A Model For The Big Data Era

Data-centric architecture is becoming fashionable again **By Rajive Joshi**

Wired and wireless communication networks are making data collection and transmission cheap and widespread. In the future, networks will weave many devices and subsystems into complex integrated distributed systems that will become the fabric of business and daily life.

This **design** approach recognizes the essential **invariant** is the **information exchange** between components.

Building such distributed systems is far from simple; they must be assembled from independently developed software components. Integration, especially combined with real-time performance demands, becomes the key challenge.

This article outlines fundamental design principles that enable integration of distributed systems from components. I use a data-centric approach to this design, as the data is the key element that must flow through the various systems.

The key to data-centric design is to separate data from behavior. The data and data-transfer contracts then become the primary organizing constructs. With carefully controlled data relationships and timing, the system can then be built from independent components with loosely coupled behaviors. Data changes drive the interactions between components, not vice versa as in traditional or object-oriented design.

The resulting loosely coupled software components with data-centric interfaces are then integrated into a working system through a data bus. The data bus connects data producers to consumers and enforces the associated quality-of-service (QoS) contracts on the data transfers. This design technique is naturally supported by the Data Distribution Service (DDS) specification (*informationweek.com/1295/ddj/spec*) for real-time systems, which is a standard from the Object Management Group (*omgwiki.org/dds*). Implementations of this standard are available from many vendors.

The techniques described here are proven in hundreds of mission-critical applications including robotics, unmanned vehicles, medical devices, transportation, combat systems, finance, and simulation.

A Future Distributed System

To understand the dynamic nature of next-generation distributed systems, it helps to examine a representative scenario: an air traffic control system. Air traffic control in the future will integrate a variety of disparate systems into a seamless whole—a system of systems. On the edge is a real-time avionics system inside the aircraft. The control tower in the center communicates with the avionics system, and then out to data servers at the airport. The system thus comprises connectivity from the "edge" (devices) to the "enterprise" (infrastructure services).

The data in the avionics system flows at high rates and is time-critical. Violating timing constraints could result in the failure of the aircraft or jeopardize safety. Although aircrafts traditionally operate as independent units, future aircraft must integrate closely with automated traffic control and ground systems.

The control tower is another independent real-time system. It monitors various aircraft in the region, coordinates their traffic flow and generates alarms to highlight unusual conditions. The data is time-sensitive for proper local and wide area system operation. However, the system may have greater tolerance for delays than the avionics systems.

The control tower communicates with the airport's enterprise information systems, which track flight status and other data and may communicate with multiple control towers and other enterprise information systems. It also must synthesize passenger, flight arrival, and departure status information. Because it isn't in the time-critical path, the enterprise information system can be more tolerant of delays.

Key Design Challenges

This so-called "system of systems" must deal with a many issues, such as correctly handling myriad differences in data exchange, performance, and real-time requirements. The architecture also involves different technology stacks, design models, and component life cycles.

To support system growth and evolution, the integration must be robust enough to handle changes on either side of an interface. To do this, only minimal assumptions should be made about the interfaces between systems—the interface specifications should describe only the invariants in the interaction. Behavior can then be implemented independently by each system; the interface between them shouldn't include any component-specific state or behavior. This avoids tight coupling.

The systems on either side of an interface may differ in quantitative aspects of their behavior, including different data volumes, rates, and real-time constraints. The term "impedance mismatch" is shorthand for all the non-functional differences in the information exchange between two systems. Critically, a developer can capture these nonfunctional aspects of the information exchange by attaching QoS attributes to the data transfer. With explicit QoS terms, responses to impedance mismatches can be automated, monitored, and governed.

Principles Of Data-Centric Design

Data-centric design recognizes that the essential invariant is the information exchange between systems or components. It describes the exchange in terms of a "data model" and data producers and consumers of the data, and it relies on four basic principles:

1. Expose the data and metadata. Data-centric design exposes the data and metadata as first-class citizens, and uses them as the primary means of interconnecting heterogeneous systems. Data is the primary means of describing the world as it is, independent of any component-specific behavior. Metadata refers to information about the data's layout and structure. A data-centric interface is defined by the metadata, which must contain all of the information required to encode and decode the data in a given format.

2. Hide the behavior. Data-centric design hides any direct references to operations or code of the component interfaces. An interface can't embed any component-specific state or behavior. Components implement behaviors that can change the data or respond to changes in data (the "world model").

3. Delegate data handling to a data bus. Separation of data handling and application logic is necessary for loosely coupled systems. The component application logic should focus on manipulating interface data, not managing and distributing it. The data bus is responsible for data handling and is the authoritative source of the world model shared amongst the components.

4. Explicitly define data-handling contracts. These contracts should be specified by the application at design time, and enforced by the data bus at runtime. Delivery contracts specify the QoS attributes on data produced and consumed by a component, including timing, reliability, durability, etc. The data bus examines these "contracts," and if compatible, establishes data flows. The data bus then enforces QoS contracts, thereby providing the application code clear, known expectations.

In contrast, traditional messaging designs focus on functional or operational interfaces and overlook impedance problems. The interface QoS and timing aren't modeled, so all the interface state and communications issues are implicitly assumed. The result: a brittle, tightly coupled design. Adding components or interactions violates the assumptions, forcing system designers to rework the interfaces. The architecture becomes very hard to maintain and evolve.

Data-Centric Interfaces

A data-centric interface specifies the common, logically shared data model produced and consumed by a component, along with the QoS requirements.

A component can be seen as plugging into a software data bus via the data-centric interface that defines data inputs and outputs. When multiple components are present, the result is an information-driven, data-centric architecture in which data updates drive interactions between loosely coupled components.

A data-centric architecture reduces the integration problem since a component only has to integrate with the common data model intrinsic to the problem domain. Components implement data-centric interfaces that declare what they produce or consume. The QoS contracts ensure timing, reliability, and other requirements are met for any component, new or old. Thus, the system can grow and evolve.

The Data Bus

From a component programmer's perspective, the application code simply consumes and produces logically shared input and output variables on the data bus. Responsibility for data routing, delivery, and managing QoS is decoupled from the application logic and delegated to the implementation of the data bus.

The data bus requirements are fulfilled naturally by software that conforms with the DDS specification. That document defines the data-centric, publish-subscribe communication model for building distributed systems.

Several implementations of the DDS standard are available today, including an open source implementation and several commercial versions from RTI, Gallium, and Milsoft, among others. Leading DDS implementations provide deterministic low-latency, high-throughput messaging and data caching. While the most natural fit for these products has been in industrial, avionic, and military applications, they also have long been used in financial services, where the rapid distribution and processing of data is critical. And increasingly, as companies must handle large volumes of data, these products are entering business IT organizations.

One of the principal benefits for businesses is that a data-centric architecture paves the way for the use of generic infrastructure components. These include databases, complex event processing modules, and Web services. These components plug into the bus without the need for extensive custom coding to integrate them into the computing infrastructure. Done right, this model makes it possible for a spreadsheet to automatically populate cells from data items it subscribes to from a larger data fabric.

Because data-centric architectures have no direct coupling among the application component interfaces, components in the DDS model can be added and removed in a modular and scalable manner, letting companies add producers and consumers of data without a jump in complexity. As data volume expands, the simplicity of this architecture is likely to become a crucial part of a business's ability to keep up.

Rajive Joshi has worked in high-performance real-time distributed systems for more than 18 years, including implementing distributed messaging and data distribution caching infrastructure. Write to us at iwletters@techweb.com.

Cloud Programming? Ready, Set ... Yow

By Andrew Binstock

Clouds today come in two basic flavors: the private cloud (wholly behind the firewall) and the public cloud, which runs on remote hosts. The private cloud currently enjoys IT management's attention, because taking lots of individual servers, putting them into a pool, and parceling out their capabilities as needed has tremendous advantages for the data center. Of these, none is more prominent than the ability to scale up resources when projects demand and back down again when the need declines.

Private clouds require little programming change. Instances of virtual machines are spun up from an administrative console, the application is migrated, and, by and large, it works as expected.

The public cloud—whose leading hosts include Amazon, Google, and Microsoft—is a different thing altogether. Code can't be migrated simply to these hosts and expected to work correctly. It won't. Google's App Engine, for example, allows only a select list of core Java classes to run on its platform. If your code relies on a proscribed class, your app won't run there.

Moreover, each platform uses its own unique datastore, which doesn't run at all like conventional relational database management systems. (Microsoft Azure does offer a "cloudified" version of its SQL Server database product as an option.) So, if you plan to run applications in the public cloud, you'll have to invest considerable effort either porting existing code or writing new apps. Doing so will reveal a second problem: No two platforms use the same API. So, from the get-go, you'll be coding to a proprietary platform, with all the constraints that implies.

This problem is widely acknowledged, but efforts to provide a universal API, such as the Simple Cloud API, have garnered little enthusiasm from cloud hosts. This puts IT in a bind. If you're considering using the public cloud, therefore, run extensive pilots before committing to a platform, and know the platform's limitations and costs intimately before making it the basis of an important app. You're likely to be residing there a long time.

Andrew Binstock is the executive editor of Dr. Dobb's. He can be reached at alb@drdobbs.com.