The Global Leader in DDS

November 2010

Data-Centric Middleware

A fundamental improvement in developing, maintaining and deploying mission-critical distributed systems

Curt Schacker, VP Worldwide Field Operations

www.rti.com

US HEADQUARTERS
Real-Time Innovations, Inc.
385 Moffett Park Drive
Sunnyvale, CA 94089
Tel: +1 (408) 990-7400
Fax: +1 (408) 990-7402
info@rti.com

Executive Summary

Mission-critical distributed systems have traditionally used a *message-centric* architecture for exchanging critical data among applications across the network. This approach results in significant inefficiencies that cost projects time and money throughout the development, maintenance, and deployment cycles. In addition, message-centric systems can be complicated to implement, which raises legitimate concerns about the robustness and performance of these systems once they reach deployment.

In contrast, a *data-centric architecture,* such as that supported by the OMG's Data Distribution Service specification, represents a fundamentally different approach to distributed computing communications and exhibits none of the shortcomings of message-centricity. As such, distributed systems architected on the data-centric model can realize a step function improvement in cost, time to market, reliability, and performance. As one example, RTI Data Distribution Service has already been used to deliver these benefits on a wide variety of projects across many industries.

A Lesson in Constraints

While working at my first job after graduating from college, a colleague introduced me to a foolproof way to beat the odds and make money in Las Vegas. The idea went as follows: place a $1 bet on the roulette wheel (either red or black, but not green). If you lose, double your bet on the next spin. If you lose that one, double your bet again. Continue as such until you win, at which point you will be exactly $1 ahead. Go back and repeat the cycle, winning $1 each time through. Can't miss right? Well, even at that age I understood that the people who run Vegas are no fools; there must be a catch, and of course there is. In this case, the catch is the table limit, aided by those 2 pesky green spaces which reduce your odds to slightly less than 50-50. A $1 roulette table might have a maximum bet of $500, which may not seem like a problem. After all, you would have to lose 8 times in a row before hitting the limit, and with a 49% chance of winning on each spin, that shouldn't be a problem, right? Wrong. I quickly wrote a roulette simulation program and discovered, after playing thousands of simulated games that, in fact, this can't miss formula offers virtually no chance of winning. Sure, you can win a dollar at a time for a while, but you will hit that limit often enough to wipe out all of your earnings and then some. As I said, the folks who run Vegas are no fools.

For me, that was a valuable lesson in the importance of constraints. A seemingly can't miss solution becomes much more challenging in the face of real world constraints and limitations. This is a great lesson to keep in mind when contemplating how to approach a large, distributed and complex software project.

"It's Just Software"

Anyone who has dealt with software has at one time or another heard the expression, "Of course we can do that, it's just software." This statement is usually offered in response to the question, "Can your software support such and such a feature?", and it refers to the inherent agility of software as a medium for implementing logic in a computer system (as opposed to hardware-based logic, which is fixed). Like the roulette game, it sounds great until you consider the real-world constraints, in this case time and money. Since both of these quantities are decidedly limited (and becoming more so), software development efforts must prioritize one objective over all others: efficiency.

Efficiency[2]

Efficiency in a software context means two things:

1. In the *development lifecycle,* implementing the required functionality **with as little application code as possible,** since each line of code costs a measurable amount of time and money to develop and test.
2. In *deployment,* implementing the required functionality **with as few computing and networking resources as possible,** since hardware and equipment also cost money.

An efficient software architecture will cost far less to develop and maintain over time and will be far less expensive to deploy.

In addition, efficiency drives another critical benefit: higher reliability. A smaller, tighter program running on fewer computers and consuming less network bandwidth will be less prone to errors and therefore more reliable, an especially important consideration for mission-critical systems, wherein failure can lead to loss of life, property, business revenues or all three. Moreover, software programs are rarely static; they typically grow in size and complexity over time as new requirements emerge. What might seem like a relatively efficient approach at the inception of a project could turn into a nightmare over time as the system scales and new capabilities are implemented. It is paramount, then, that software development teams choose the most efficient strategy available for their projects, taking into account the future probable growth of the program.

Middleware and the Software Ecosystem

These days, very few software programs are written entirely from scratch. Modern projects often start with legacy code and are usually implemented by licensing a number of third party components – such as operating systems, databases, and GUI's – and writing applications that invoke the services of these components. For distributed systems (ones in which multiple computers are orchestrated to solve a given problem), one of the most critical components is the communications 'middleware' that enables applications to share information with each other over some type of network. Middleware solutions like Java Message Service (JMS) and Tibco Rendezvous allowed applications to be insulated from underlying operating system and network interface details while supporting an asynchronous communication pattern. These products implemented what is now called a message-centric approach, wherein the middleware enables computers to send messages to each other without regard to physical location and message contents (a property which in technical terms could be described as queue-based or topic-based generic message distribution). While message-centric middleware was an important step in the right direction, its opaque quality imposed enormous responsibilities on the application programs which resulted in a severely adverse impact on efficiency, as we will now explain.

The Wrong Tool for the Job

Let's use an example to illustrate the point.

Asset tracking is a common application found in many mission-critical systems. Examples of asset tracking include: tracking the progress of packages as they make their way through a commercial shipping system; tracking the physical location of vehicles in a commercial or military context; or tracking the location of supplies and projected deliveries in an assembly plant.

Asset tracking applications are often implemented on message-centric middleware products like JMS, Tibco Rendezvous, or IBM MQ. These products provide a generic interface for exchanging data-containing messages from a source to endpoints, using daemons or message brokers as intermediaries in between.

This generic messaging infrastructure imposes significant responsibilities at the application level, to wit:

- A set of messages must be defined based on the information exchanges required to support the expected set of use cases. These messages are hard to define and constantly evolve as new use cases and applications are anticipated and integrated.
- Each application must assume responsibility for ensuring that the tracking data it receives is in the expected format and contains the expected fields; a "type" mismatch can cause the program to miss important information, fail or behave erroneously if the error isn't detected. It is incumbent upon applications to notify administrators of errors.
- Each application receiving tracking messages must keep its own local representation of the state of each tracked item; there is no consensus state to which each application can refer.
- Each application must implement custom logic to notify other applications of the creation and deletion of items, and to communicate changes in their state or value.
- If only a certain subset of the information is relevant to a given application, filtering out of unwanted data must be done at the application level, resulting in poor use of network bandwidth and resources.

- If a new application comes on line, it must announce itself to the system and find the tracking information it needs, typically by registering with a server somewhere in the system.
- If a new stream of tracking information becomes available, dynamic applications must somehow be notified, determine their level of interest, and make arrangements to receive the new data.
- In situations where there may be a network disruption resulting in the potential loss of some tracking information, either the middleware must cache all messages—regardless of their resource usage or whether they will be irrelevant by the time reconnection occurs—or application logic must be implemented to manage this scenario in a more appropriate and efficient way.

All of the logic required for each of these circumstances must be implemented with application level software - *for every information exchange in the system!* Even a relatively simple system could require thousands of lines of application code to support all of these requirements. Given this state of affairs, it is not difficult to understand how a message-centric approach yields such a heavy burden on a project's software development budget or why such systems take so long to develop and maintain.

As bad as this is, we need to consider another crucial point. Numerous studies have cited the direct relationship between the size of a software code base and its corresponding propensity for errors. This relationship is logical and easy to understand: the more software we have to write, the higher the chances of introducing a programming error. Given the heavy software development burden imposed by message-centricity, we must ask ourselves if there is a better way to deliver robust, reliable systems.

A Step Function Improvement in Efficiency: Data-Centric Middleware

Data-centric middleware, such as implementations of the OMG's Data Distribution Service specification, takes a fundamentally different approach from message-centricity. As the name implies, data-centric middleware focuses on sharing data and system state information, not on use-case or component-specific message sets. Moreover, the middleware is "aware" of the data being shared between applications in a distributed system. The messages are built by the middleware to communicate the updates and changes of the system state. In other words, the messages are *not* generic as is the case with message-centricity. Instead, they are derived from the system data model. This data awareness leads to a critically important set of capabilities that are simply not achievable using a message-centric approach. Let's examine how this works in practice.

When using a data-centric approach, we first construct a data model which captures the system information model and state. This information model is not monolithic; rather it is an aggregate of smaller data models representing the state of the various parts of the system. For example, in asset tracking we would model each type of asset or "stateful" element in the system: trucks, trains, packages, people, orders, etc. The information model captures both the structure of the information and, if necessary, constraints on the content.

Once the model is defined, producers and consumers communicate through a simple, peer-to-peer, publish-subscribe interface. Applications make a change to the information model (e.g. updating the current location of a truck), and the middleware propagates that change to all interested subscribers. All of the data marshalling and unmarshalling, and much of the data validation, is taken care of by the middleware, so no special application logic is required. For example, if a subscribing application is only interested in a subset of the state that is being changed by a publisher (e.g. only the location of trucks within a geographical area, or those belonging to a specific company), it can declare that at the time the subscription is established; the middleware will then deliver only the subset of interest *with no additional logic required at the application level.*

Let's go back to our asset tracking example and examine the implications using of the data-centric approach in detail:

Using Message-Centric Middleware	Using Data-Centric Middleware
A set of messages must be defined based on the information exchanges required to support the expected set of use cases. These messages are hard to define and constantly evolve as new use cases and applications are anticipated and integrated.	A data model is defined grounded on the information needed to describe the system state: vehicles, assets, orders, etc. As new types of vehicles or assets are added, the data model is expanded to include them. Changes on use-cases or the introduction of new application components do not affect the data model.
Each application must assume responsibility for ensuring that the tracking data it receives has the expected contents and format; a mismatch can cause the program to miss important information, fail or behave erroneously if the error isn't detected. It is incumbent upon applications to notify administrators of errors.	The middleware only allows producers and consumers to communicate if they agree on the data's type. A type mismatch results in an error notification to both applications and administrators. Misunderstandings of this kind literally cannot occur, so no application logic is required to manage this scenario.
Each application receiving tracking messages must keep its own local representation of the state of each tracked item; there is no consensus state to which each application can refer.	The middleware maintains a local model of the system state for each application automatically; therefore applications do not need to keep track of a localized view.

Each application must implement custom logic to notify other applications of the creation and deletion of items, and to communicate changes in their state or value.	The middleware automatically manages the lifecycle of items and makes this information available to applications using standard, well-defined interfaces.
If only a certain subset of the information is relevant to a given application, filtering out of unwanted data must be done at the application level.	The middleware filters data as specified by the application; time and/or content filters may be declared when communications are established or modified later. No application logic is required for filtering.
If a new application comes on line, it must announce itself to the system and find the tracking information it needs, typically by registering with a server somewhere in the system.	When a new application wants to receive information that is being produced, the middleware will automatically discover this new consumer and begin making the data available to it automatically. No other effort is required by the application, and no centralized servers—with their associated costs and risks—are needed.
If a new stream of tracking information becomes available, dynamic applications must somehow be notified, determine their level of interest, and make arrangements to receive the new data.	When new information becomes available, applications are notified automatically, and interested consumers of that information simply begin subscribing to it. The middleware automatically manages all of the connection logistics. No other effort is required by the application.
In situations where there may be a network disruption resulting in the potential loss of some tracking information, either the middleware must cache all messages—regardless of their resource usage or whether they will be irrelevant by the time reconnection occurs—or application logic must be implemented to manage this scenario in a more appropriate and efficient way.	When information exchanges are unsuccessful for whatever reason, including network failures, the middleware will keep track of what data has been missed, eliminate obsolete or irrelevant data, and automatically deliver the appropriate information once the network impairment is rectified. No other effort is required by the application.

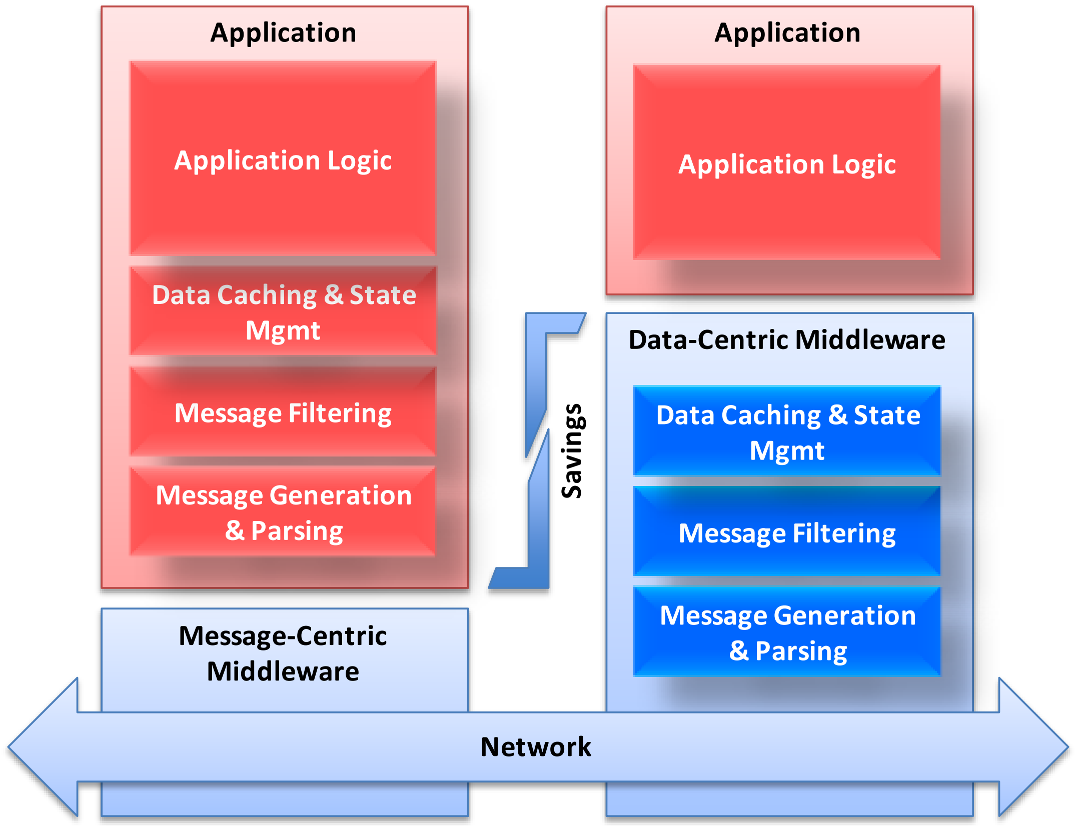
Application
Application Logic
Data Caching & State Mgmt
Message Filtering
Message Generation & Parsing
Message-Centric Middleware
Savings
Application
Application Logic
Data-Centric Middleware
Data Caching & State Mgmt
Message Filtering
Message Generation & Parsing
Network

Figure 1 - Illustrates the differences between message-centric and data-centric approaches with respect to consumer-side logic

As this example clearly shows, using a data-centric architecture can save the application team from having to write many thousands of lines of unproven and sophisticated code that would otherwise be required using a message-centric approach. This wholesale reduction in the scope of the software development effort translates directly into tangible benefits that any project team would appreciate, including:

- Tremendous savings in development and maintenance budgets
- Significant reduction in the time required to bring projects to market
- Greatly enhanced reliability of deployed systems

Reliability: Error Avoidance, not Error Handling

Development teams must place a premium on delivering reliable systems, given the extreme consequences of failure in deployment. But unfortunately, the more code they have to write, the greater the chance that some of that code will contain defects. Data-centricity therefore delivers an important measure of reliability based on the reduction and simplification of the software development effort, as just described. However, data-centric middleware includes additional features which further contribute to reliability. As discussed earlier, the first step in designing a data-centric system is to define the data model, that is to say, to define the information necessary to understand and maintain the state of the system, the structure of data contents, and any constraints to be applied to their use. This up-front rigor in the design phase can identify and avoid many defects that would only be caught in a message-centric system after system integration. This is a much more precarious point at which to attempt to manage error conditions, and failure to do so adequately may result in incorrect message exchanges.

Once the data model is in place, the middleware enforces its properties in the form of explicitly defined "contracts" between information producers and consumers.

- **Contracts govern data contents.** For example, if our Asset Tracker application publishes tracks using X and Y mapping coordinates and a consuming application mistakenly attempts to subscribe to those tracks using a longitude/latitude data structure, the middleware will disallow the communication; hence, the error is never propagated to the application level—in fact, this class of error can often be prevented at the time an application is built, long before it is ever deployed.
- **Contracts also govern delivery qualities of service.** For example, if a subscribing application expects to receive updated tracking information at least every minute, it can expect to communicate successfully with a publishing application that promises to provide that information twice per minute, but it may not behave correctly when matched with a publisher that only updates every five minutes. Data-centric middleware allows applications to express constraints like this and can enforce them at run time.

It is a well established principal of software engineering that preventing programming errors is far more effective for delivering reliable systems than managing errors as they occur at run time. This principal is fully supported by the data-centric model, whereas it is completely unaddressed by message-centricity.

RTI Data Distribution Service

Hopefully, we have made clear the merits of data-centricity such that we can now consider what practical means are available for using this approach to develop real mission-critical systems. Fortunately, the Object Management Group (OMG) oversees Data Distribution Service for Real-Time Systems (DDS), an international standard which defines a formal specification for data-centric middleware. RTI Data Distribution Service is the leading implementation of the DDS standard and has been used in a wide variety of projects across applications as diverse as combat management systems, medical imaging, unmanned vehicles, air traffic control, and financial trading (to name just a few) to fully deliver on the promise of data-centricity.

Because RTI Data Distribution Service supports the DDS standard, it delivers on the complete cost, time, and reliability benefits of data-centricity. Moreover, it was designed to support the most demanding, high-performance, mission-critical systems with an eye toward real-world business considerations, and this led to two additional important benefits that are derived from the RTI implementation.

First, RTI Data Distribution Service delivers extremely high performance (see figure 2). The performance characteristics are derived from an efficient, C-based, fully peer-to-peer architecture. Message brokers, which are required by nearly all message-centric middleware implementations, introduce traffic choke points, single points of failure, and security vulnerabilities, issues which affect not only performance but reliability as well.

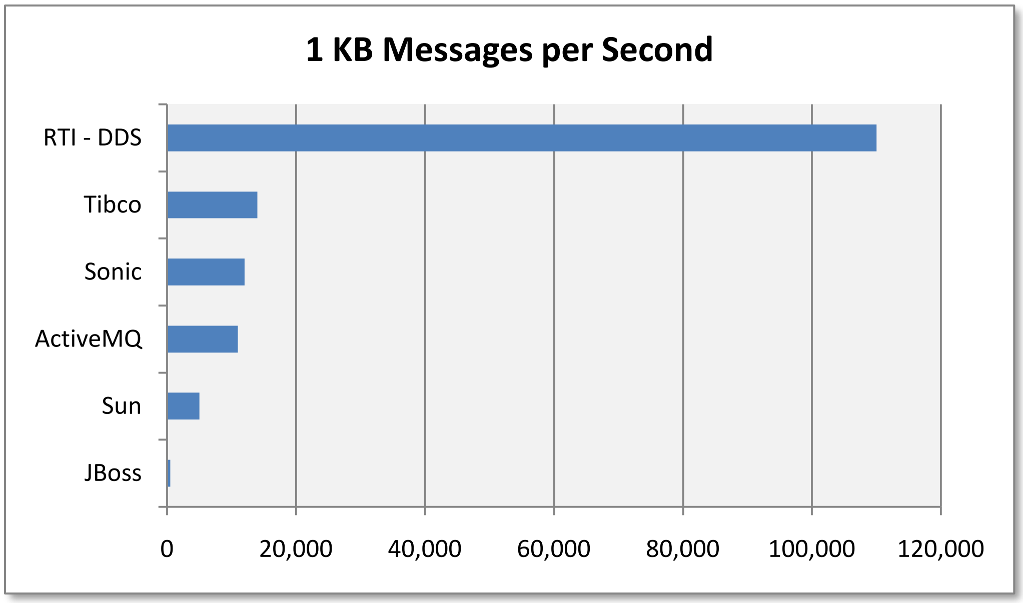

Figure 2 - JMS performance of RTI Data Distribution Service vs. other message-centric middleware

The efficient implementation yields the second important benefit: lower cost and complexity of deployed systems. Because no expensive servers are required, these expenses can be struck entirely from the deployment cost; as the system scales, this benefit magnifies in scope (see figure 3). In addition, data-centric techniques like data model design, producer-side filtering, and data serialization make for very efficient utilization of networking bandwidth, further contributing to cost reductions in deployment.

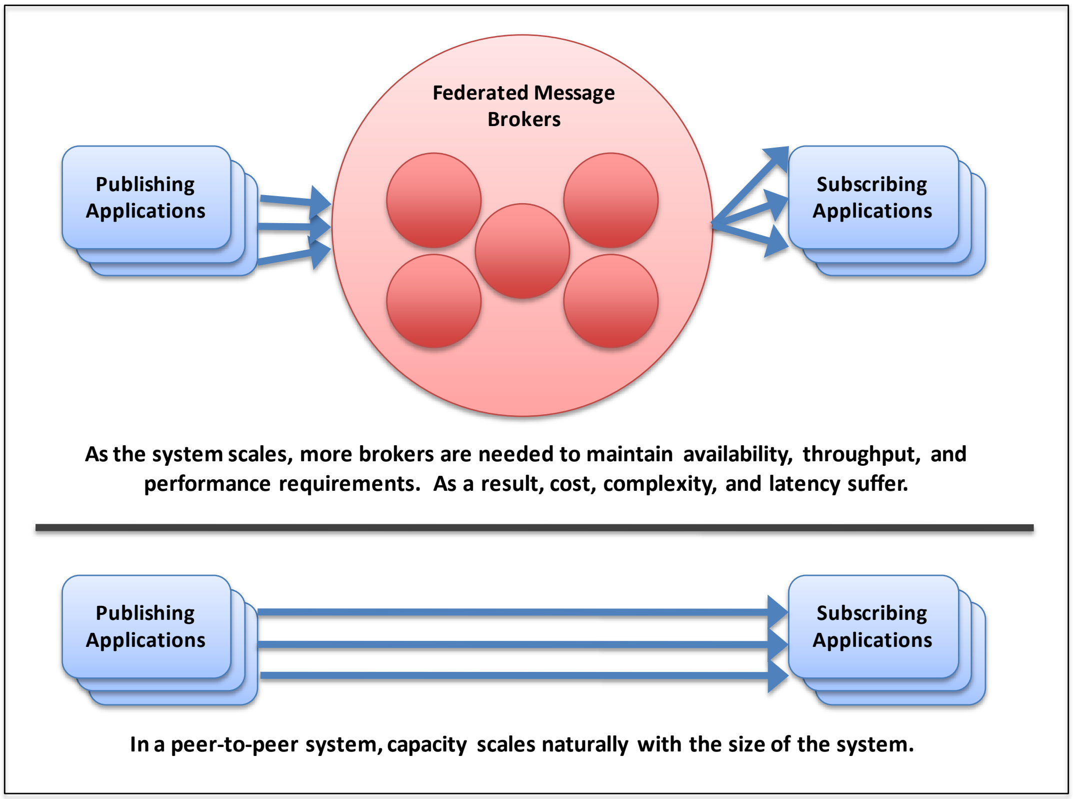
Federated Message Brokers
Publishing Applications
Subscribing Applications
As the system scales, more brokers are needed to maintain availability, throughput, and performance requirements. As a result, cost, complexity, and latency suffer.
Publishing Applications
Subscribing Applications
In a peer-to-peer system, capacity scales naturally with the size of the system.

Figure 3 - Peer-to-peer architectures are far more scalable than broker-based systems

Conclusion

In the modern world, two powerful forces are at absolute odds: system complexity is increasing while budgets are tightening. It seems clear that, in order to manage this state of affairs, we must look for new and different ways to do things rather than just making incremental improvements on the old ways. It should by now be clear that data-centric middleware provides that opportunity by enabling a fundamental step forward in efficiency for designing, developing, and deploying next generation, distributed mission-critical systems. As compared to the older message-centric approach, data-centricity delivers unmatched advantages, including:

- Lower lifecycle costs in deployment and maintenance
- Faster time to market
- Improved reliability in deployment

As these benefits are becoming better understood, commercial data-centric middleware products like RTI Data Distribution Service, are finding greater success. RTI Data Distribution Service goes beyond the core benefits of data-centricity by providing:

- Lower deployment costs
- Much higher performance
- Additional improvements in reliability

Having now been successively employed on many projects, a data-centric architecture must be seriously considered for any new mission-critical undertaking. To learn more, go to http://www.rti.com/.