What Is Real-Time SOA?

By Stan Schneider, Ph.D.

Real-Time Innovations, Inc.

385 Moffett Park Drive

Sunnyvale, CA 94089

June 2010

Synopsis:

Integration effort and risk grow dramatically with system and team size. Connecting large systems developed by independent teams at different times requires connecting multiple systems, thus creating a "system of systems". Enterprise systems use a design called Service Oriented Architecture (SOA) to effect this integration. However, the technologies used in the enterprise do not apply well to real-time systems; they cannot handle the strict delivery and timing requirements.

So what is "Real-Time SOA"? Ideally, it is an architecture that, like Enterprise SOA, can effectively integrate large systems of systems. However, unlike Enterprise SOA, Real-Time SOA must handle demanding real-time systems, connecting them with each other and with enterprise technologies. The Object Management Group (OMG) Data Distribution Service (DDS) standard provides the right substrate. However, as a peer-to-peer protocol, DDS cannot automatically mediate between multiple systems. With the right technology to route between DDS systems, we can integrate multiple applications built by different teams. Adding in adapters to other protocols and enterprise technologies allows connecting much larger systems.

This paper outlines how to apply SOA principles to create a fast, distributed system of systems based on DDS. The system can include both real-time and enterprise-class functionality.

How critical is integration?

On Nov 2, 2006, the Navy intentionally sank the AEGIS cruiser *Valley Forge*. It was the 4th Ticonderoga-class AEGIS cruiser to be built, in service for only 18 years. It was designed to serve at least three decades.

So what happened? The hull was sound. The engine and major components were serviceable. However, the system software could not integrate new technology and modern weapons. Shockingly, upgrading the software cost too much to justify the continued existence of a billion-dollar asset[i]. So, down it went.

How could this happen? Integrating complex systems is difficult, made even more so by the challenge of coordinating teams working on different subsystems. In the past, the only way to make this work was to have the entire system built under the direct control of a single organization, known as "stovepiping". Stovepiping is expensive, and worse, it does not encourage designs that can evolve. Thus, integration and maintenance costs balloon. In the case of the *Valley Forge*, those costs were more than the ship was worth. And the *Valley Forge* is hardly an isolated case. Integration costs for real-time systems are out of control in many industries.

This cannot continue.

The integration problem

Integration effort and risk grow dramatically with system and team size. Very small teams require no particular effort beyond informal communication. If the team grows to ten programmers, they need weekly meetings to review progress, interfaces, and design changes. A team of a hundred must divide into groups. Design documents, reviews, and project management become important.

By the time a team grows to 1000 people, it faces all the stresses that dominate large-company projects. The project may even include multiple organizations. Now, the interfaces must be negotiated in detail. Technology selection becomes a political nightmare. Full-team design reviews become the critical milestones. Integration dominates the system cost, especially if you include the overhead of processes and pre-work. Even minor changes require careful documentation, schedule slips, and additional payments.

If you add in the time factor, things get worse. Large systems are developed and maintained in pieces over time. That means that old technologies and designs must somehow interface with new versions. If the original design didn't do a good job defining the interfaces, this can be fatal. Mistakes like creating "custom" versions of software (a real temptation with open source) prevent upgrades to the latest versions of fundamental infrastructure. Without the latest platforms, all dependent technologies must be ported or adapted. "Reuse" is a never-realized dream. Instead of upgrading gracefully, contractors charge hundreds of millions for seemingly simple refreshes.

Simply put, integration costs dominate large system development. It gets worse, not better, during the maintenance phase.

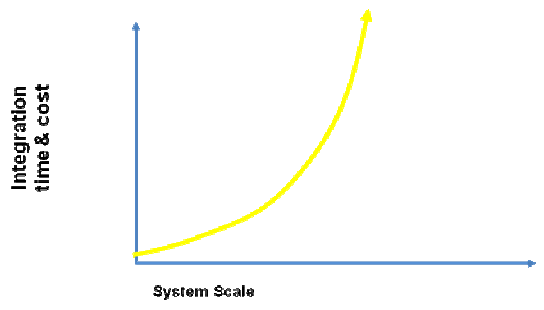


Figure 1: Integration Costs Explode as Systems Grow

Integration grows dramatically with system size. The cost and risk of building and maintaining large systems is completely dominated by integration.

The fundamental problem: coupled development

So why is this? The root cause is quite obvious. Systems are made of modules. Different teams make the modules. As teams get large, communication (and agreement) gets much more difficult.

Thus, large teams can only effectively build systems from independently designed, independently implemented, independently managed modules that can evolve independently over time.

But, that's easier said than done. It implies that we can integrate modules not designed to work together. And that's Not Easy.

Integrating real-time systems

Our first challenge is to put together high-performance real-time distributed systems from independent modules.

Figure 2 shows typical designs with traditional "client server" or "remote object" technologies, such as CORBA, ODBC, RMA, OPC, etc.. The development process is to first determine functional boundaries, then design access methods and interfaces for each piece of information. The data is "hidden" behind object abstractions. Data requirements of each device or processing endpoint are considered only as an afterthought. For larger systems, developers then decide what information will reside on which servers, and where to put the servers and clients.

This design assumes the network is relatively static, servers are always present and accessible, the server/client relationships are clear, clients know where and—most importantly—when to request data, and all nodes have similar delivery requirements. It also assumes a "synchronous" processing model; clients request service and then wait for a reply. This "server centric" design quickly breaks down as complexity grows. It's hard to add new data flows (red lines). Every server is essentially coupled to every client. Unless the access methods happen to provide everything needed, every single interface the new component touches must be reworked.

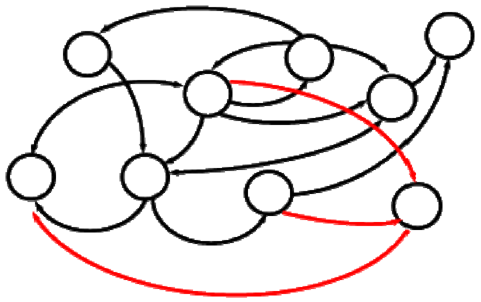


Figure 2: Traditional OO Designs Lead to Tightly Coupled Networks

Traditional technologies require developers to define access methods for each function, and then implement servers and other special interfaces. This leads to "spaghetti" network design and "stovepipe" systems[ii].

Data-centric design, in Figure 3, offers an alternative. With data-centric design, developers specify only the data requirements—inputs and outputs—of each subsystem. This is the exact opposite of the above approach; rather than hiding data, designers specify exactly what information flow affects each module. Integration middleware discovers the producers and consumers of information and provides the data immediately when needed. This "data-centric publish subscribe" design eliminates coupling and greatly simplifies integration of systems with demanding or complex data-sharing requirements. Driven by the rapid adoption of the Object Management Group (OMG) Data Distribution Service (DDS) standard, many fielded systems are using this approach. DDS is the planned future integration technology of most military systems in the US and its allies. Its use is also spreading into many other applications, including financial trading, automotive driver assistance, air-traffic control, and electrical power generation.

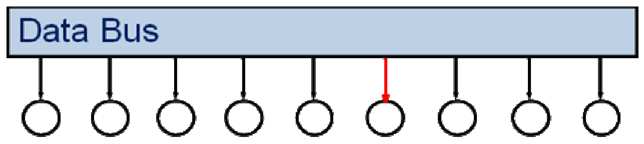


Figure 3: Data Centric Design Implements an Abstract "Bus"

DDS Data Centric Publish Subscribe specifies the interfaces and information model crisply. With clear interface specifications, components can be plugged into an information bus.

Plugging in components sounds simple, but it is not trivial

The key benefit of DDS is that you can "plug in" new modules without redesigning other interfaces (red line in the figure). That looks simple, but it is anything but obvious how to do that. Having a pluggable interface assumes that the "bus" specification captures all the key real-time constraints: What is the data? When is it available? When is it needed? What happens if it's not on time? Are there backup sources? How are they engaged? It also makes a huge assumption that adding more load will not affect the overall system performance; if it does, other components can fail even if there is no direct connection. That's an insidious type of indirect coupling.

At a high level, participants on a DDS bus just subscribe to the information they need and publish what they produce. Information is identified by "Topics", and is strongly typed for safety and easier integration. The middleware automatically "discovers" new entrants. Types are communicated before information flow begins, so all modules know what they're getting (called being "content aware"). And because data can flow without servers or synchronization delays, the system is blazingly fast; DDS is easily the highest-performance middleware standard[iii].

But perhaps the key to the success of DDS is that it crisply defines an information model. Fundamentally, if you define the information model correctly, then components can specify their needs *at design time*. At runtime, they then produce and consume via the model, implemented by the middleware. The modules are not directly dependent on each other. This is the key to avoiding coupling. The components adapt to the model, not the other way around. Other messaging technologies (e.g JMS) can't do this because they don't define an information model[iv].

DDS defines the information model with over 20 "quality of service" (QoS) parameters. QoS settings stipulate if a publisher can supply reliable information flow, how fast it can supply data, and when you will find out if it's up or down. A subscriber can specify exactly what it needs, even down to filtering out updates based on timing or the content of the message. The middleware then looks at requests from "plugged in" subscribers, and decides if any publishers can satisfy those requests. If so, it establishes "contracts" between the publisher and subscriber and starts sending data. If not, it reports an error. An example contract might be, "The publisher will update the subscriber at least every millisecond, and the communication must be reliable, so retry to repair transmission errors. If the publisher fails, switch to a backup."

That's a small example. The full set of DDS QoS parameters can integrate hundreds of very complex, demanding applications. And because the middleware is fast, the specifications can be set to satisfy very demanding real-time requirements. Fielded DDS systems send up to millions of messages per second, with latencies measured in the tens of microseconds, through millions of publish-subscribe pairings. Designers can specify timing requirements, supporting systems that have deterministic delivery needs. Reliable multicast ensures that new "plug ins" don't burden the system. It scales well; applications comprising up to 1000 computers are deployed.

DDS particularly excels when the application needs non-stop reliability, such as a in a battle management or an electrical grid system.

Radar example

QoS contracts are made on a per-data-flow-path basis. So, for instance, consider a high-speed radar application that publishes "tracks" to many different subscriber applications. It must send updates 2000 times a second to a weapon system trying to track incoming missiles. For this subscriber, the latest data is best, so if a sample is lost due to a transmission error, don't waste time retrying old data. At the same time, a logging system needs the radar to reliably deliver everything being tracked for later analysis. At the same time, this data should be multicast to 50 operator screens, but they can only take 10 samples a second. Each screen may further select a region of interest, e.g. display only tracks within 5 miles, coming towards me, and only things that are not identified as friendly.

DDS can handle all this and more. All at the same time. And, because these issues are being managed by the middleware, the application can simply focus on logic and processing.

Of course, all these issues are handled behind the scenes by the middleware. It doesn't impact the application. You can imagine how much this simplifies application development and integration.

Now, suppose all this is working and deployed. And then we need to add a new capability. For instance, let's use our radar to coordinate air traffic in our local area. Since every service has crisply defined its capabilities, we don't require any change to any components. The new application simply plugs in and works.

Of course, this assumes performance is not affected by loading. DDS implementations vary in their ability to scale, but those that implement direct peer-to-peer reliable multicast suffer very little performance hit with higher offered load. Implemented well, DDS is orders of magnitude faster than other middleware designs. And the "QoS contract" concept adds a layer of failure detection. If new load does cause an overload, the middleware will immediately notify the other applications that their contracts were violated. That helps find design issues early, during system test. It also allows live systems to take effective remedial action, should this occur during operation.

Integrating systems of systems

So, to summarize the above, DDS's data-centric publish subscribe connects real-time systems. It's the right architecture for these applications; it defines interfaces, timing, and QoS control. Everything is decentralized for redundancy and speed.

However, it's not the whole story. Large systems that must be developed and maintained over many years must be able to combine without any premeditated coordination. Fundamentally, for instance, to plug into the bus, all the components on a DDS network (a "domain") must agree on topics and types. Even if everyone uses DDS, different groups will choose different topics and types. And even a single group's design will change over time. So, to integrate these systems, we at least have to merge multiple, different DDS domains. Moreover, not all systems use DDS; we must also merge many different protocols, legacy systems, and enterprise technologies. DDS alone can't handle that.

Fundamentally, DDS decouples modules, but not development. It's a great substrate. But it's not enough.

SOA: Decoupled development in the enterprise

Let's take a step back and look at the successful enterprise technology for integrating systems of systems: SOA.

SOA stands for "Service Oriented Architecture". A SOA is a system architecture that builds a complex system from independent "services".

If you are familiar with the term "SOA", you probably know it as the design behind web services. Web services and SOA are phenomenally successful in the enterprise. Because they enable decoupled development, SOA and web services dominate today's enterprise architecture. In fact, it's not much of an overstatement to say that SOA *is* today's enterprise architecture.

SOA, implemented as web services, integrates the enterprise. Services are independent entities that implement business functions. Applications that execute as loosely coupled services can be developed independently and linked later. That's how a web store can charge your credit card and arrange shipping via multiple vendors. Those applications weren't designed together. They were integrated from independent modules. The web obviously can plug and play components built by different teams at different times, even using different technologies.

How do web services work? A detailed description is beyond this paper. However, fundamentally, the key features are crisp interface definitions (WSDL files), known places to get those interfaces (UDDI servers), and shared type definitions (XML).

How does this compare to DDS? Although the implementation is very different, the analogy to DDS is striking: both support interface definitions (WSDL v QoS), interface service (UDDI v DDS topic discovery), and content-aware type sharing (XML v DDS type discovery). These are the core features of a distributed integration substrate.

The enterprise integration point: ESB

But enterprise SOA has something that DDS does not. At run time, enterprise SOA leverages a key integration technology: an application called an Enterprise Service Bus (ESB). An ESB runs on a single logical machine; it is not a distributed concept, despite the "bus" in its name. The ESB's function is protocol interchange. Fundamentally, it "routes" incoming messages to the right service. It can speak many protocols, e.g. SOAP and JMS. It can also translate types between services (called "mediation"). With these abilities, an ESB can connect independently developed applications, even if they don't share types or know where to route their requests. See Figure 4.

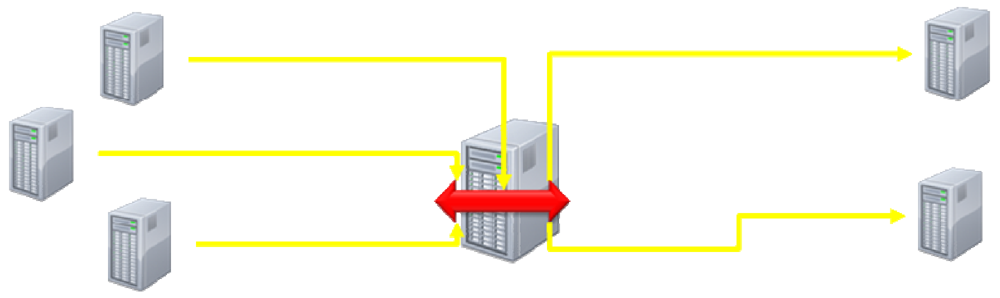


Figure 4: Enterprise Systems Leverage an ESB for Integration

Enterprise SOA uses an Enterprise Service Bus (ESB) to route requests to the right service. It allows development of complex systems built by independent teams at independent times. The ESB can connect them at runtime.

To integrate independently developed systems, DDS needs the equivalent of an ESB.

Integrating real-time systems of systems

So, if our goal is to integrate real-time systems with each other and with enterprise systems, this analysis brings up three obvious questions:

- Can we use enterprise SOA technology to integrate multiple real-time systems?
- Can we connect a real-time system to an enterprise system with enterprise SOA?
- Can we use SOA principles to extend DDS to integrate real-time and enterprise systems?

Let's take a look at each of these questions.

Can we use enterprise SOA to integrate real-time systems?

The first obvious question is if we can just use enterprise ESB technology for real-time systems. It certainly can integrate many technologies.

Unfortunately, web services and ESBs are woefully inadequate for integrating real-time systems. They have no concept of timing control. There's weak QoS control, and the parameters are wrong for demanding real-time applications. Worse, enterprise SOA systems are centralized for easy deployment, administration, and management. While the net may look like a "cloud", the key components like ESBs fundamentally live in data centers, and you can't put a data center on a battlefield (at least, not if you want to win).

And then there's performance. Enterprise SOAs are just not fast on the scale required to integrate a combat system or other "real-time" applications. Some web services vendors call their web services “Real-Time SOA”. What they mean by that is a system capable of responding to web transactions in a second, or perhaps push data out every tenth of a second or so. That’s fast compared to human response time. But, it can’t control a radar system. Real-time systems must send thousands or millions of messages a second with sub-millisecond latency. Web services are way (way) too slow.

And even a fast web service wouldn’t work. Web services don’t model time, so you can’t request response within an amount of time or find out if it will be late. Real-time is more about consistency of delivery (“determinism”) that just speed. Radar systems, for instance, flat-out fail if they deliver data a little bit late. Users will wait a little longer while the system responds. Incoming missiles won’t.

Enterprise SOA fails in real time systems.

Can we connect a real-time system to an enterprise system with enterprise SOA?

So, if enterprise SOA can’t connect real-time systems, can it at least connect real-time systems to enterprise systems? There are certainly many candidate enterprise technologies that could talk to DDS. Apache Camel, for instance, supports 1000 plug-in protocols. If you want, for example, to connect a tactical real-time system with its enterprise support environment, why not just plug in DDS and go?

This is appealing. In fact, RTI tried it. It makes a nice demo…but flat-out failed in real application. Fundamentally, it doesn’t solve the real problem. To understand why, let’s look at some of the important differences between real-time and enterprise systems.

Real-Time systems:

- May send millions of messages/second
- Need latency measured in the microseconds
- Must be distributed for redundancy and speed
- Require strict QoS and timing control
- Use fast binary data formats

Enterprise systems:

- Send at most hundreds or a few thousand messages/second
- Expect latency of at least many milliseconds, or even seconds
- Must be managed centrally (administration, backup, etc., are all in data centers)
- Are burdened with layers of business logic support
- Use heavyweight wire data formats like XML

For these and other reasons, enterprise technologies do many things that Don’t Make Sense for real-time systems. For instance, ESBs normalize messages. Camel translates everything to Java objects and then back. That’s slow for little benefit in a real-time environment. If you expect the ESB to perform key functions like mediation for the real-time system, it will quickly be overwhelmed.

So, bridging real-time systems through an enterprise ESB fails for many good reasons. It's slow. You lose all timing & QoS information. It imposes single points of failure and choke points. Bottom line, it's fundamentally the wrong end to fix problem. It just doesn't make sense to connect the real-time system directly through an ESB.aragraph header

Can we use SOA principles to extend DDS to integrate real-time and enterprise systems?

So, DDS is missing the key ESB-like functionality, "routing". Can we do that in a way that's compatible with the reality of real-time demands? That's an interesting proposition. It defies the "conventional wisdom" of just slamming all the integration load onto the ESB. But as we've seen, DDS is also a powerful integration substrate. And it has the key advantages of being fundamentally distributed and fast. So…can we build a fast, distributed analog of an ESB?

Real-time Routing

Connecting DDS domains

Let's consider the simplest problem first: say we have two DDS applications, maybe a radar system and a display system. Now, obviously, it would be best to design them together, use a single DDS bus, and connect cleanly through shared types, topics, and QoS parameters. Then we could develop a high-performance, reliable platform by starting with solid data-centric design principles that encompass the whole application.

But, let's assume that's not possible in this case. These are big systems involving many applications each. They were originally developed for other uses; perhaps we are "borrowing" the display subsystem from a ship architecture, and merging it with a radar from a ground-based anti-missile system. Perhaps these systems are modules that must be used in more than just our one application, and those other uses have other requirements. Or maybe the radar is a very new technology that we're trying to integrate into our ship, to keep it affordably competitive.

So, our challenge is to reuse Big Modules that were developed independently, even by other teams or other companies at other times. As every program manager knows, this is a common desire. (And the customer rarely understands why it's so hard or expensive.) Unfortunately, reuse is usually an unfulfilled dream (hallucination); integrating two large systems that were not designed to work together is a famously frustrating exercise.

Routing service

How can we effectively connect our two independent DDS domains? What would such a routing service need to do?

Without a routing service, the only way to connect domains is to build a custom “bridge” that knows exactly what data to pass, subscribes to it, translates it for the other side, and publishes it with the schema and topic names that the other side expects. This is doable. However, custom bridges are expensive to write and maintain.

Our routing service clearly needs to bridge packets between the networks. But to do routing generically, our service needs to provide two key functions in configurable form: guards and translators. Then, it needs to connect to both domains, find the right data, and route it to the right destination.

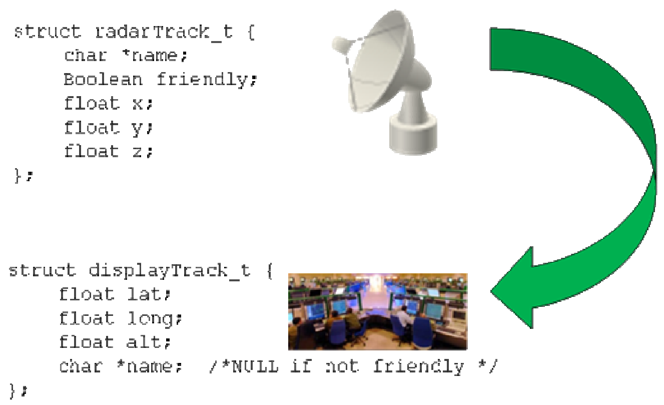
Guards

Guards decide what our subsystems will export. Obviously, the display does not need access to every single piece of data flying around in the radar system. It only needs tracks (in our simplified example). The guard’s job is to select just that information and expose nothing else. It can also change the QoS of the exported information; perhaps we want to control our commitment to external systems. So, a guard is a filter that intelligently allows only the right information through to the other side.

Importantly, the guard is also an obvious security point. Because the routing service controls all information entering or leaving the system, it is a key opportunity to provide security. That is critical, but beyond this paper.

Translators

Secondly, we need to transform languages between the systems. Our radar and display both understand the concept of “tracks”, but suppose the radar publishes tracks in X-Y-Z position relative to its location, while the display expects latitude and longitude (see Fig 5).



```
struct radarTrack_t {
     char *name;
     Boolean friendly;
     float x;
     float y;
     float z;
};
```

```
struct displayTrack_t {
    float lat;
    float long;
    float alt;
    char *name;  /*NULL if not friendly */
};
```

Figure 5: Translation and Guarding are the Key Integration Functions

Any system that must connect independently developed components must translate between the data types on each end. It also must know what information to expose. The only alternative is an expensive rewrite of one of the end applications.

This is indeed a very simple example. Real track data structures can have hundreds of fields. The concepts are unchanged; the translator must match the topic names and type schemas on both ends.

Tap into discovery on both sides

So how can we do these steps generically? The routing service must be able to find the right information within the network and publish it to the other side. Since our routing service can tap into DDS automatic discovery, finding information is manageable. The routing service must also understand and process topics and types. Since DDS is content aware, these functions are implementable. The translators become simple scripts or plugins for each schema processed. Guards become configuration scripts that specify which topics to translate. As a bonus, DDS content awareness means that the routing service checks all types at runtime; the system can thus work safely in an environment where the schema may change. RTI has implemented such a routing service; it is configured via XML scripts.

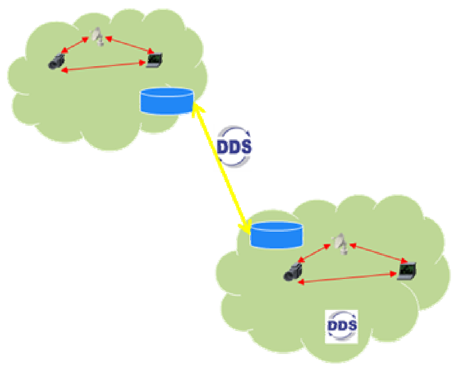


Figure 6: RTI Routing Service Connects Domains

RTI Routing Service connects two DDS domains. It implements both guarding and translation. The guard selects which data will be exported from the domain. The translator matches topics and schema between domains. It can find the data it needs via DDS discovery.

Leverage easy distribution to make it scale

Now, suppose we want to integrate multiple DDS domains. An ESB-like approach would imply that we should expand our routing service to process multiple DDS data streams. However, with the DDS bus available, there's no need for a monolithic ESB. We can just instantiate more routing services, and distribute them around the system as needed. That's a much faster and more robust design. We can even have multiple routing services between single pairs of domains for redundancy.

Build DDS into a real-time distributed ESB

This last point is suggestive of a bigger solution. Of course, Routing Service (RS) can do more than just connect using the DDS protocol. We can extend it with plug-in protocols. Then it can connect DDS domains through TCP, for example. This allows DDS to cleanly traverse firewalls. The plugin can encrypt traffic. Thus, we can securely span WAN networks.

We can take this even further. The RS can connect DDS to non-DDS systems. With plugins, the RS can be a programmable bridge to any protocol, e.g.: Link 16, STANAG 4586, JMS, C37.118, compressed DDS streams, and many others. Each RS, running a plugin, can integrate another protocol into the system. Of course, DDS makes it easy to have many of these. Each can run on a separate node. The result? An efficient, distributed integration design.

Besides combining protocols, DDS can be used to integrate other technologies. For instance, RTI has DDS services to integrate databases, web services, and tools like Excel. A JMS API on the middleware can offer clean and easy integration with enterprise technologies. The JMS API with guarding can also interface more cleanly—without the intermediate "normalization" step—with enterprise ESBs (like Camel).

Together, we could call design a "Real-Time Service Bus". It fulfills the same function and delivers the same benefit as an Enterprise Service Bus, except that it's fundamentally distributed, fast, and timing aware. Modules can live on any node to integrate any technology. The Real-Time Service Bus provides the integration "point" that DDS was missing, in a distributed architecture.

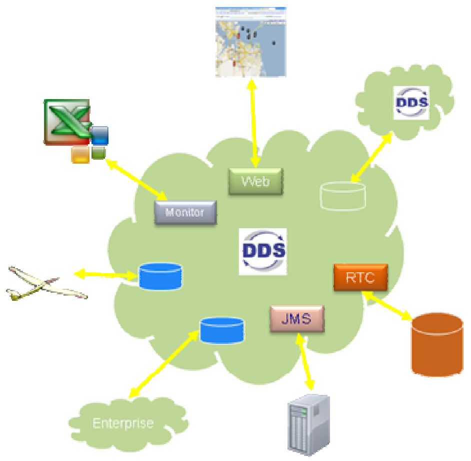


Figure 6: A Real-Time Service Bus

By adding protocol plug-ins and enterprise technology adapters, DDS becomes the substrate for a Real-Time Service Bus that functions as a "distributed ESB". It delivers the key benefits of real-time SOA without the disadvantages of web services implementations.

So is that Real-Time SOA?

This brings the key benefits of a SOA to true real time

The architecture outlined here is certainly a service-oriented architecture (SOA) by the formal definition. With RTI's powerful Routing Service, designers can build real-time systems from many independent services.

More importantly, it delivers the key decoupled-development benefit. Services, even entire DDS or other applications, can be developed by separate teams, at different times, with different designs. The translation function allows them to then be integrated later. If a component evolves (and this is key) only the translation configuration needs to be adjusted.

We will repeat that the innovation here is vastly different from what enterprise vendors call "Real-Time SOA", which really means faster web services. The DDS-based Real-Time Service Bus is a SOA that truly integrates a real-time system of systems. And, it satisfies demanding performance, reliability, and control needs. With protocol plugins and other technology interface components, it takes this a step further. It allows the development of complex systems that include both real-time and enterprise components. It integrates large real-time systems with each other and with enterprise systems.

Even lying on the bottom of the Pacific, the *Valley Forge* can still serve software designers. Its lessons? Integration is key. Architecture enables integration. Real-time systems require real-time technology. And emerging real-time technology can deliver the levels of integration that can prevent this waste in the future. The key lesson: designing for integration from day one with the right approach is perhaps the most important consideration in building and maintaining *affordable* distributed systems.

Next steps

To learn more about RTI Data Distribution Service or to download a fully-functional software evaluation, please contact us at info@rti.com or visit http://www.rti.com.

[i] Admiral Frick, PEO IWS, at the 2006 ASNE Combat Systems Symposium

[ii] Figure modified from Raytheon talk at OMG DDS Day seminar.

[iii] All performance results in this paper are dependent on the right implementation. RTI publishes copious data and numerous studies comparing middleware implementations at www.rti.com.

[iv] DDS also allows interoperation across operating systems, languages, even DDS implementations. No other middleware standard does all that either, but that's beyond this paper.