The Data-Centric Future

A white paper by Stan Schneider

Pervasive data in distributed applications simplifies the design task for complex embedded systems

Truly profound technologies become part of everyday life. Motors, plastics, computers, and now networking have made this transition in the last 100 years. These technologies are embedded in billions of devices; they have melted into the assumed background of the modern world.

Another stage is emerging in this progression: pervasive, real-time data. This pervasive real-time infrastructure is an extension of the Internet; the new capability is to connect devices at very high speeds, rather than people at human-interaction speeds. Just as the “web” connected people and fundamentally changed how we all interact, pervasive data will connect devices and change how they interact.

Today’s network technology makes it easy to connect nodes, but not so easy to find and access the information resident in networks of connected nodes. This is changing; we will soon gain the ability to pool information from many distributed sources, and access it at rates meaningful to physical processes. Many label this new capability the “network centric” architecture. However, a more appropriate term is “data centric” because the change, fundamentally, is driven by fast, easy availability of information, not by the connectivity of the network itself. Whatever the name, this infrastructure will drive the development of vast, distributed, information-critical applications.

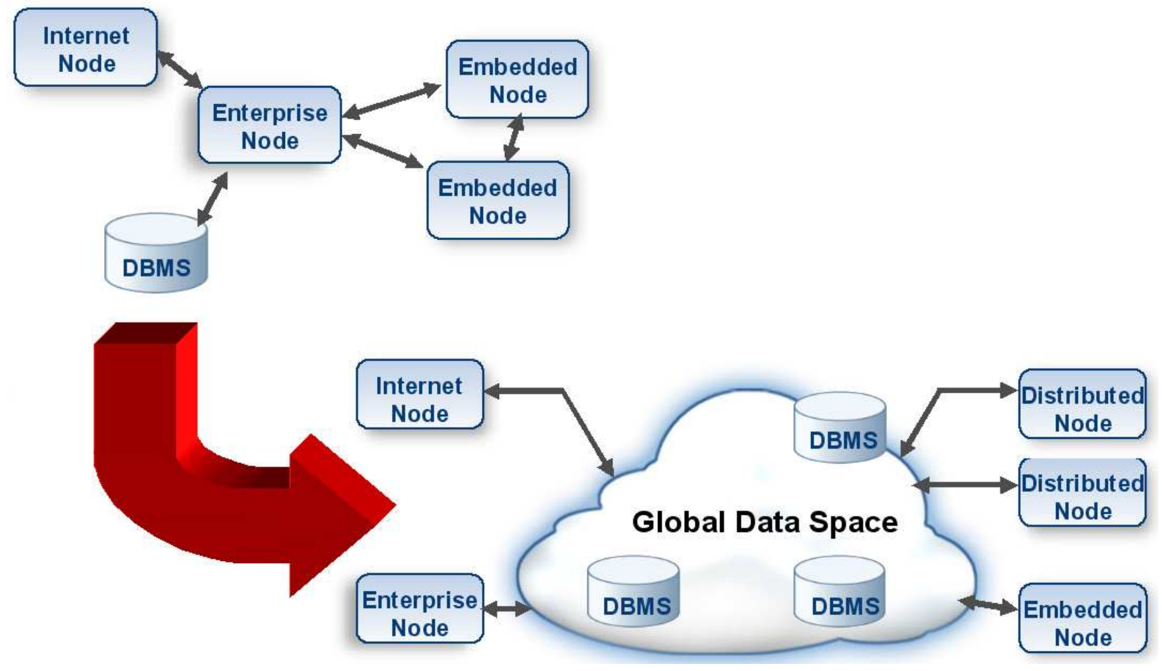

Figure 1: The Data Centric Future

Technology is evolving from data networks to pervasive data. With pervasive data, all information is available anytime at any place, without consideration of its origin.

Real-time middleware is the technological key driving this transformation. We will now examine the issues in developing a middleware that can succeed.

The rise of data-centric thought

Most systems built today are connected to a network. However, most are not designed to flexibly deliver data to support truly distributed applications. They are instead designed around clients and servers, application interfaces, and other code-centric concepts.

The real change is from code-centric or architecture-centric thinking to data-centric design. Instead of configuring clients and servers, or building data-access objects and invoking remote methods, data-centric design implies that the developer directly controls information exchange. Data-centric developers don't write or specify code. They build a "data dictionary" that defines who needs what data. Then they answer the information questions: How fast is the data coming? Does it need to be reliable? Do I need to store it? Where is it coming from? What happens if a node fails?

With this information in hand, the next task is to map the information flow. Publish-subscribe middleware is the key enabling technology for data-centric design.

Publish-subscribe nodes simply "subscribe" to data they need and "publish" information they produce. Thus, the information flow map is directly translatable to publishers and subscribers. The middleware does the rest: messages pass directly between the communicating nodes. The fundamental communications model implies both discovery—what data should be sent where—and delivery—when and how to send it.

This design should be familiar; it mirrors time-critical information-delivery systems in everyday life. All mass-media communications, including television, radio, magazines, and newspapers, are fundamentally publish-subscribe technologies. Publish-subscribe systems are good at distributing large quantities of time-critical information quickly, even in the presence of unreliable delivery mechanisms.

With direct access to data, system integration is orders-of-magnitude easier. Designers have always built interface specifications that detail which information flows between system components. With a publish-subscribe information bus, ***the interface is the design***; interface specifications can essentially be directly implemented. (See figure 2 below). This data-centric design technique eliminates numerous intermediate steps, and with them all the overhead and opportunity for error they entrain.

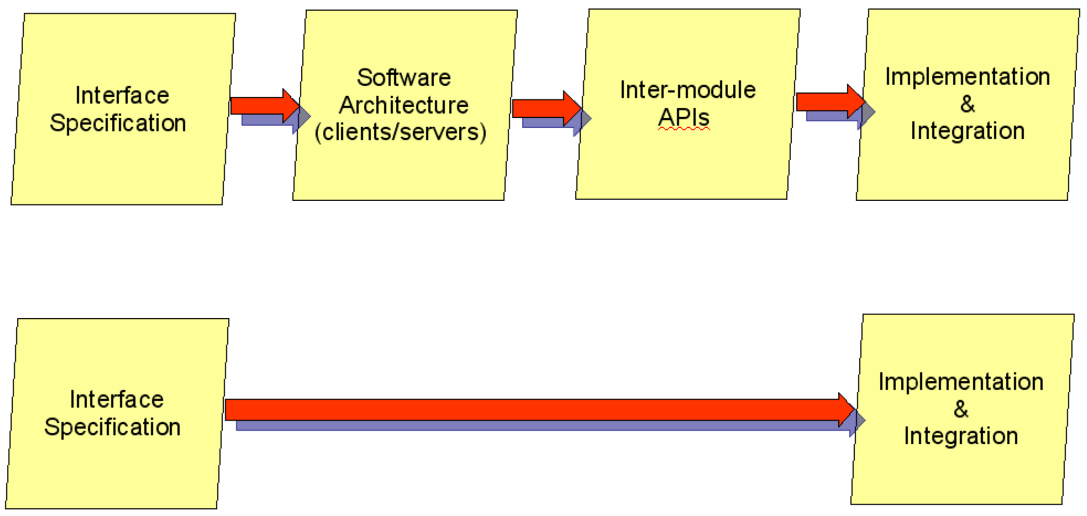
Interface Specification
Software Architecture (clients/servers)
Inter-module APIs
Implementation & Integration
Interface Specification
Implementation & Integration

Figure 2: Conventional vs. Data Centric System Design

Networking middleware

Networking middleware is a code layer above the basic TCP/IP stack that implements sophisticated data transfer and management models. Networking middleware consists of software agents that shuttle information between different components of complex, distributed applications.

There are many types of middleware. Distributed Hash Tables and peer-to-peer document sharing, including technologies like BitTorrent and Kazaa, create a distributed source for individual files. They were first widely used (or abused) by music sharing services. These technologies are massively scalable sources of relatively low-bandwidth data. They make virtually no attempt to keep the data consistent throughout the network.

Transactional systems, designed for applications like banking, offer guaranteed integrity and persistent data, even in the face of system failures. However, high-performance scalability is a challenge.

Data distribution and messaging systems strive to update multiple nodes with rapidly changing data at speeds measured in microseconds. Technologies include the traditional “middleware” such as CORBA, message queues like IBM’s MQueue, and publish-subscribe systems like JMS. While some of these older middleware solutions deliver reasonable data transfer performance, most target smaller systems of only a few hundred nodes, and leave issues of data consistency as an exercise for the application.

The emergence of DDS: the real-time publish-subscribe middleware standard

Embedded systems connect devices; since devices are faster than people, these networks require performance well beyond the capabilities of traditional middleware. Hundreds of embedded designs have now been completed with advanced publish-subscribe middleware. This experience drove the first accepted standard that targets real-time distributed systems, the Object Management Group’s Data Distribution Service for Real Time Systems (DDS) Standard.

DDS is sophisticated technology; It goes well beyond simple publishing and subscribing functions. It allows very high performance (tens of thousands of messages/sec) and fine control over many quality of service parameters so designers can carefully modulate information flow on the network. A detailed description of DDS is beyond the scope of this paper; see www.omg.org. However, it is based on very familiar concepts. We present a high-level analogy here.

DDS is like a neighborhood news service for computers. It puts reporters at every “house” and delivers the stories they write to every other house that cares to subscribe. Each house has fine control over what news it gets when. You can subscribe to every single update on every topic or weekly summaries of just one. You can have reliable delivery directly to your iPod, or have them just toss a paper on your doorstep when they can. You don't know or care where the stories come from, so multiple reporters can back each other up to make sure you always get your morning traffic report. You can even set deadlines for delivery, ask to be notified when your neighbors are on vacation, and much more.

Of course, this is just an analogy; DDS does not target neighborhood or inter-human communications. DDS targets high-performance networks. It can deliver tens of thousands of "stories" a second to any or every “house” on your network. The “stories” are data updates of any type; “houses” are any computer or embedded device. The “fine control" options are Quality of Service (QoS) parameters that fine-tune delivery requirements. QoS enables a dynamic contractual negotiation between every data publisher and subscriber to determine and set up access to a particular dataset under a specified delivery terms.

This infrastructure fundamentally changes how easy it is to design and set up a highly-connected real-time networked application. Complex, distributed, information-critical applications are the next generation of computing. The impact on many of today's real-world applications will come much sooner and be just as transformational.

What makes a system real time?

Let's take a second to examine the challenge of working in real-time, especially for distributed systems.

There is a famous Stanford PhD qualifying-exam question that asks, "Why are clear things clear?" It's a trick question, because there is no property of materials that yields clearness. Things are clear because they don't absorb, reflect, or scatter light. Fundamentally, they are clear because they fail to be unclear.

Systems are real time for a similar reason: they fail to be non-real time. Consider a control system that must respond to a sensor (e.g. a temperature rise) by changing some actuator (e.g., closing a valve). To respond to the stimulus, the sensor must detect the stimulus, that detection must be reported to peripheral hardware, the peripheral must interrupt the CPU, the operating system must process the interrupt, the application must calculate a response, the response must be passed to the actuator, and the valve must close. If each of these steps takes a fixed amount of time, the system will be deterministic. If those times sum to a small enough value, the valve may close before the boiler overheats, and the system can be called real time. If any of these steps takes an unknown or excessive time, the system will fail to be real time.

If the sensor and actuator are connected over a network, the chain of events minimally includes creating a network message, passing that message through the network stack, putting the message through some transmission medium, receiving the message at the other node, passing it back up through the stack, and interrupting another processor that controls the valve. Even this simple one-to-one system has many more opportunities to fail to be real time.

What makes distributed middleware real time?

Complex, distributed real-time systems present the interesting new challenge. Throwing a network into the mix complicates things greatly. Network hardware, software, transport properties, congestion, and configuration affect response time. "On time" may have different meanings for different nodes. Even the simplest question of all, "what time is it?" can be hard to answer in a distributed real-time system. Although it may be tough, solving the real-time challenge across a network is critical. It is the key to truly pervasive data.

So, what makes middleware real time? The easy answer would be that middleware is real time if it can always process requests in a sufficiently-short deterministic timeframe[1]. Unfortunately, this is rarely possible or even definable in distributed systems. Most designers cannot assume a reliable or strictly time-deterministic underlying network transport, as most cost effective real-time systems must operate without these luxuries. Most architects do not know the real performance bounds of their systems. Usually, you know only that the network transport seems to afford the raw performance to succeed if properly managed. The key to successful distributed real-time middleware is to optimize those last two words: “properly managed”. That discussion is beyond this paper...

Data-centric development in action

Data-centric development methodology is not some speculative future technology – it is already in use today in a wide variety of applications. It is typically employed where the increased complexity of the application and the magnitude of the system design task makes a move to a simpler and more flexible development environment an irresistible proposition. Such systems also benefit from the enhanced performance and scalability inherent in this data-centric design approach. Let us look at two recent – and very different – examples.

Tokyo's Highway Line system consists of a central information-control center and hundreds of information kiosks and displayed scattered along the city's highway system. They needed a low maintenance, high reliability communications system that was sufficiently robust for delivery of constant updates to the kiosks. The drivers and passengers who are stopped at the parking area rest stops are able to get information on traffic conditions, projected arrival times to particular locations, alternate routes, and enforcement points where traffic is being redirected or controlled due to obstructions in the roadways caused by construction or accident.

The size and complexity of such a widely distributed real-time system pose a number of specific design challenges, notably:

(1) how to provide reliable real-time information to commuters about arrival times, traffic problems, changes in schedules, potential problems and alternate routes;

(2) ensuring the provision of information to transit officials and employees on a timely basis;

(3) how to ensure delivery of this information on transmission links that vary in bandwidth, signal to noise ratio, and physical location;

(4) how to develop, operate, maintain and eventually upgrade a complex system running on a variety of computer server and client platforms with diverse hardware and software incompatibilities.

[1] Of course, none of the rest of the system can fail to be real-time. Thus, for instance, for maximum determinism, you need a real time operating system and a sufficiently fast, unloaded network. Other sources have covered these issues in depth.

By using a publish-subscribe model based on RTI's implementation of DDS, the developers of the Highway Line network designed a system using asynchronous message passing between concurrently operating subsystems, connecting a variety of anonymous information producers (central office) with many other equally anonymous information consumers (kiosk and terminals).

In such a system, when a data producer – the server - publishes some data on a topic, all the consumers – the kiosks and terminals - subscribing to that topic receive it. The data producers and consumers remain anonymous, resulting in a loose coupling of sub-systems and a robust service that decouples participants from one another, provides location transparency, and the flexibility to dynamically add or remove elements.

In addition to the benefits of performance and flexibility inherent in the data-centric publish-subscribe model, what also appealed to the Tokyo developers was the speed and efficiency of the development process. Unlike the typical client/server link, DDS essentially provides a connectionless API. Thus it does not require that the system designer get involved in the underlying communications protocols. The developer only needs to tell the system what the bandwidth constraints are, the information needed at each node, what actions needed to be taken, when to send it or to receive it, and what is required in response.

Instead of a direct, active link to a server in which the client is required to query for updates, RTI’s DDS implementation is very information centric, and does not require such active linkages, which require constant updating. This allowed the Tokyo system developers to simply tell the DDS API what information is needed at which terminal and on what schedule. As a subscription protocol, the system designers can designate beforehand the quality of service and the delivery profile rather than negotiating this each time a transaction is initiated.

The flexibility of the publish-subscribe framework and the fact that it is independent of the underlying server, terminal or network hardware or software configuration also gives the developers of the Tokyo Highway Line network the ability to be more open ended in relation to future extensions of the system in terms of number of terminals, the underlying hardware or OS, the physical network and the bandwidth required.

Another, and perhaps even more challenging, application involves the use of RTI's DDS implementation in the US Navy's ambitious program to develop its Sea Power 21 concepts in support of Joint Vision 2020 for airborne early warning surveillance, battle management and Theater Air and Missile Defense (TAMD).

As part of this system, the US Navy's E-2C Hawkeye aircraft provides all-weather airborne early warning, airborne battle management and command and control functions for the Carrier Strike Group and Joint Force Commander.

The newest variant of the E-2C (Hawkeye 2000) with its new mission computer, improved radar displays and Cooperative Engagement Capability (CEC), combines with the shipboard Aegis weapon system to provide an enhanced area defense system and will form the cornerstone of future sea-based TBMD.

RTI Data Distribution Service has been used in the development of the upgraded Hawkeye project, and has been selected for use in the next generation of the Aegis system, due for deployment in 2008.

Figure 3 below shows a real-world example taken from the design of the Hawkeye aircraft. Again, data-centric thinking transformed the whole architectural approach in this demanding "hard real-time" application.

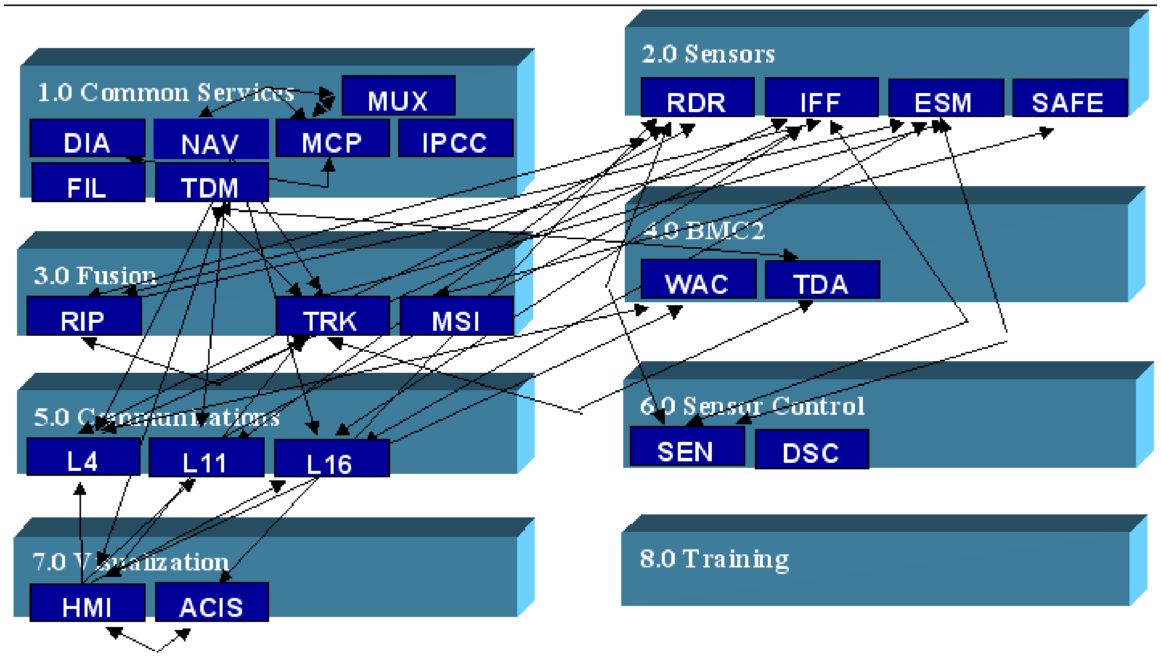

Figure 3a: Functional Design

Functionally-oriented software modules must talk to many other modules. Grouping into functional clusters does nothing to change that reality and ease software integration.

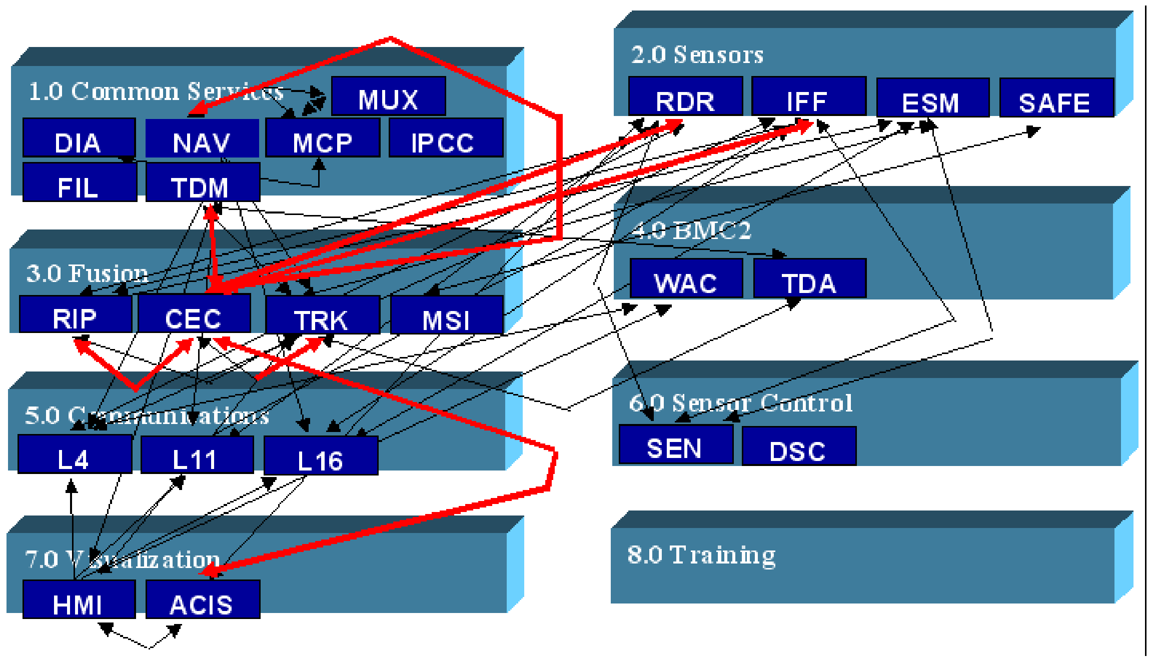

Figure 3b: Integration
Adding new functionality cascades integration re-work across many modules.

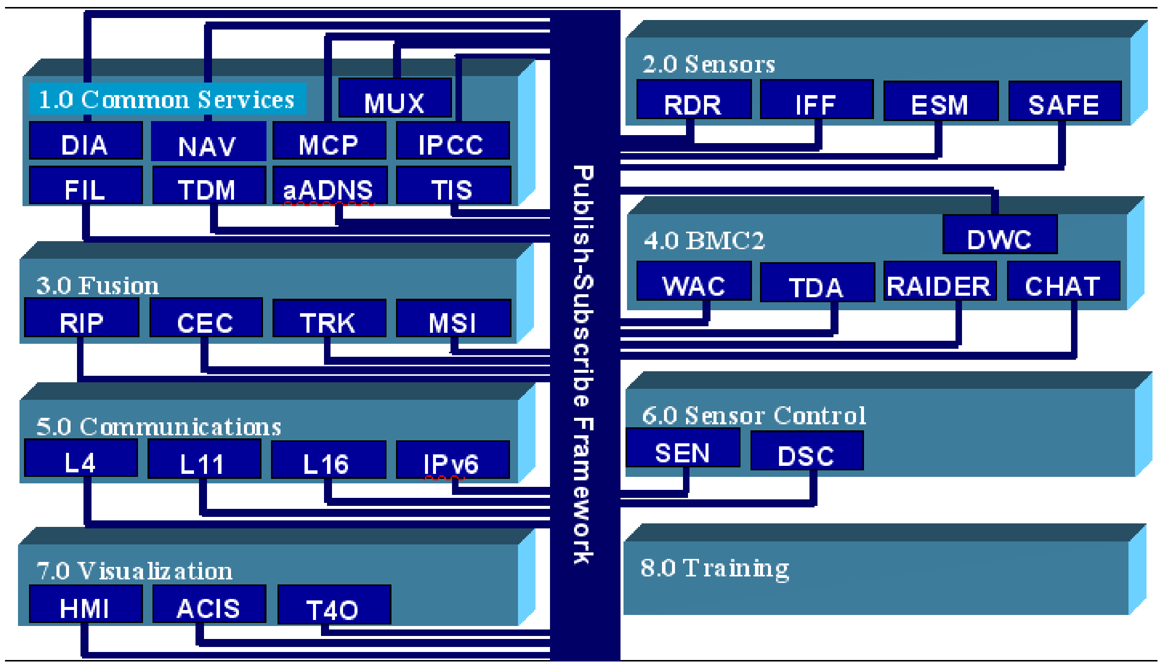

Figure 3c: Publish-Subscribe Design
Publish-subscribe architecture simplifies data communications, greatly easing integration.

What does the future hold?

There is a new generation of connected systems in sight. Data-centric architectures will change the world by making information truly pervasive. Pervasive information, available in real-time, will enable much more capable distributed, data-critical applications.

Real-time middleware is the technological key driving the data-centric transformation. The next exciting step is merging information distribution, storage, and discovery into a pervasive data model. These technologies are evolving rapidly. They already satisfy many of the real-time performance requirements of embedded systems and are developing to integrate high-performance data access to the enterprise. Although there are many challenges in performance, scalability and data integrity, a data-centric, pervasive-information future is coming. Soon.