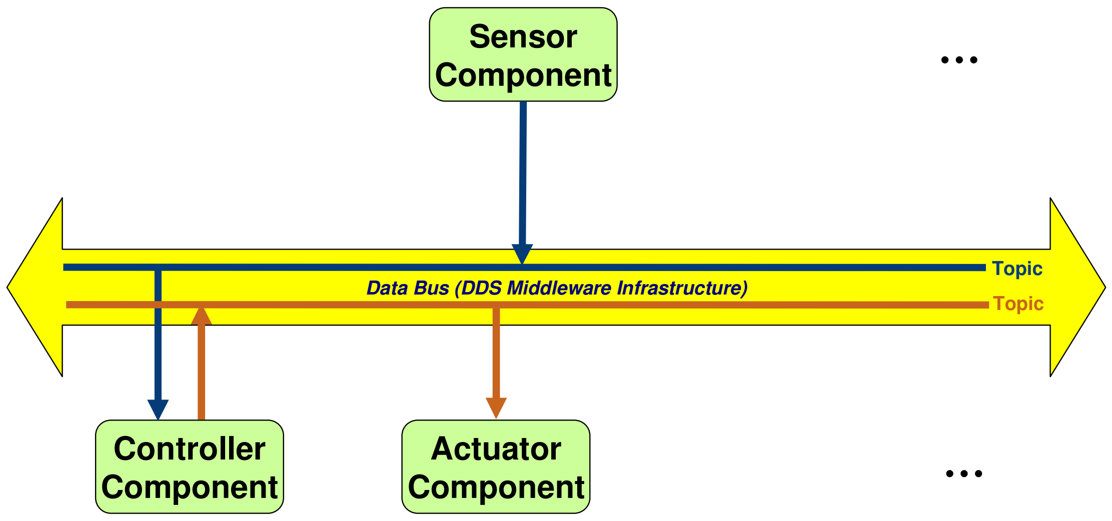


Figure 9 Data flow architectural style can be seen as a specialization of the generic data-oriented application architecture enabled by using a DOP and DDS.

The QoS are chosen to achieve the desired flow characteristics. For the sensor data topics, QoS are chosen to retain only the latest most up-to-date values.

In this architecture, components operate periodically and concurrently. A controller component may have an operational loop as follows.

- Latch all the inputs
- Process data
- Write output

4.1.2 Event Driven Architecture (EDA)

The event driven architectural (EDA) style is most common in enterprise systems. An “event” is defined as something “notable”, for example, a significant change in state that happens inside or outside the system. An “event generation component” produces events. An “event processing component” consumes events. An event can be a trigger for initiating downstream action(s). It processes the events against various rules and initiates actions. In addition, it may post new events for other “downstream activity” components. This architectural style can be viewed as a special case of the generic data-oriented integration architecture, as shown in Figure 10.

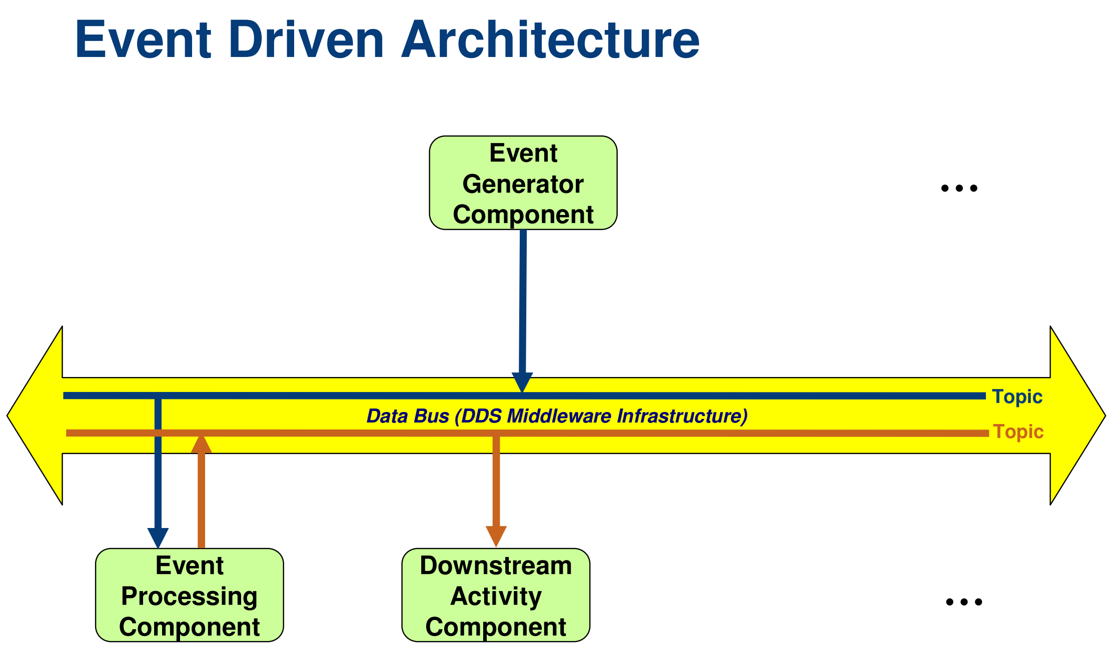

Event Driven Architecture
Event Generator Component
…
Topic
Data Bus (DDS Middleware Infrastructure)
Topic
Event Processing Component
Downstream Activity Component
…

Figure 10 Event driven architectural style can be seen as a specialization of the generic data-oriented application architecture enabled by using a DOP and DDS.

The QoS are chosen to achieve the desired flow characteristics. For event data topics, QoS are chosen to retain all the last N events, in the order they happened.

Events are commonly used to drive the real-time flow of work, and take the lag time and cost out of a business operations. Generally, an event processing component is a *rule-based engine*. In a simple event processing engine, each event occurrence is processed independently. In a complex event processing engine (CEP), new event occurrences are processed in context of prior and future events.

4.1.3 Client Server Architecture (CSA)

The client-server architectural (CSA) style is most common in enterprise systems, and is also applied for integrating edge and enterprise systems. A "service" a discretely scoped business or technical functionality, and is offered by a "service provider" or server to "service requestors" or clients. Operationally, a service is defined by messages exchanged between clients and servers.
This architectural style can be viewed as a special case of the generic data-oriented integration architecture, as shown in Figure 11.

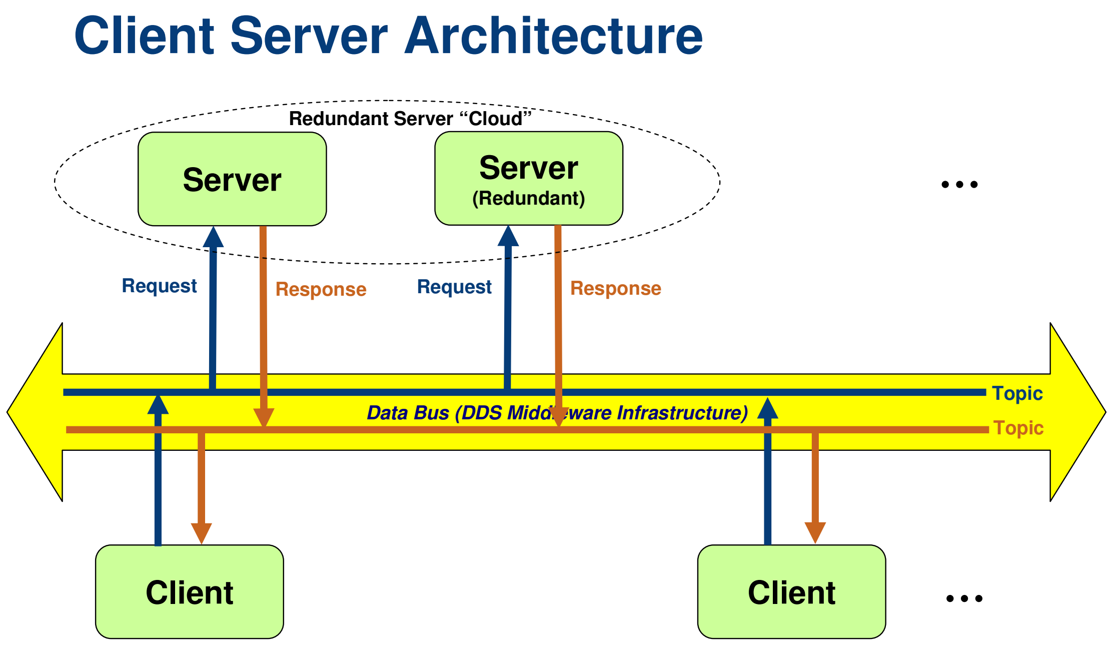


Figure 11 Client-Server architectural style can be seen as a specialization of the generic data-oriented application architecture enabled by using a DOP and DDS.

The request and responses are correlated with a client specific "`correlation_id`" field in the data model. The QoS and are chosen to achieve the desired flow characteristics. For request and reply topics, QoS are chosen to deliver all the data samples in the order they happened. Content filtering is used on the response topic to receive only the responses intended for the requestor.

It is interesting and ironic to note that despite the hype, the vast majority of the implementations of SOA relies on tightly-coupled technologies and design principles, resulting in tightly-coupled SOA software. This runs counter to the promise of SOA, and can prove detrimental to its success in the long run. We have outlined an approach that results in "*loosely-coupled SOA*" software, and can indeed unleash the full potential of SOA.

4.2 Data-Oriented Integration Architecture

Large scale distributed SoS are loosely often a mish-mash of systems with different architectural styles, created by independent parties, often using different middleware technologies, with misaligned interfaces. A naïve approach to integrating such systems results in N*N point-to-point custom integrations, for each pair of systems. This approach does not scale, yet, often it is the end outcome in practice.

A better approach is to use the DOP paradigm, and explicitly formalize the data and meta-data produced and consumed by a system. The DDS middleware infrastructure can be used as the integration "glue" to connect the disparate systems, as shown in Figure 8. This generic architecture can accommodate a wide variety of architectural styles, and reduces the integration problem from an O(N*N) problem to an O(N) problem.

Can we do better? We observe that a larger number of application components are built using "off-the-shelf" application platform components. These include databases for storing and retrieving data, event processing engines, application servers providing web services, enterprise service bus for transformation and routing of data, workflow engines using business process execution language, and so on. Thus, an "out-of-the-box" integration of DDS and these popular application platform technologies can indeed reduce the integration problem from O(N) to that of exposing the data and meta-data of just the legacy "one-of" application components, and integrating them into the DDS middleware infrastructure bus. New integration projects can leverage the widely available and tested integrations for the popular application platforms. This concept is illustrated in Figure 12.

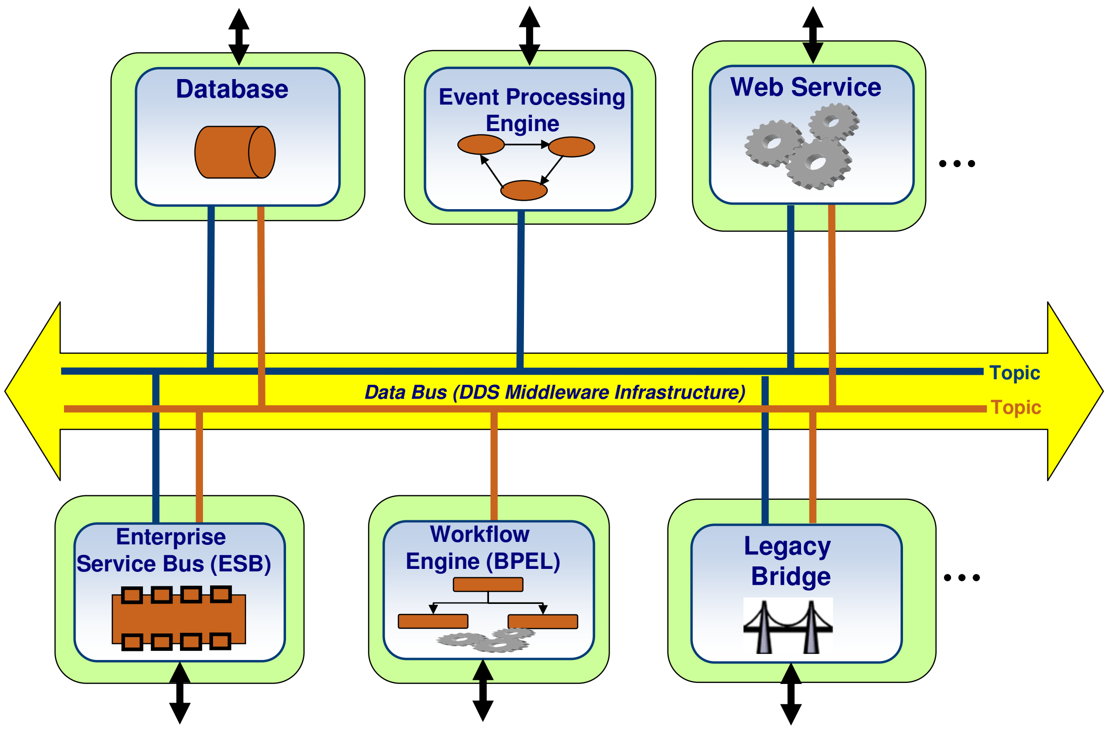


Figure 12 "Out-of-the-box" integrations for popular application platform components can speed up the integration task, when used with the generic data-oriented integration architecture enabled by DOP and DDS.

Leading DDS middleware infrastructure vendors have already begun providing products that integrate popular application platform technologies into the DDS software bus [RTI].

4.3 Real-Time System-of-Systems Scenario Redux

In this section we illustrate the concepts discussed in the previous sections with the help of a real-time package tracking scenario. We illustrate how the data-oriented integration architecture (Figure 14) based on the DOP design paradigm and DDS middleware infrastructure can be used to rapidly develop a working demonstration in a very short time frame. The rapid implementation was a direct result of the ready availability of a DDS bridge for relational databases, and supporting tools.

4.3.1 Real-Time Package Tracking Scenario

Consider a real-time package tracking scenario, shown in Figure 13. Packages shipped by various means need to be tracked, and a unified view presented in a web browser based unified "dashboard" for in-time executive level decision-making.

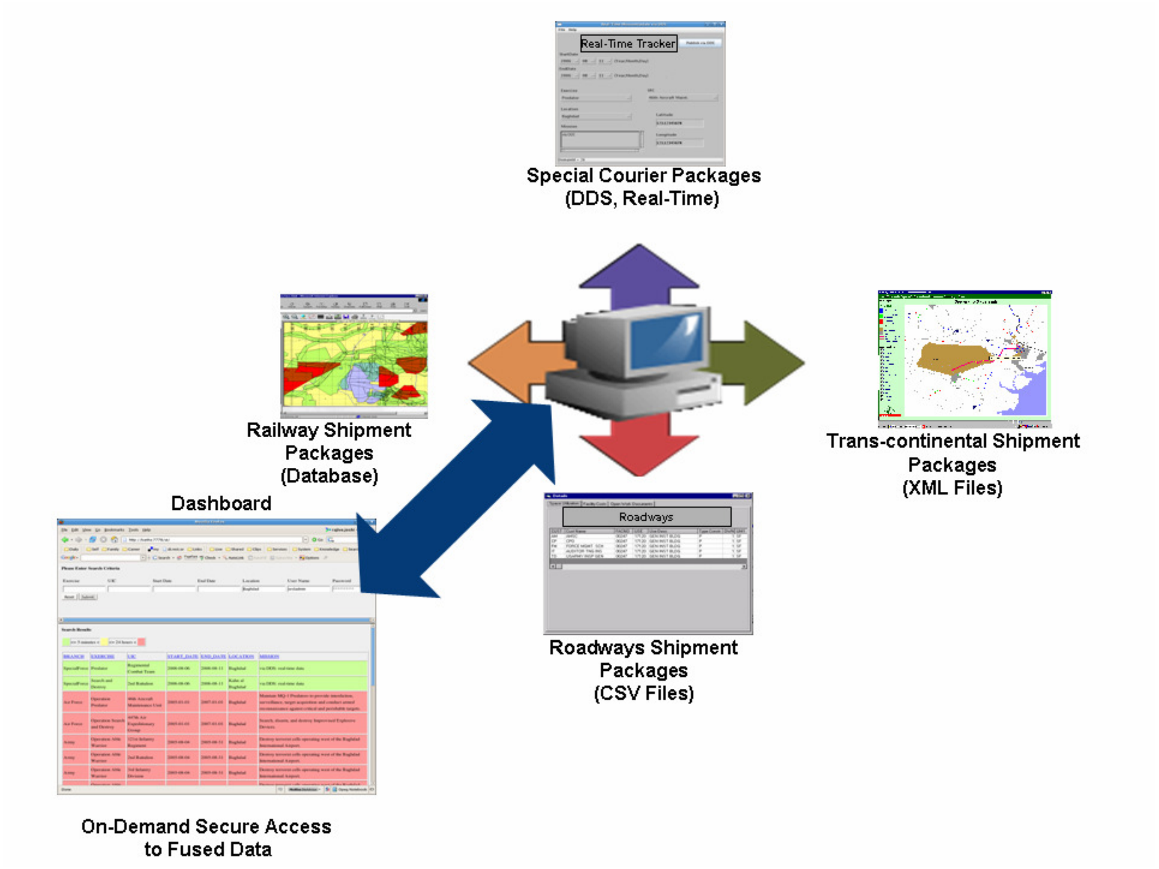

Real-Time Tracker
Special Courier Packages (DDS, Real-Time)
Railway Shipment Packages (Database)
Trans-continental Shipment Packages (XML Files)
Dashboard
Roadways
Roadways Shipment Packages (CSV Files)
On-Demand Secure Access to Fused Data

Figure 13 A system-of-systems for real-time package tracking. Status of the packages shipped by various means is displayed in an executive "dashboard" for real-time decision making. Each means of shipment is a separate system, independent of the executive dashboard system.

There are five independent systems involved: each mode of shipment is an independent system, and the executive "dashboard" is yet another system that interacts with all these systems.

Each shipping system uses a different method for keeping track of its operations, and makes its package status data available in its own format, at different intervals, by different means. For example, the “railway” shipping system logs the status of its packages into a persistent storage database, whenever a train makes a stop. The roadways shipping system produces a comma separated value file containing the current package status, and sends it over email every 4 hours. The trans-continental shipping system updates an XML file accessible over FTP, with the package status whenever it crosses a border or changes carriers. The special courier shipping system is decentralized: each courier uses RFID sensors to publish its local snapshot of packages on a periodic basis, at least once every hour. The courier updates are published over a DDS middleware infrastructure bus.

The special courier shipping system might be classified as real-time “edge” system; the rest might be considered “enterprise” systems.

4.3.2 Data-Oriented Edge to Enterprise Integration Architecture

The data-oriented edge to enterprise integration architecture is shown in Figure 14.

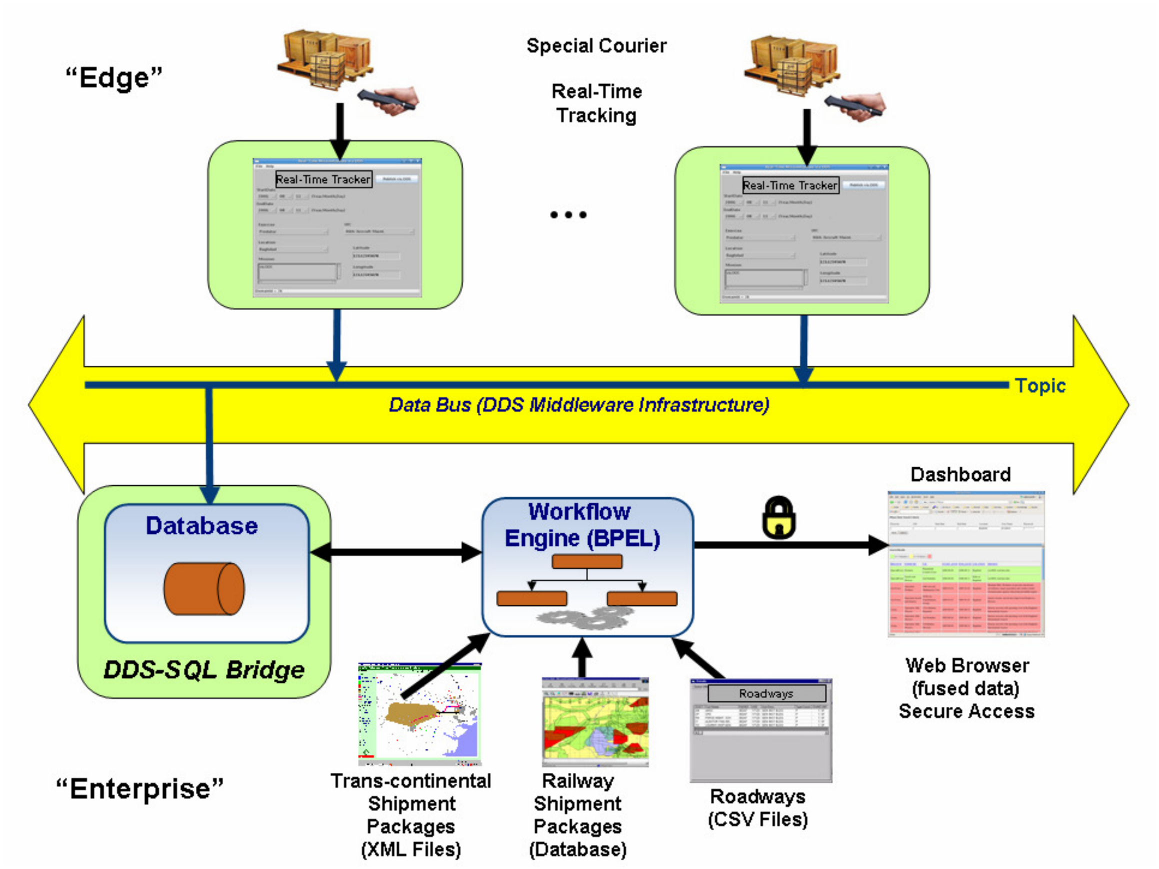
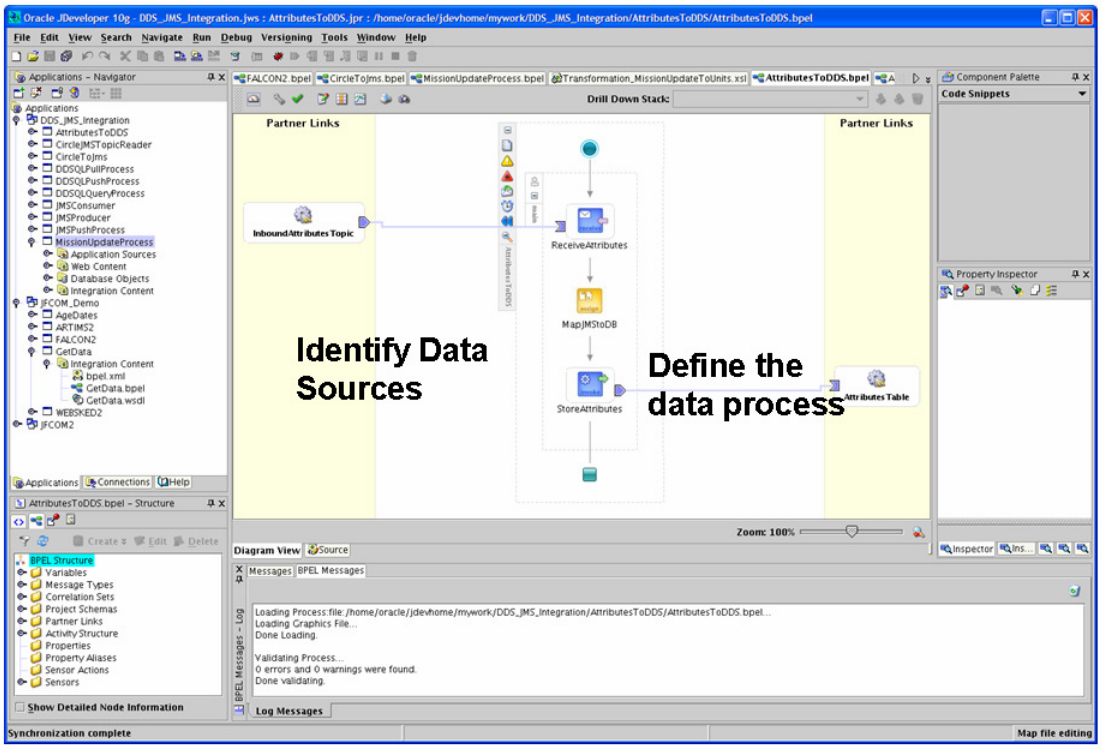


Figure 14 Data-oriented edge to enterprise integration architecture for the real-time package tracking SoS.

The architecture includes several application components, summarized below.

- The decentralized special courier shipping device, capable of publishing package status over DDS. These may provide a user-interface for the courier operator to augment additional information, not sensed by RFID tags.
- The enterprise dashboard system which includes
    - A database for caching the "edge" data from the distributed special courier devices. A readily available *DDS-SQL bridge* is used to automatically store the data on the DDS topics.
    - A workflow engine using business-process execution language (BPEL, Figure 15) to orchestrate among the different systems, and fuse the incoming data in different formats into a common fused data schema.

    - The BPEL workflow engine has adapters for accessing data in a Database, an XML files, a CSV file, and eMail, produced by the railway, trans-continental, and roadway shipping systems. In our implementation, these adapters were used to access the different data sources from the BPEL engine. A downside of this approach is that it introduces a very *tight coupling* between the BPEL processes and the systems, and the data is not easily accessible from other applications.
        - Ideally, to be completely loosely-coupled, we would publish the data produced by the railway, trans-continental, and roadway shipping systems on the DDS software bus. However this was not practical in the available short time. Had there been, off-the-shelf bridges available for these data formats, the situation might be different. We opted to accept the tight-coupling for our demonstration purposes.
    - The BPEL workflow engine has concurrently running asynchronous processes (for each shipping system) to load the incoming data; transform it into a common fused data schema; and save it into a local database table.
    - The BPEL workflow engine has a synchronous process waiting to serve up requests for the fused data.
- A web-browser to access the executive “dashboard” which offers unified view of package status across all modes of shipment, and supports querying. A secure web-services gateway is used to guard against un-authorized access to the data.



```
Loading Process:file:/home/oracle/jdevhome/mywork/DDS_JMS_Integration/AttributesToDDS/AttributesToDDS.bpel...
Loading Graphics File...
Done Loading.

Validating Process...
0 errors and 0 warnings were found.
Done validating.
```

Figure 15 Workflow engine uses Business Process Execution Language (BPEL) process to orchestrate services and manipulate data.

The data-path from the special courier “edge” system in the field, to the “enterprise” executive dashboard in the boardroom, is shown in Figure 16.

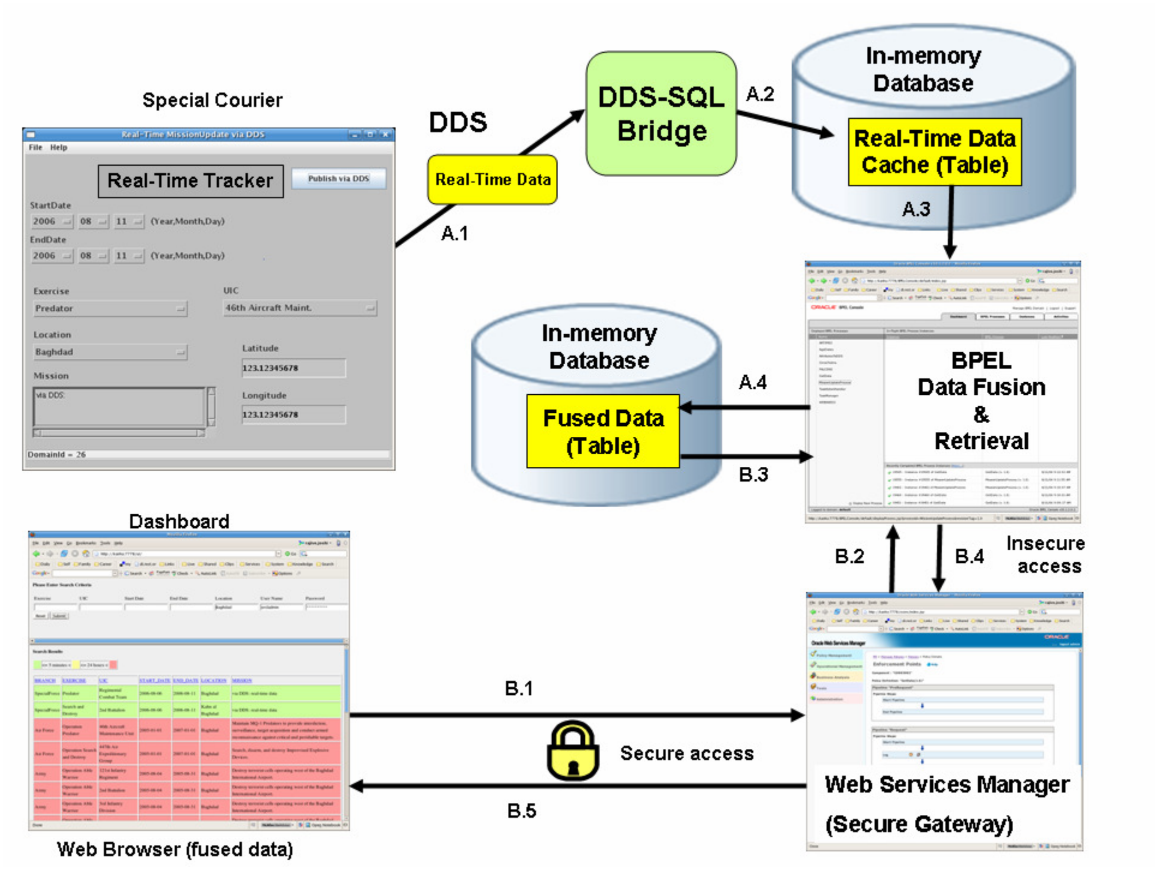


Figure 16 End to end data paths from the "edge" data collection system in the field to the "enterprise" executive dashboard in the boardroom. The sequence A.1…A.4 shows the data path from the edge system to the enterprise system. The sequence B1..B.5 shows the secure access of the integrated view maintained by the enterprise system from the executive dashboard.

The data from the special courier's device is published on a DDS topic (A.1) and received by a DDS-SQL bridge. The DDS-SQL Bridge logs it into a table associated with the DDS topic (A.2). Note that the DDS-SQL bridge (commercially available) is capable of automatically creating the table schema from the meta-data associated with a DDS topic. The DDS-SQL bridge is also the *impedance matching device* for filtering and sub-sampling the real-time data being produced at much higher rates than the capacity of the BPEL workflow engine. The impedance tuning is achieved by adjusting the DDS QoS for the DDS consumers associated with the DDS-SQL bridge. An in-memory database is used with the DDS-SQL bridge for real-time performance.

An asynchronous BPEL process loads the changes in the data table (A.3) using a BPEL-Database Adapter (commercially available). It transforms the data into a common “fused” data format that unifies the data schema across all the different shipping systems (Figure 17). The BPEL process stores the transformed data into a different table (A.4), which is a cache of the fused data. Similar BPEL processes are concurrently active for the other shipping systems, and updating the fused data cache.

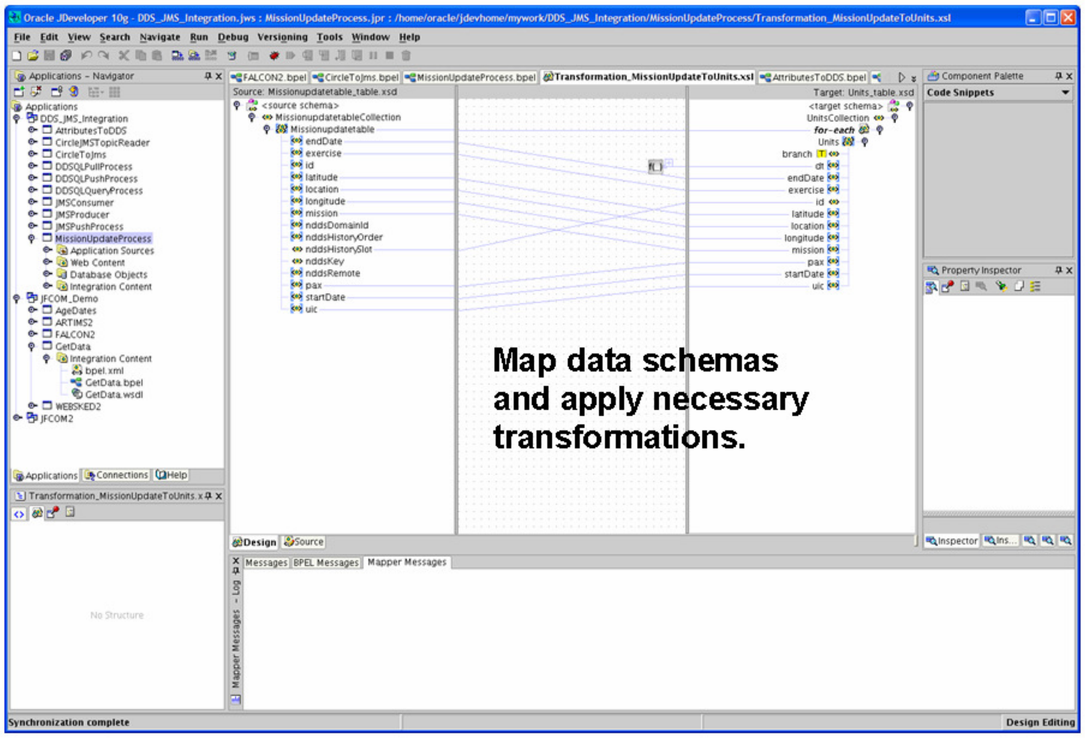


Figure 17 Data in one schema is mapped into a common fused data schema. Missing fields in the source format are given default values. Arbitrary transformations can be inserted in the mapping process.

A web browser is used to access the fused data. A request to access the fused data (B.1) is first intercepted by a web-services gateway (commercially available), that authenticates the request, An authenticated request is forwarded (B.2) to BPEL process waiting to serve up such requests. It looks at the request parameters, formulates an appropriate query and retrieves (B.3) the request fused data from the database table. The response is sent back (B.4 and B.5) to the web-browser.

In this architecture, we see a confluence of many different design paradigms, middleware infrastructure, and architectural styles. Albeit simple, this example captures the flavor of the real-life integration issues we face today. In reality, practical constraints and economic factors will dictate where to put (and not put) points of tight coupling, as we did in this exercise.

It is important to note that DOP is a central tenet of this architecture. The meta-data (the FMRFD) is specified *only once*, in an IDL file, as shown in Figure 18. All the other technology specific data schemas were “derived” from this FMRFD in IDL.

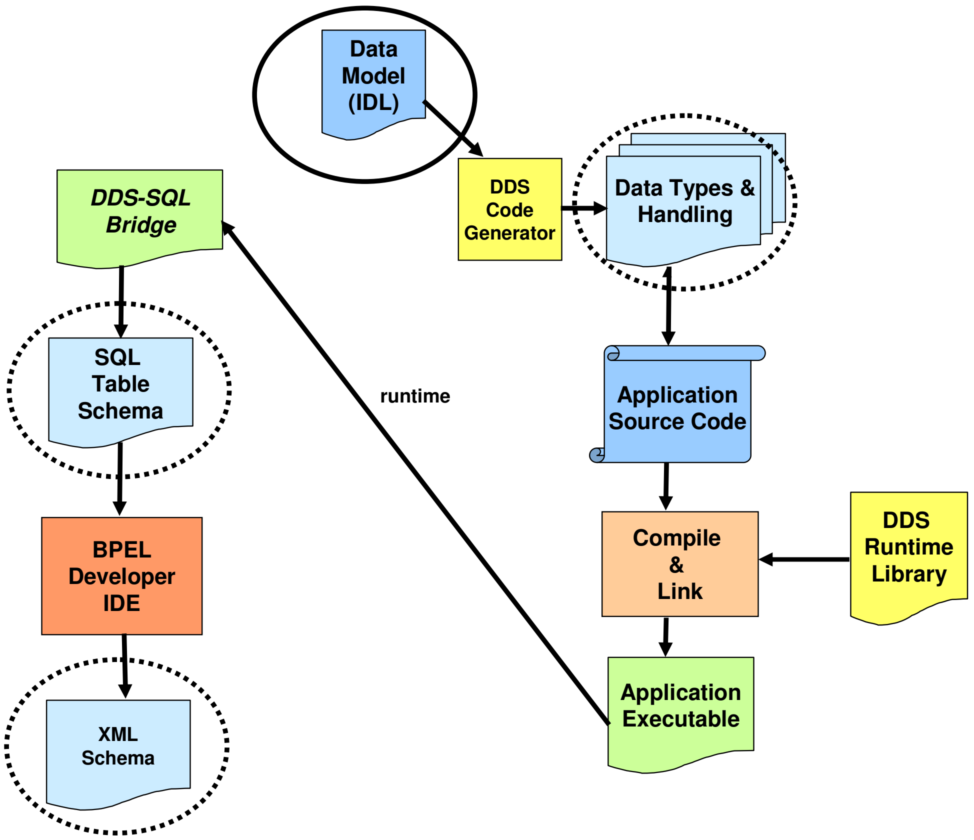


Figure 18 The meta-data is defined once by the developer in a FMRFD. The supporting tool-chain converts it into various representations for use with different technologies.

Thus, for the edge courier shipping devices, the meta-data was an IDL file describing the structure of the package status data. The DDS code generator produced the language specific types used by the application programs in the courier shipping system's edge devices. The DDS-SQL bridge automatically created the table schema for storing the contents of a topic, by looking at the runtime type-code information (meta-data) associated with the DDS topic. The BPEL Developer Integrated Development Environment (IDE) was capable of importing a database table schema and converting it into an XMLSchema, which was needed by BPEL processes to transform and manipulate the data.

Note that while DOP is critical to this integration architecture, the ready availability of a supporting tool-chain is what made this practical, and have a quick turnaround.

5 Conclusions

We looked at the key challenges in building the next generation of distributed systems. These will be loosely coupled systems that support incremental and independent development, and are tolerant of interface changes; can systematically deal with impedance mismatches; and work well in dynamically changing real-time situations; and can scale in complexity while delivering the required performance. We established that the existing design techniques that are effective for tightly-coupled systems do not work well in these scenarios. We established that a new design paradigm is needed.

We introduced the "data-oriented" design paradigm to address the design of loosely-coupled systems. We presented the fundamental principles with an example, and argued it as the basis of popular concepts of REST and SOA.

We examined the role of the middleware infrastructure when using data-oriented design, and argued that the data distribution service (DDS) is the best suited among the currently available standards, for data-oriented design.

We described a data-oriented application architecture based on data-oriented design and DDS middleware infrastructure. We shows that popular architectural styles, including data flow architecture, event driven architecture, service oriented architecture can be regarded as special cases, by the appropriate assignment of roles and choice of QoS.

We argued that data-oriented application architecture can cut down the complexity of the integration problem from O(N*N) to O(N), while preserving loose-coupling and scalability.

We illustrated the data-oriented application architecture by describing a working implementation of a real-time package tracking system-of-systems, built using off-the-shelf application platform components. We concluded that having readily available middleware infrastructure bridges for popular application platform components can greatly boost productivity and the pace of integration.

6 References

[HLA] DMSO. High-Level Architecture. US DoD Defense Modeling and Simulation Office. https://www.dmso.mil/public/transition/hla/

[JMS] J2EE Java Message Service (JMS), http://java.sun.com/products/jms/

[Joshi 2006a] Rajive Joshi. Building a effective real-time distributed publish-subscribe framework Part 1 – 3. Embedded.com. Aug 2006. http://www.embedded.com/showArticle.jhtml?articleID=191800680

[Joshi 2006b] Rajive Joshi and Gerardo Pardo-Castellote. OMG's Data Distribution Service Standard: An overview for real-time systems. Dr. Dobbs's Portal. Nov 2006. http://www.ddj.com/dept/architect/194900002

[Joshi 2003] Rajive Joshi and Gerardo Pardo-Castellote. A Comparison and Mapping of Data Distribution Service and High-Level Architecture. Simulation Interoperability Workshop. Fall 2003. http://www.sisostds.org/index.php?tg=fileman&idx=get&id=2&gr=Y&path=Simulation+Interoperability+Workshops%2F2003+Fall+SIW%2F2003+Fall+SIW+Papers+and+Presentations&file=03F-SIW-068.pdf

[Kaliski 1993] Burton S. Kaliski Jr. A Layman's Guide to a Subset of ASN.1, BER, and DER. An RSA Laboratories Technical Note. Revised November 1, 1993 http://www.columbia.edu/~ariel/ssleay/layman.html

[Kuznetsov] Eugene Kuznetsov. DOP: Data Oriented Programming. DataPower Technology, Inc. kuznetso@alum.mit.edu, eugene@datapower.com.

[Lamport 1978] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM 21(7). July 1978. http://research.microsoft.com/users/lamport/pubs/time-clocks.pdf

[OMG 2002] OMG. Interface Description Language (IDL). http://www.omg.org/cgi-bin/doc?formal/02-06-39

[OMG 2004] OMG. Common Object Request Broker Architecture. http://www.omg.org/docs/formal/04-03-12.pdf

[OMG 2006] OMG. Data Distribution Service for Real-time Systems, v1.2, http://www.omg.org/cgi-bin/doc?ptc/2006-04-09

[Pike 1989] Rob Pike. Notes on Programming in C. February 1989. http://www.lysator.liu.se/c/pikestyle.html

[Prescod] Paul Prescod. Roots of the REST/SOAP Debate. ConstantRevolution Consulting. http://www.prescod.net/rest/rest_vs_soap_overview/#section_1

[Richards 2006] Mark Richards. The Role of the Enterprise Service Bus. Oct 2006. http://www.infoq.com/presentations/Enterprise-Service-Bus

[RTI DDS] RTI Data Distribution Service, http://www.rti.com/products/data_distribution/RTIDDS.html

[RTI] Real-Time Innovations, Inc. http://www.rti.com

[Skonnard 2005] Aaron Skonnard. Contract-First Service Development. MSDN Magazine. May 2005. http://msdn.microsoft.com/msdnmag/issues/05/05/ServiceStation/

[Waldo 1994] Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall. A Note on Distrbuted Computing. Sun Microsystems Labs Technical Report TR-94-29s. Nov 1994. http://research.sun.com/techrep/1994/abstract-29.html

[Van Den Hoogen 2004] Ingrid Van Den Hoogen. Deutsch's Fallacies, 10 Years After. Java Developer’s Journal. Jan 2004. http://java.sys-con.com/read/38665.htm?CFID=716161&CFTOKEN=D3D89802-A102-8FD9-08669AC052B6E771

[W3C WSDL] W3C. Web Services Description Language (WSDL) Version 2.0. http://www.w3.org/TR/wsdl20-primer/

[W3C XML] W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition). http://www.w3.org/TR/REC-xml/

[W3C XMLInfoSet] W3C. XML Information Set (Second Edition). http://www.w3.org/TR/xml-infoset/

[W3C XMLSchema] W3C. XML Schema Part 0: Primer Second Edition. http://www.w3.org/TR/xmlschema-0/

[Wikipedia EDA] Wikipedia. Event Driven Architecture. http://en.wikipedia.org/wiki/Event_Driven_Architecture

[Wikipedia SOA] Wikipedia. Service Oriented Architecture. http://en.wikipedia.org/wiki/Service-oriented_architecture

7 Acronyms

Acronym	Description
API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
DDS	Data Distribution Service
DCPS	Data Centric Publish Subscribe
DLRL	Data Local Reconstruction Layer
EJB	Enterprise Java Beans
HLA	High Level Architecture
JDBC	Java Database Connectivity
JMS	Java Message Service
JNDI	Java Naming and Directory Service
Java EE	Java Enterprise Edition (previously known as J2EE)
JTA	Java Transaction API
MOM	Message Oriented Middleware
OMG	Object Management Group
P-S	Publish Subscribe
PtP	Point-to-Point

Pub/Sub	Publish Subscribe
RTI	Real-Time Innovations
RTOS	Real-Time Operating System
SOA	Service Oriented Architecture
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language