Meeting Real-Time Requirements in Integrated Defense Systems

An RTI Whitepaper

Real-Time Innovations, Inc.
385 Moffett Park Drive
Sunnyvale, CA 94089

www.rti.com

Modern military operations integrate many functions including command and control, weapons, and self-defense. Developing and integrating these applications is particularly challenging because of their strict real-time performance demands. Often, latency and throughput requirements exceed the capabilities of traditional enterprise messaging and integration middleware.

Fortunately, extremely high-performance middleware has evolved that can handle millions of messages per second with tenths of milliseconds of latency---many times faster than messaging systems developed for less demanding enterprise applications. The technology is fast, flexible, scalable, deterministic, and reliable. Equally important, the middleware supports a net-centric design paradigm that greatly eases system integration and evolution.

This paper examines the challenges faced by defense network software, traces the evolution of real-time requirements, and examines how the required performance is possible with modern, standards-based, commercial technology.

The Driver: Distributed Defense Systems

In the recent past, most defense systems comprised independent, relatively isolated modules. A radar system, for instance, may have used a shared-memory multiprocessor for high-speed communications. However, this design implies that all systems that

depend on the radar's tracking information must also physically connect to the radar itself. These monolithic designs are expensive to design and maintain.

Thus, modern defense systems must be distributed. On the surface, this would seem a solved problem; many technologies have arisen to build large distributed systems in the enterprise. However, defense systems have some truly unique needs not met by enterprise solutions. For instance, these systems require extreme performance and reliability. Both the sheer volume of communications possible across the system (throughput) and the time between sensing and responding (latency) are critical. Real-time middleware must also provide reliability, fault tolerance, and system integration in environments quite different from enterprise computing.

As a result, enterprise middleware is often inadequate to deal with the requirements of integrated defense systems. System architects need a new approach.

Examples

Real-world applications drove the evolution of real-time middleware. Figures 1 through 4 show some of the hundreds of applications that have shaped real-time middleware.

Ship control; the LPD-17.

Figure 1 shows a ship in the US Navy designated LPD-17, the first of the San Antonio class of Navy vessels. LPD is a large ship, approximately 2/3 the size of a Nimitz-class aircraft carrier.

Real-time messaging middleware underlies a system on the ship called the Ship-Wide Area Network (SWAN). The SWAN is responsible for:

- Shipboard control
- Machinery control
- Damage control
- Integrated condition assessment
- Steering control
- Advanced Degaussing (Magnetic signature)
- Mission control systems
- Navigation systems
- Communication systems
- Support systems for "visitors"

The SWAN is absolutely critical to the ship's operation and achieving its mission. It comprises hundreds of computers. Because this ship may find itself in the middle of a conflict, it must also be able to take a hit anywhere and continue operations. This required the ship builders to look for middleware that supports features like automatic discovery so there would be no configuration. It supports redundant data sources, sinks, data paths, and transports. In fact, you can physically cut the ship network in half and the two halves will work independently. If you reconnect them, they will automatically rediscover and heal the system.

LPD was an early application of real-time middleware; the original design started in the late 1990's. It drove the middleware to be highly reliable and scalable to hundreds of nodes.

Figure 1: LPD-17

RTI middleware runs the entire Ship-Wide Area Network on the LPD-17. This application drove reliability and scalability

Ship defense control

Figure 2 is the control room for a ship self-defense system. This system is a last-line defense against incoming missiles and aircraft. It directs thousand-round-per-second depleted-uranium guns to track and shoot down incoming targets traveling at hundreds of miles per hour. It coordinates high-speed radars, decision systems, and fast, automated guns. These systems are now being deployed at sea on aircraft carriers and other large ships.

A few years ago, this system would have to be deployed under the control of a single computer "box", with all the disadvantages of a monolithic design. There was simply no way to communicate the data required fast enough over a network. As networking hardware has stepped up to the required performance level, the software has followed suit. This system drove the middleware to meet extreme performance requirements. It required distributing data with sub-millisecond latencies to dozens of nodes.

Figure 2: SSDS Control Room

The SSDS is coordinates high-speed radars and ship defense weapons. This application required extremely low-latency performance.

Radar systems

Figure 3 shows a series of images from a simulation test bed built at the Naval Surface Warfare Center (NSWC) in Dahlgren, VA. Navy engineers developed it to test load balancing and coordination for advanced radar systems. Similar radar applications include the Aegis and new DDG-1000 ships, as well as airborne sensors such as the Airborne Warning and Control System (AWACS) modified 707 in Figure 4.

Advanced radar systems must handle tens of thousands of simultaneous "tracks"---continuously-updated positions of aircraft, ships, vehicles, and missiles in an area. Ten thousand radar tracks generate a lot of data, and it must be delivered on time. This system required the middleware to reliably support extremely high throughput.

Figure 3: HiPer-D Simulator

This system simulated hundreds of computers cooperating to track tens of thousands of targets. It drove the middleware to support load balancing and high throughput.

Figure 4: AWACS

The Airborn Warning and Control System radar coordinates many high-speed radar and communication systems.

Characterizing Real-Time Requirements

Of course, these are only a few of the applications. Real-time middleware today is driving communications subsystems, fusing disparate sensors into a single world view, connecting motion control, graphics, and controls in flight simulators, integrating sensors and command for unmanned vehicles, and much more. Each of these applications requires "real time" response. Unfortunately, that isn't always well defined. We turn now to the metrics that characterize timeliness.

What is real time?

Formally, a computer system is "real time" if a correct response must be both computationally accurate and timely with respect to its external environment. For example, a control system that must respond to a sensor (e.g. a temperature rise) by changing some actuator (e.g., closing a valve) is real time if the valve reliably closes before the boiler explodes. Reliability is the key trait, and speed is important for reliability. This formal definition isn't often useful. In practice, a wide range of systems are called "real time", from a sub-millisecond weapon controller to a "fast enough" online airline reservation system.

Over time, two general meanings have arisen for real time, one for enterprise software and one for software embedded in devices such as defense electronics. In the enterprise, a system is real time if it responds "now" as perceived by a human. Thus, a system that reports stock prices within a few seconds is real time with respect to a human trader. In embedded systems, real time means predictable and fast with respect to physical processes. Generally, a real-time embedded system must reliably respond in a millisecond or less to be considered real time.

So, "real time" in enterprise applications can be orders of magnitude slower than "real time" for embedded applications. Over time, this discrepancy drove the technology

developed for the two industries apart; embedded real-time middleware is much evolved past its enterprise counterparts in timing reliability, or determinism.

Performance Metrics

To succeed, each application needs to execute its tasks quickly, reliably, and economically. However, even on a single computer, performance is not well defined. On a network, complex issues such as delivery to many simultaneous nodes, bandwidth utilization, media reliability, congestion, and failure recovery come into play. Each of these is, by itself, a dimension of performance. However, four key metrics do a good job of characterizing basic performance: latency, jitter, throughput, and efficiency.

Latency is the time between when one application sends a message and other applications receive it. Latency determines how quickly a system can respond to an external event.

The best way to reduce latency is to reduce handling. Each "hop", or server, that handles a message adds significant latency. Within a processor, latency is a layer game; after an application sends data it must still pass through the middleware layer, the network stack, the operating system, and the device drivers. Each takes time, but can be optimized by reducing processing and copying data. Latency also grows as message size makes transport time significant.

Jitter is a measure of how variable the latency results are from one message to another. A system with low latency and controlled jitter will deliver messages quickly without suffering from "outliers," or very late messages. For real-time systems, jitter is critical. In fact, jitter is often more important than average latency; a system with high jitter is unreliable.

Throughput is the total number of messages or quantity of data sent per unit time. For small messages, the overhead of passing through all the layers of software dominates throughput. As messages get larger, throughput usually increases; each delivers more data for roughly the same overhead. Ideally, throughput is limited only by the wire speed, e.g., a gigabit of data per second on a Gigabit Ethernet.

Efficiency measures the processor load needed to support the middleware. Most designs should strive to limit their middleware overhead to 15% of the CPU. That leaves sufficient room for application software and future growth.

Real-Time Middleware Technology

Now we turn to some of the unique aspects of real-time middleware and contrast it with enterprise designs. Examples are all based on RTI's implementation, which has been used in nearly 500 unique real-time applications, including those cited above.

Design Principles

We start by reviewing some of the assumptions about the environment and resulting design principles.

There can be no single point of failure or loading

Most messaging middleware uses a daemon or a broker-based architecture, which can lead to partial failure and complex failure/repair modes during recovery. The RTI infrastructure is based on a decentralized protocol, with no single point of failure, thus minimizing these partial failure situations.

The transport and the network are unreliable

RTI protocols assume that the media drops packets, links go down, and hardware fails. Thus, reliability is built into the messaging protocol above the network stack. Application Programming Interfaces (APIs) allow applications to control the reliability and receive notifications of network errors.

Local cache improves performance and resiliency

The RTI infrastructure provides a local in-memory cache for all data. The cache provides a simple layer of data management; the middleware can store recent values and provide them quickly without retransmission. Some alternate approaches push the burden of managing a local cache onto the programmer; with RTI's solution, the local caches are automatically updated to deliver the highest performance under changing network and dataflow conditions.

Topology can change on the fly

Operational parameters must be changeable on "live" systems without disruption. RTI's messaging and caching infrastructure provides automatic self-discovery and configuration, so that components can be added and removed dynamically on a live system without disruption. RTI-based applications do not suffer from the issues arising due to partial or broken connections because the underlying protocol is not connection-oriented. Furthermore, there are no startup dependencies imposed by the RTI infrastructure — applications can start in any order.

Changing requirements demand a tunable infrastructure

A high performance middleware infrastructure should be tunable, so that it can be optimized to meet specific application requirements. Thus, RTI exposes detailed configuration and behavior parameters that control every aspect of the messaging and caching infrastructure. For ease of use, all the "knobs" have default values "out-of-the-box" to address standard application requirements. The power and flexibility of RTI's solution comes through these "Quality-of-Service" (QoS) controls. Alternative approaches do not expose this level of flexibility and control, making the hidden/implicit assumptions opaque to the developer and non-customizable.

Meeting the Needs

Real-time middleware underlies many very-diverse systems. However, they all share common needs: flexibility, performance, service control, fault tolerance, and system integration. We now examine how the technology meets these needs.

Flexibility

Real-time systems are complex. As is often the case, the best way to address complexity is with a simple concept; the publish-subscribe paradigm. Conceptually, with publish-subscribe, you simply ask for the information you need and send the information you have. The middleware matches senders to receivers, ensuring that each data "contract" is satisfied. Of course, it's not that simple in practice; many details must be specified. But the overall model is intuitive and usable.

Real-time publish-subscribe differs most markedly from traditional middleware in tuning options. For example, most middleware is built on top of the Transmission Control Protocol (TCP). TCP was designed in the 1970's; it provides reliable byte-stream connections between two computers. Especially because it ensures reliability, TCP is very useful…but it's also very restrictive. For instance, TCP only supports communications between two computers. Its driving state machine has many timeouts, none of them user-settable. It supports reliability, but hides away all the important details: how many times to retry dropped packets, how much memory to use, when to send retries, etc.

RTI middleware, by contrast, is built on top of the User Datagram Protocol (UDP), a much simpler technology. The middleware takes control of reliability, retries, memory, and timing. All parameters are set to defaults, but exposed to the user application. Thus, users can tune the real-time middleware to handle much more demanding requirements with higher performance.

Reliability is but one of many, many parameters that affect a real-time system. RTI middleware also allows tuning control over timing and timeouts, memory and resource usage, network transports, priorities, and more.

Performance

Real-time middleware must be blazingly fast; the first step to meeting this challenge is cutting overhead to the bone. For example, older "client-server" designs require a round-trip request/response cycle for each message. With publish-subscribe, there is no request traffic for each message. All the information required to form virtual "connections" is exchanged through a process called "discovery" that occurs during initialization and when new nodes join. At send time, the sending node already knows exactly where to send the information.

Most middleware today uses central servers to coordinate data flow. Servers slow dataflow; sending to an intermediate server at least doubles the latency of sending a "nonstop" packet, since the packet must be both received and sent a second time. In practice, servers cause even more latency, since they may be loaded, congested, or not immediately responsive for many reasons. Interposing a server into every transmission also doubles the total traffic on the network. RTI middleware requires no servers, brokers, or daemons in the data path. Data flows directly, end-to-end, from sender to receiver.

On current hardware and operating systems, the stack and raw network transport can handle around 50,000 messages per second. This is largely independent of the message size until the wire approaches saturation. Broker-based middleware, such as most "MQ" products and JMS implementations, must pass each message through a network stack at least four times (sender, broker in, broker out, receiver). This reduces performance considerably, see Figure 5. Batching, the process of consolidating multiple application-level messages into a single transport-level datagram, minimizes the number of discrete messages passed through the protocol stack and sent over the wire. After batching, server-based architectures achieve throughput in the range of 100,000 messages per second with latencies of several milliseconds.

Because it skips all intermediaries, RTI middleware is capable of nearly this throughput performance without batching. With batching, RTI can achieve point-to-point throughput of over 3,000,000 application-level messages per second[1].

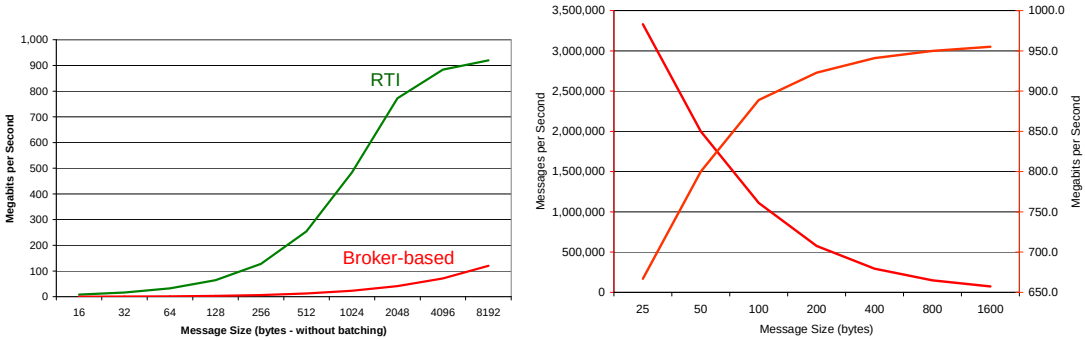

Figure 5: Throughput

RTI's direct peer-to-peer architecture achieves point-to-point throughput over twenty times better than broker-based designs. With batching (on right), it can send millions of messages per second.

Latency is also crucial. RTI has measured single-message latency (no batching) below 65 microseconds (0.065 millisec) with no appreciable jitter variation (Figure 6). Latency performance requires strict attention to real-time design principles. For instance, RTI allocates no memory after initialization, because finding memory blocks can add unpredictable delays. Also, RTI supports direct transmission in the user thread context. In this mode, there are no task switches or other operating system delays between the application send call and the activation of the network stack.

[1] 8-byte messages. Measured over Gigabit Ethernet between single-threaded applications running on 2.0 GHz Opteron processors with 32-bit Red Hat Enterprise Linux 4.0.

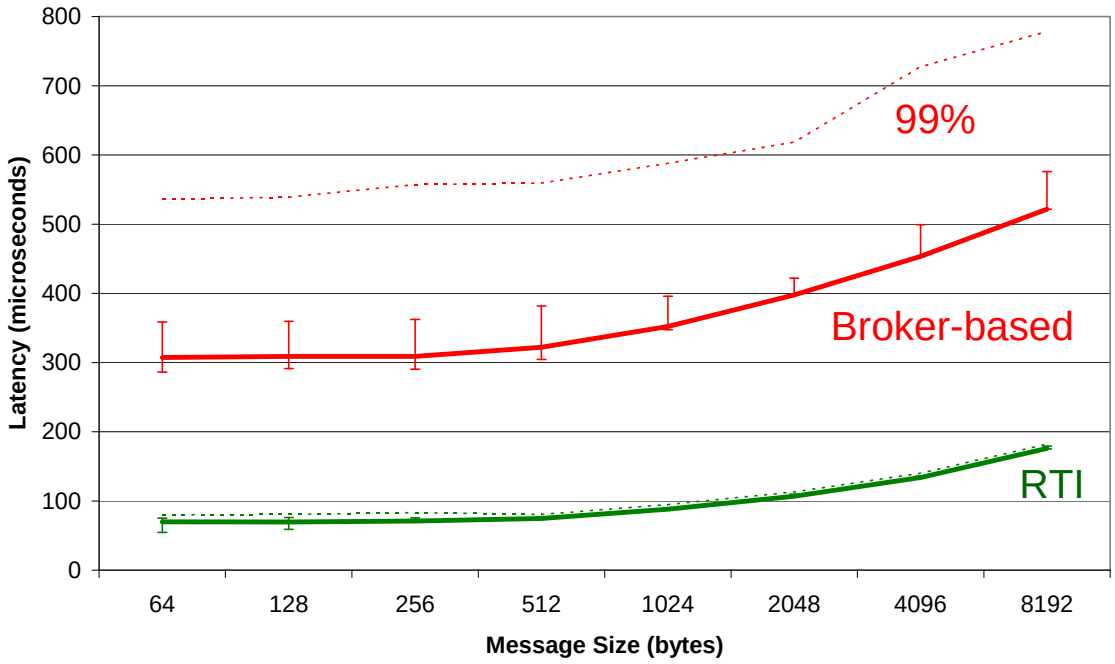

Figure 6: Latency

This graph compares RTI's latency with broker-based middleware. The solid line is the median latency with standard deviation error bars. The dotted lines show the "99%" latency; 99% of all messages fall within this bound. RTI delivers much faster latency with no significant jitter.

The real challenge is to combine high latency with low throughput. If the offered load (feed rate) is high, then many messages can be quickly combined into a single packet. RTI's batching implementation can run as a separate thread, allowing multicore processors to aggregate messages without incurring transmission delay. Thus, even an aggregated stream can achieve very low latency. With bursty offered-load rates, there is a tradeoff between low latency and high throughput. RTI lets the application make this tradeoff through both byte and time limits; it sends the message when either is surpassed. The time limit controls worst-case latency; the message is sent within a fixed latency of the first data queuing event. The byte limit keeps the message sizes within optimal ranges.

Reliability

Networks drop packets. This happens for many reasons, including medium intermittency, congestion, and stack buffer limitations. The only way to ensure delivery of lost packets is to detect the loss and retransmit. However, that takes time. Thus, there is a fundamental tradeoff between delivery reliability and time determinism.

Most middlewares offer little-to-no control over this tradeoff. RTI offers very fine control, all the way down to a per-data-stream level; you can choose strictly reliable or "best efforts" transmission differently for each data stream sent to each different node. RTI middleware additionally offers levels of reliability between these extremes by allowing control over the memory buffers used for reliability. On the publishing side, the memory is used to save a designated number of old messages for possible retransmission or updating a newly-joined application. On the subscriber side, the memory can buffer packets received after a lost packet, letting the system efficiently request retransmission of only the few packets that were actually dropped.

Multicast

Multicasting, the ability to send a single packet to many destinations, drastically cuts overall latency and raises effective throughput while increasing efficiency. Multicast can theoretically send information to 50 nodes 50 times faster than unicast.

Multicast reliability is a famously-hard challenge. Simple methods, such as acknowledging every packet, flood the network with acknowledgment traffic, defeating the prime benefit (Figure 7).

Multicast reliability requires sophisticated negative-acknowledgment (NACK) strategies. Instead of sending positive acknowledgments, NACK-based protocols request retransmission only for the few messages actually dropped. Configurability is important; the reliability protocol must ensure rapid detection of missed packets to fulfill latency requirements while minimizing network overhead.

The RTI protocol tags messages with sequence numbers. A subscriber detects that a message is lost when it receives a message out of order. Publishers periodically send "heartbeat" messages with the last sequence number to bound lost-message detection time. Heartbeat messages are normally piggybacked on application messages to reduce network overhead. To avoid flooding the network if many nodes miss a packet (a NACK "storm"), each subscriber applies a random delay before replying to a heartbeat with an acknowledgement. This prevents multiple subscribers from responding at the same time.

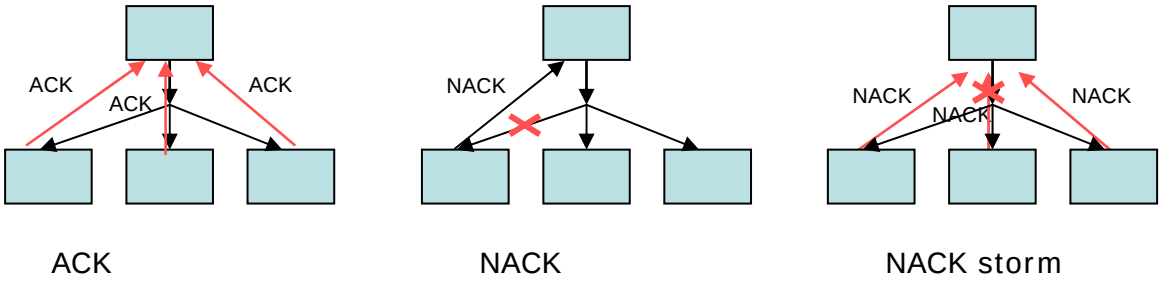

Figure 7: Reliable Multicast

Multicast is efficient, but reliability is a famously-hard problem. Acknowledging every message just shifts the congestion to the confirmation stage. Negative-acknowledgement works well for occasional drops, but results in "storms" of retry requests if many nodes miss a message.

Bandwidth Control

When many processors share a network, there is always the danger that one will put so much data on the network that it locks out the others. Thus, bandwidth control is critical; optimal overall operation requires limiting the bandwidth that a single publisher can use. To control bandwidth the middleware sets the maximum data sent in a specific time window. The system automatically buffers any overflow, and feeds it gradually into the network later.

Filters

One key way to increase performance is to be smart and deliver only the data really needed. RTI offers several ways to accomplish this, including partitioning, time filters, and content filters.

Partitioning logically divides the network into several sub-networks. Messages may be published into one or more logical partitions; only subscribers in one of the publisher's partitions will receive the data. This can greatly reduce the dataflow traffic. Equally important, it simplifies discovery by restricting the potential extent of the data distribution.

RTI also offers time-based filters. A time-based filter sets a minimal separation period between updates. This can be used in a radar application, for instance, to update an operator's console with new information at most once every 5 seconds. Without this, the application would have to sort through potentially thousands of updates to keep the display current.

Content-based filters only send updates that match pre-set content criteria. Thus, for example, a HMI program could be configured to only accept radar tracks within a

specified range, or a monitor program could request an alarm if traffic exceeded 80% load.

The middleware attempts to make intelligent decisions about publisher versus subscriber-side filtering. Since it reduces network traffic, publisher-side filtering is preferable when the middleware can easily determine that there are no subscribers requesting the data. However, subscriber-side filtering is much more efficient, for instance, when a multicast packet is destined for 80 of 100 subscribing nodes.

Quality of Service Control

Many designers overlook the effect of *Quality of Service* (QoS) on system performance and capability. By QoS, we mean the semantics of delivery of data, including factors like

- which data to deliver,
- when delivery is expected,
- what to do in case of failure, and
- how many resources to use in the attempt to deliver data.

QoS settings optimize network resources to fit the problem. RTI middleware can, for instance, send radar track updates every 10 seconds to an HMI operator station, reliably record every reading on a database server, send another application updates only for those targets within a 5-mile radius, while simultaneously multicasting thousands of updates per second to hundreds of cooperating applications on the network.

RTI offers a unique per-data-stream control over QoS. It does this through request/offered semantics: publishers offer levels of service and subscribers make service requests. The middleware establishes communication only if the request can be satisfied by the offer. The middleware then enforces this semantic contract; any violations are reported to the application.

For instance, a publisher can offer "best efforts" updates, meaning that it will not save any old data for attempted retransmission. If a subscriber requests reliable service, the middleware will deny this logical connection. On the other hand, a reliable publisher is capable of serving a best-efforts subscriber, so that connection will work.

Fault Tolerance

Complex systems need the ability to detect and manage faults. Of course, this is even more important---and difficult---in a distributed system. Fault tolerance provides the ability to handle unexpected or degraded conditions and keep going. This is no small issue; handling error conditions accounts for up to 80% of typical networking code.

The first requirement is to know when part of the system has failed. All middleware designs provide this in some form, either through exceptions on transmit (most enterprise implementations) or explicit "are you alive" protocols (RTI). The designs differ mostly a) in how and when they notify the application, and b) in the granularity of notification,

for instance whether you are notified of failure of just your communication or of all the responsibilities of a node when it goes down. RTI provides much finer notification control and faster notification than other designs, because it was intended from the beginning to be a real-time protocol.

Once detected, the fault must be handled. The first priority is to continue operations without impacting the rest of the system. This is sometimes non-trivial; for instance, older TCP-based designs often suffer huge delays while connections time out; they lack the QoS control to adapt quickly. Because the RTI publish-subscribe architecture is not connection-oriented, it can offer an intuitive first-level failover capability. The middleware allows multiple publishers of the same data stream and arbitrates which data stream is received by subscribers based on a specified priority.

System Integration

Today's network technology makes it easy to connect nodes, but it's not so easy to find and access the information resident in networks of connected nodes. This is changing; publish-subscribe middleware allows applications to pool information from many distributed sources and access it from many locations at rates meaningful to physical processes. Many label this new capability the "net-centric" or "data-centric" architecture.

System requirements are also becoming much more dynamic. In the past, software may have assumed a fixed application design and a fixed number of nodes. Increasingly, we see that distributed systems must adapt to changes in scale, function, and performance requirements before, during, and after deployment. System designers need to "plug and play" network nodes and topologies as they match the evolving needs of the distributed application. They must architect distributed systems that meet the needs of the target application today but can also embrace the changes of tomorrow. Delivering such a high level of flexibility in these designs creates a major challenge for the development infrastructure.

A data-centric architecture fundamentally changes how easy it is to design and evolve a networked application. Figure 8 shows a real-world example taken from the design of the US Navy's E-2C Hawkeye aircraft. Data-centric thinking transformed this design. The data-centric architecture is more modular and maintainable than older client-server based designs. It provided a structured overall design paradigm that allows expansion, changes, and independent development.

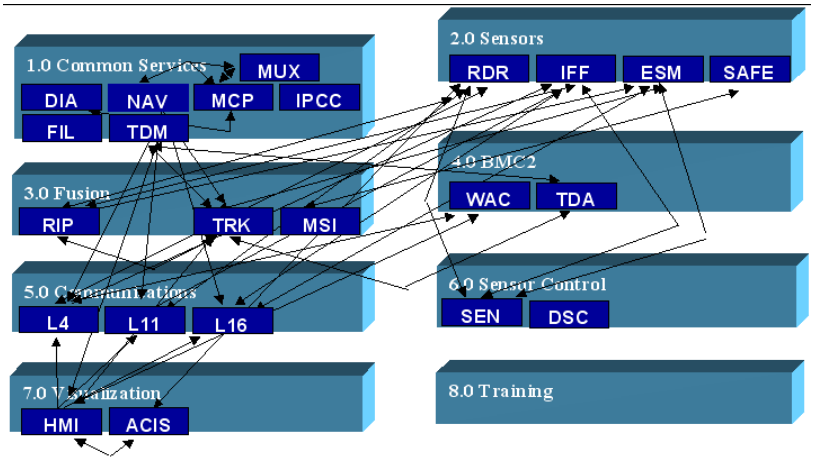

Figure 8a: Functional Design

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

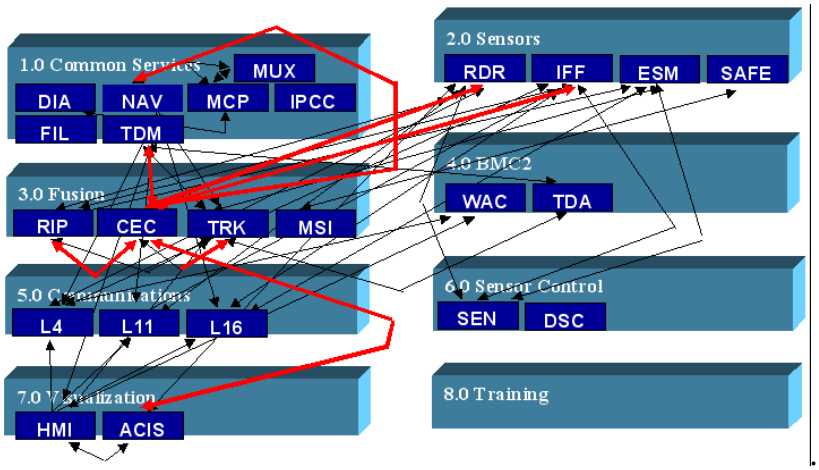

Figure 8b: Integration

Adding new functionality cascades integration re-work across many modules.

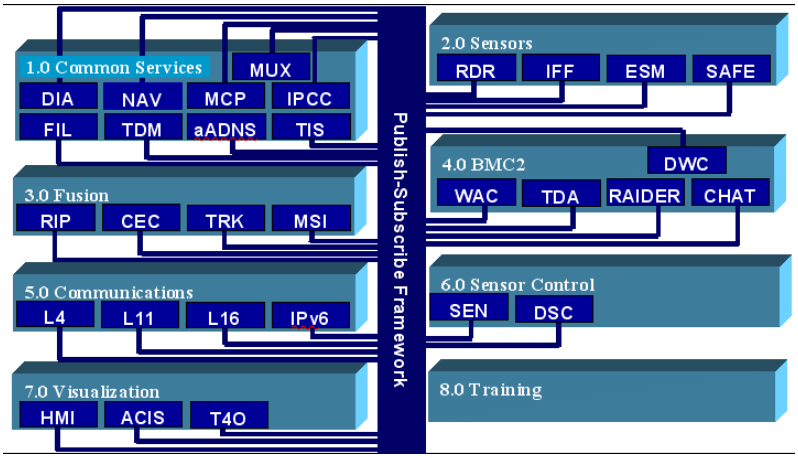

Figure 8c: Publish-Subscribe Design

Publish-subscribe architecture simplifies data communications, greatly easing integration.

The DDS Standard

Because of their long lifecycles and multi-vendor contributions, defense systems require standards. In 2005, the Object Management Group (OMG) adopted a standard called the Data Distribution Service for Real-Time Systems (DDS). This standard is the first middleware specification that directly targets high-performance distributed systems. The standard includes both an API specification and a wire-protocol design. RTI was one of the key drivers of the DDS standard.

The DDS standard has become the rallying point for high-performance, standards-based middleware. For example, nearly every US Navy surface ship under design or refresh has standardized on DDS for networking middleware. The penetration goes beyond ship-wide networks to include most high-performance weapons, radar, and communications systems on the ships. The Army's huge Future Combat Systems (FCS) program has also chosen DDS as a standard rallying point. Aircraft are also rapidly adopting the technology, starting with on-board radar systems and moving to communications and UAV ground-support systems. Recently, the joint Air-Force/Navy communications initiative, the Net-centric Enterprise Solutions for Interoperability (NESI) mandated use of DDS for the 160 programs it oversees. Finally, The Defense Information Services Agency (DISA) has mandated DDS for all data-distribution applications throughout the US Military.

International adoption is also on the rise. Most major NATO weapons system designs are upgrading to DDS technology. Also, DDS forms the core communications capability for South Korea's most important new ship systems.

RTI is the DDS market leader with over 80% market share according to two independent market analyses.

The Future: Benefits of Real Time

Today's defense systems rely on distributed designs. To take advantage of the benefits of distributed computing, these systems must depend on communication middleware layers. Fortunately, real-time middleware is proven, capable, and easily applied; it is changing system architectures and leading designers to faster, higher-performance, more maintainable distributed systems. These advantages are critical to lowering the military's spiraling IT costs.

In defense systems, including C4ISR, radar, and simulation applications, fractions of a second make the difference between success and failure. The performance achieved by current technology is impressive. With current hardware, real time publish-subscribe can achieve latency less than 65 microseconds without significant jitter. Throughput for larger messages is limited only by the wire speed. With batching, sustained throughput for smaller messages can exceed 3 million messages per second.

Defense systems also need flexible designs that will adapt to changing requirements over time. The rise of the DDS standard helps significantly. With DDS, defense architects have a much more flexible, high-performance integration infrastructure. Fine Quality of Service control is especially important for ensuring high performance, for controlling reliability, and for developing fault-tolerant systems.

Real-time middleware is mature and proven; it is broadly-deployed technology used in hundreds of actual applications working under harsh real-world conditions. It delivers the performance and functionality to address the new realities facing the defense industry.