Applying Publish-Subscribe to Communications-on-the-Move Node Control

J. Darby Mitchell, Marc L. Siegel, M. Curran N. Schiefelbein, and Armen P. Babikyan

Modern military satellite communications terminals have typically been built as multiprocessor systems. Because of increasing pressure for reuse and modularity, current programs have been encouraged to consider the use of component middleware. While Common Object Request Broker Architecture is the most mature middleware standard available, its invocation semantics present considerable challenges for the development of such systems. Through reasoning about quality attributes, we found that a real-time publish-subscribe middleware reduces coupling, improves composability, and reduces the risk of architectural mismatch, deadlock, and integration problems compared to an invocation-based system. In building a communications-on-the-move (COTM) node, we found that this type of middleware, which exemplifies an implicit-invocation architectural style, promotes ease of system evolution and an incremental integration approach.

Because of the computational demands of modern military communications terminals, systems tend to be implemented as distributed real-time embedded (DRE) systems. We divide functionality among several processes on different processors for two reasons: to enable the system to meet the real-time requirements imposed on it, and to inject external inputs into the system, whether from a user interface or some sort of sensor. Because these processes must cooperate to realize the functionality of the system, the consequence of this design decision is that they must exchange data and control messages. Therefore, one of the first design decisions an architect must make, after deciding to distribute functionality, is how to facilitate this exchange of data and control messages.

DRE systems architects are increasingly looking to middleware* to provide this capability. Middleware creates an abstraction layer that decouples an application from the system calls and network interfaces required to send and receive data on a particular platform. Middleware typically provides location transparency, synchronization, and bit representation conversion, as well as well-defined semantics for exchanging data. While middleware may not be applicable to all DRE systems, it is certainly worthy of consideration.

The use of middleware is not an all-or-nothing proposition. High-rate data traffic that may not be able to absorb the overhead of any abstraction can still be passed by lower-level network interfaces. In modern military communications systems, it is commonly accepted that middleware can be used for control, if not for data [1]. This article discusses the justification and implementation of a software control framework for a prototype communications system using publish-subscribe middleware.

* Middleware is software that mediates between an application program and a network. It manages the interaction between disparate applications across heterogeneous computing platforms [2].

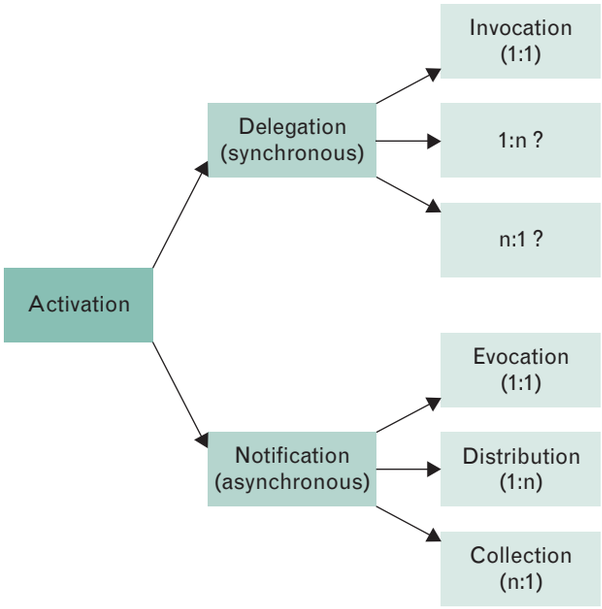


FIGURE 1. Partial taxonomy of connector semantics, showing the relationship between the different types of connector semantics for distributed interaction.

Table 1. Taxonomy Terms

Activation	A general term for interaction with a remote component.
Delegation	A general term for synchronous activation, whereby the originator waits for and receives a response that implies completion of the operation.
Invocation	A strictly one-to-one delegation.
Notification	A general term for asynchronous activation, whereby the originator immediately continues its execution after the message is sent.
Evocation	A strictly one-to-one notification.
Distribution	A general term for one-to-many notification.
Collection	A general term for many-to-one notification.

Terminology

Before continuing, we must review a few terms that describe how components exchange data. A.D. Birrell and B.J. Nelson introduced Remote Procedure Call (RPC) in 1984 [3]. One of their guiding principles was for the semantics of RPC to be as similar to the semantics of local procedure calls as possible. Object-oriented programming introduced the term *invocation* to refer to calling a procedure exported by an object's interface (hereafter referred to as a *method*). With the rise of middleware in the 1990s, software developers began referring to invoking a method of a remote object as remote invocation. However, one of the stated goals of middleware is distribution transparency, which implies that local and remote invocations are semantically and syntactically equivalent. As a result the term invocation is often used to refer to calling a method, whether or not the object is local or remote.

It is also common to see the shorthand *A invokes B* to refer in general to activation of the object B's interface by object A. While middleware may obscure the distinction between local and remote invocation from a programmer's perspective, from a software architecture perspective these two uses of the term are distinct, with very different implications. R. Guerraoui and M. Fayed provide a good discussion of architectural concerns with distribution transparency [4].

This article focuses on software architecture for DRE systems, and uses the partial taxonomy illustrated in Figure 1 and Table 1 to provide a vocabulary to discuss the semantics of connectors. D. Garlan and M. Shaw state that "connectors mediate interactions among components, that is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required [5]." We say partial taxonomy because (a) only two dimensions are represented (synchronization and connection type), and (b) the graph is obviously incomplete with respect to synchronous connectors.

While this taxonomy is by no means complete, it has the advantage of being convenient. It allows us to discuss the design of a distributed system in terms of the connectors rather than the components. Assume that middleware, which provides a type of connector, handles all the details of implementing that connector: message passing, synchronization, and marshalling* for any data exchanged with remote processes, and perhaps some sensible alternative to message passing for local processes such as shared memory.

* Marshalling is a term used in middleware to refer collectively to serialization and representation conversion.

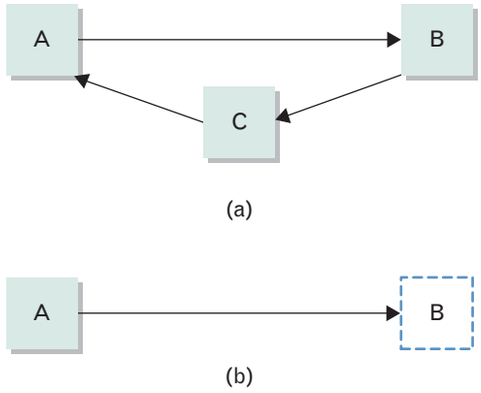


FIGURE 2. Potential deadlock in invocation semantics. (a) Cyclical dependency; (b) invocation of an unavailable object.

Previous Work with Invocation

In our previous efforts, our team developed a custom middleware layer for control messages. With limited time and resources, we developed a simple framework similar in concept to basic Common Object Request Broker Architecture (CORBA) [6]. This framework provided both invocation and evocation semantics, and included features like serialization, representation conversion, and name resolution. The majority of the component interactions were designed as invocations consistent with object-oriented programming practices. However, as we designed this framework, we realized that there were conditions that could result in deadlock. Given our limited time and resources, this led us to strict design constraints for how the top-level components could interact.

Deadlock Potential

In 1999, in its Evolutionary Design of Complex Software (EDCS) announcement, the United States Defense Department stated that

> "a major theme of this year's demonstrations is the ability to build software systems by composing components, and do it reliably and predictably. We want to use the right components to do the job. We want to put them together so the system doesn't deadlock." [7]

Because invocation semantics are synchronous, and an invoked method may, in turn, invoke other methods on other objects, connections may form a cyclic dependency graph, as shown in Figure 2(a). Unless a separate thread handles incoming invocations, this dependency results in deadlock. Also, if an object tries to invoke a method on an object that does not exist, as shown in Figure 2(b), deadlock can occur.

Generally, middleware frameworks implement several tactics to avoid deadlock. Even if the middleware offers advanced options for server-threading models such as support for concurrent requests [6], deadlock may still be a risk if there is contention for shared resources among methods. Time-outs are generally used to mitigate the availability problem shown above. However, potential for deadlock continues to be a concern in the design of distributed object systems [8]. While progress is being made toward proving deadlock freedom for distributed-object systems [9, 10], this design process ignores the broader issue: invocation semantics limit the ways in which objects can be composed into a system. This reliance on invocation semantics limits the reuse potential of a component.

Some may argue that deadlock due to resource contention occurs in multi-threaded non-distributed systems, and therefore is an essential, rather than an accidental, complexity of the design process [11]. We assert that deadlock need not be an *integration* concern at all.

Architectural Mismatch

Two commonly used patterns for exchanging data via invocation are illustrated in Figure 3. A client may request data from a server, which returns it. Alternately, a forwarder may forward data to a receiver. The difference is that the data are either *pulled* or *pushed*.

The architectural mismatch illustrated in Figure 4 suggests further limitations of invocation with respect to reusability [12]. We cannot connect a PositionClient and a PositionForwarder, even though one requires position and one provides it, because they each expect to initiate the invocation. Obviously we wouldn't design a

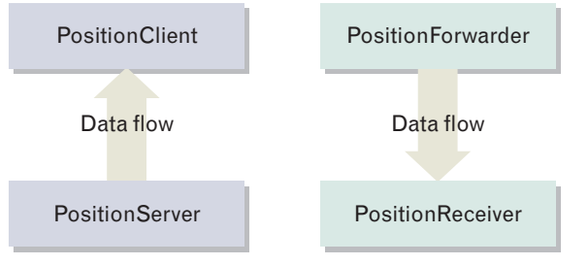


FIGURE 3. Data flow in common design patterns for invocation. Comparing this with Figure 4 illustrates how data flow and direction of invocation are independent of each other.

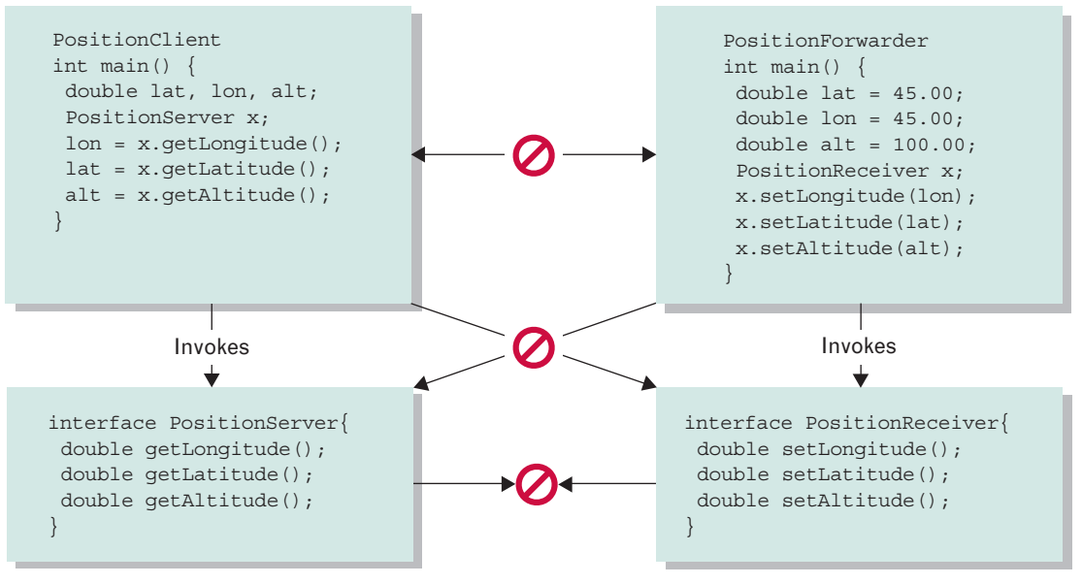


FIGURE 4. Invocation direction in common design patterns. Components that are built to use invocation potentially exhibit architectural mismatch.

system with a PositionClient and a PositionForwarder, but the fact that the direction of data flow is independent of the direction of invocation is a risk to the reusability of these components.

Architectural mismatch will probably always be a challenge. However, we assert that it should not be so between two components that use the same type of connector and agree on data format.

Timeliness

Timeliness concerns can propagate through invocation semantics. Consider the illustration in Figure 5. The timeliness of component A is dependent on the time required for component C to process method `foo`, which is dependent upon the time to invoke methods on components D and E. Assume that B invokes method `bar` immediately after A invokes method `foo`. In a single-threaded server model, B's invocation will not occur until after `foo` (and the invocations to D and E) has completed. Even in a very clever threading model, the invocation process can still pose a problem if `foo` and `bar` both require exclusive access to the same shared resource.

If component A or B has very strict timeliness constraints, those constraints are inherited by components C, D, and E. If A's invocation must complete in time t, then `foo` can take no longer then $t - l$, where l is the round-trip latency of the request-response from A to C. The time l then imposes timeliness constraints on D and E, which include the round-trip latencies for their invocations as well.

Another problem related to timeliness that can result from the use of invocation on systems with priority-based scheduling is priority inversion. We were not using priority-based scheduling in our previous work, but we mention it here because of its implications to timeliness. Since method invocations are synchronous, it is possible for a higher-priority client to synchronize with a low-priority server process, which is then preempted by a medium-priority process. Such issues are generally mitigated with an end-to-end priority policy and priority inheritance, although this policy does not completely eliminate the problem.

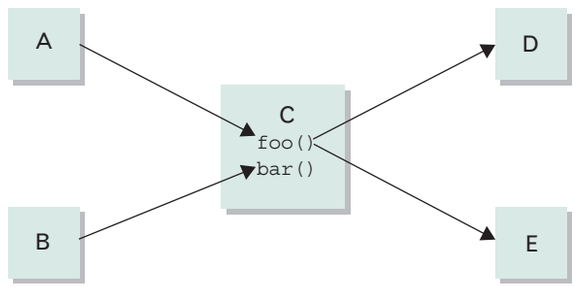


FIGURE 5. Timeliness in invocation. Invocation of one component can lead to invocations of other components and impact timeliness.

Considering Notification

We consider the partial taxonomy of architectural styles found in L. Bass, P. Clements, and R. Kazman's book [13], and find that systems of cooperating components can be built using the implicit-invocation style, a substyle of event systems, as shown in Figure 6. Event systems are also referred to as reactive or selective broadcast systems. M. Shaw and D. Garlan discuss the implementation of implicit invocation:

> "…Instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with it. When the event is announced, the system itself invokes all the procedures which have been registered for the event." [14]

While the style refers to invocation, this is actually a reference to the callback procedure invoked. This is not a constraint on the connector used between processes, which could be notification.

According to Garlan and Shaw, implicit invocation is very good at promoting reuse and extensibility, which are essential requirements for our system. They mention that it "eases system evolution," which will be discussed in a later section [15].

We look to architectural design patterns for examples of an implicit-invocation pattern for distributed systems communication. One communication design pattern, Publisher-Subscriber [16], uses notification semantics. We prefer the term Publish-Subscribe to Publisher-Subscriber because the former emphasizes the connector rather than the components. This distinction seems appropriate when we consider an architectural design pattern for connecting components. Publish-Subscribe specializes the distribution and collection connectors with a registration scheme, and then composes them to create a Publish-Subscribe connector by adding a variation of the Mediator pattern [17] called a Topic, or event channel. A Topic is an intermediate abstraction represented by a name and type.

A Publisher registers its intent to publish a particular Topic. A Subscriber registers to receive updates on a particular Topic. These two events can occur in any order. The middleware maintains mappings of Publishers to Topics and Topics to Subscribers. When a Publisher has an update, the middleware publishes it to all current Subscribers of that Topic. In this way, Publish-Subscribe provides location transparency and many-to-many, *anonymous* notification between Publishers and Subscribers, as shown in Figure 7.

According to Bass et al., consideration of quality attributes is an integral part of the architecture design process [13]. It is important to recognize that there are trade-offs inherent in any design, and that there is no way to maximize all quality attributes. It is also important to realize that architectural design is a sequence of design trade-offs, and the most important decisions should be made first. In the design of the architecture for a distributed system, the way the components connect should be one of the first decisions the designer should make.

Given our resources and time constraints, we were not able to perform an extensive analysis of quality attributes, or develop a complete set of quality attribute scenarios for our system. On the basis of Bass et al.'s work

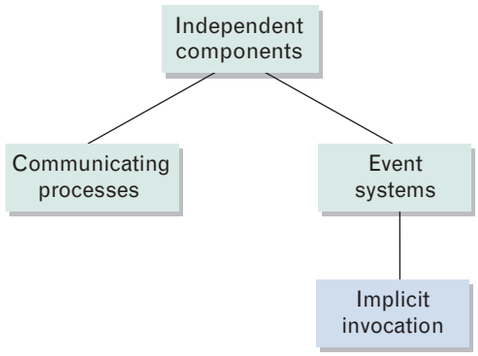


FIGURE 6. Partial taxonomy of architecture styles for cooperating components. Implicit invocation is a type of event-driven architectural style for cooperating components.

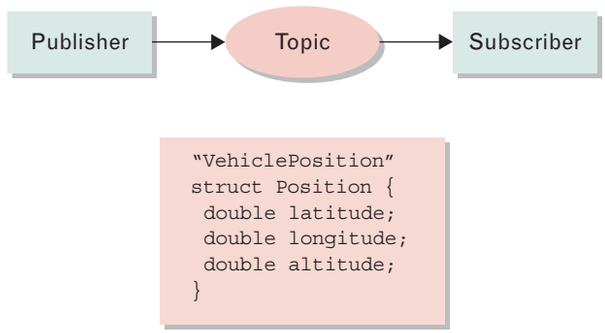


FIGURE 7. Publish-Subscribe. A publisher and a subscriber are decoupled through a topic, which is defined by a name and a type.

[13], we were able to reason about quality attributes and use them as a guide for architectural design. Now we examine the trade-offs among different qualities for invocation versus notification.

Simplicity and Composability

There are several dimensions to simplicity, but obviously one of them is algorithm simplicity. With invocation, programming a sequence of synchronous interactions between components is a trivial exercise. Implementing a sequence of synchronous interactions is more work with notification. One approach is by implementing the components as state machines. Clearly, this approach is not as straightforward as the invocation approach.

Algorithms are further complicated by using Publish-Subscribe, since there are no guarantees that there are any Subscribers when an update is published. Likewise, there are no guarantees to a Subscriber that the system contains a Publisher for a given Topic. So, Publishers and Subscribers can make no assumptions about which other components are present in the system, or when they are present.

The simplicity of invocation semantics comes at the cost of potential for deadlock, as mentioned previously. Notification, being asynchronous, does not suffer from this drawback. If the system uses notification exclusively, it has the advantage of removing deadlock from the integration problem. Removing deadlock as an integration concern removes constraints on how components may be composed to create a system.

Reliability and Predictability

When considering invocation and notification, we thus know that there is an obvious trade-off between reliability and predictability. Using invocation, the calling process waits for and receives a result, guaranteeing that the remote method has completed successfully. The trade-off for this reliability is the fact that the time required to complete an invocation depends on the object and method invoked, the current load of the processor it resides on, and the current network load.

With notification, the sender must assume that the message will be received and processed correctly. The time required to complete a notification is only the time required to create and send a message, which is independent of the object(s) being notified. Obviously, if distribution semantics were not optimized with some multicast or broadcast messaging scheme, notification would be slightly less predictable, since it would create and send a message to each receiver.

Timeliness

The timeliness of a notification is independent of the processing time of the receiver(s). It is solely dependent on the processing required to create and send the notification(s). As long as the component is not designed to *busy wait* for a response to a notification (which would be a serious design mistake, given the architectural style), its timeliness is completely independent of the other components in the system.

The design trade-off is one of progress. While a component's timeliness can be verified in component testing, its progress is solely dependent upon the inputs it receives, and therefore the other components it is integrated with. If a component expects to be provided with vehicle position, and uses that information to calculate velocity, then it obviously will not make progress in calculating velocity if it receives no notifications of position.

Reusability

We have identified two risks related to reuse for invocation: limitations on system structure due to deadlock, and potential for architectural mismatch. We have already observed that deadlock should not be an integration problem with notification. There is also less risk of architectural mismatch because the data source always initiates the interaction. With publish-subscribe, components cannot make assumptions about the existence of other components. Of course, components must agree on data type to exchange data in any case. Evocation could also have a mismatch if the interface-method signatures were not the same. However, publish-subscribe does not suffer from this limitation.

As we consider these qualities in the context of a DRE system, predictability should be given priority over reliability. We don't mean that reliable communications aren't necessary. However, the nature of architectural design is a series of trade-off decisions. Since we are building a system that must be predictable by definition, we choose to make a decision that promotes predictability and defer the reliability problem. It still must be resolved, because a system that loses control messages will not act predictably either. As we will see, however, there are other ways to achieve reliability that don't have the same trade-off with predictability.

Reusability and extensibility are essential as well, since the plan for this system calls for incrementally adding functionality over the course of at least three spirals spanning several years. These were the factors that led us to consider applying publish-subscribe to our mobile communications node prototype.

Middleware Comparison

We considered two potential middleware frameworks: CORBA and the Network Data Distribution Service (NDDS). CORBA, a standards-based middleware specification published by the Object Management Group (OMG), has many commercial and open-source implementations. NDDS is a commercial product of Real-Time Innovations (RTI).

CORBA

The design of CORBA has evolved from its origins in enterprise-distributed systems. The fact that its primary connector type is invocation is an artifact of those origins. With the publication of the Real-Time CORBA specifications for dynamic [18] and static [19] scheduling, it has been retrofitted for use in distributed real-time systems. However, the primary connector is still invocation.

CORBA has the advantage of flexibility: we can use invocation, evocation, or a special connector called *deferred synchronous*. This last option is like evocation with a return value that gets cached by the middleware. The trade-off of this flexibility is that the more connector types that middleware provides, the more opportunities there are for architectural mismatch. This flexibility thus represents an increased risk to potential reuse.

CORBA has the advantage of maturity and standardization. A number of open-source and commercial CORBA implementations can interoperate with one another. Several are fully compliant with the Real-Time CORBA specification and the minimum CORBA specification (designed to reduce memory and storage footprint) [20]. CORBA does have a fault-tolerance specification, but it is not clear how much vendor support it has. Furthermore, it is not at all apparent how to integrate Fault-Tolerant CORBA with Real-Time CORBA.

Due to its maturity, CORBA has the additional advantage of offering a host of supporting services such as Naming, Event, Notification, Lifecycle, Concurrency, Security, and Transaction. A CORBA vendor is not required to implement all these services, but many implementations supply most or all of the services. Some implementers of the Real-Time CORBA specification provide real-time implementations of the Event [21] and Notification [22] services.

CORBA has language mappings for C, C++, Java, Ada, COBOL, Smalltalk, Lisp, Python, and several other languages. CORBA's Event and Notification services do not specify any multicast optimizations.

NDDS

NDDS has been designed and built for distributed real-time control systems. The fact that its primary connector type uses notification semantics is an artifact of its origins. NDDS did provide a client-server option in the version we used, but it was being phased out. The latest version of NDDS offers only notification semantics. This version is not as flexible as the alternatives available in CORBA, but it does help mitigate the opportunities for architectural mismatch resulting from mismatched connectors. The version of NDDS we used was a proprietary commercial product. The vendor did publish their wire format, Real-Time Publish-Subscribe (RTPS) [23].

NDDS provides a variety of parameters to tune Quality of Service (QoS), and provides end-to-end QoS guarantees. One of the disadvantages of this flexibility is the potential for architectural mismatch. Two components may agree on a topic, but disagree on QoS expectations that may prevent them from communicating. Also, not all combinations of parameter settings are semantically valid, which can make tuning them a challenge.

NDDS was available for Java, C, and C++, which was sufficient for our needs. NDDS provides a multicast option to optimize its distribution semantics. NDDS provides a reliable notification mode, which adapts notification with an acknowledgement scheme that is handled completely by the middleware. Publishing is still asynchronous. However, the middleware sets a timer and expects an acknowledgement from the subscriber's middleware. If the timer elapses without acknowledgement, the middleware resends. This option trades off a small amount of predictability (i.e., the uncertainty of when an acknowledgement will be handled and the possibility that the middleware may have to resend a message) to ensure reliable delivery.

Reliable delivery is a weaker guarantee than the reliability of invocation, since a complete invocation guarantees that the receiver has successfully processed the

message. However, there seems to be an appropriate separation of concerns. The message is a concern of the sender until passed to the middleware. It is the concern of the middleware until delivered to the receiver, and it is then a concern of the receiver for correctly processing the message.

NDDS has a mechanism for active fault tolerance using the concept of publication strength. Two identical copies of a component can be run on different processors. Since they subscribe to the same publications, they receive the same inputs and generate the same outputs in the form of publications. However, one of the components can be set with a higher publication strength, meaning that its publications supersede those of its twin. If this *primary* component crashes, the backup (publishing at a lower publication strength) is still active.

Comparing Publish-Subscribe in CORBA and NDDS

Both CORBA and NDDS provide publish-subscribe capabilities. However, what is the primary communication mechanism in NDDS is an add-on service in CORBA. The CORBA Event and Notification services provide publish-subscribe semantics. They are standard CORBA layered services, meaning that they are built on top of the object request broker (ORB) and general inter-ORB protocol (GIOP). The limitations of the CORBA Event service are discussed by D.C. Schmidt and C. O'Ryan [24]. The CORBA Notification service is based on the Event service, and is designed for just a few event channels and receiver-side content filtering. NDDS is designed for many topics and effectively filters on the publisher side.

OMG has recently published the Data Distribution Service for Real-Time Systems Specification for CORBA [25], which specifies publish-subscribe semantics. This is a *specialized CORBA specification*, meaning that it does not mandate the use of a layered implementation that is based on ORB and GIOP. RTI, who developed NDDS, is one of the primary contributors to this standard. The specification identifies different profiles, representing levels of compliance. The latest version of NDDS complies with several of the profiles identified in the DDS specification. The version of NDDS we used was not compliant with the standard, which was still being finalized when we were considering middleware.

While it is possible to do asynchronous messaging with CORBA, or to use an add-on service to approximate Publish-Subscribe semantics, the model doesn't

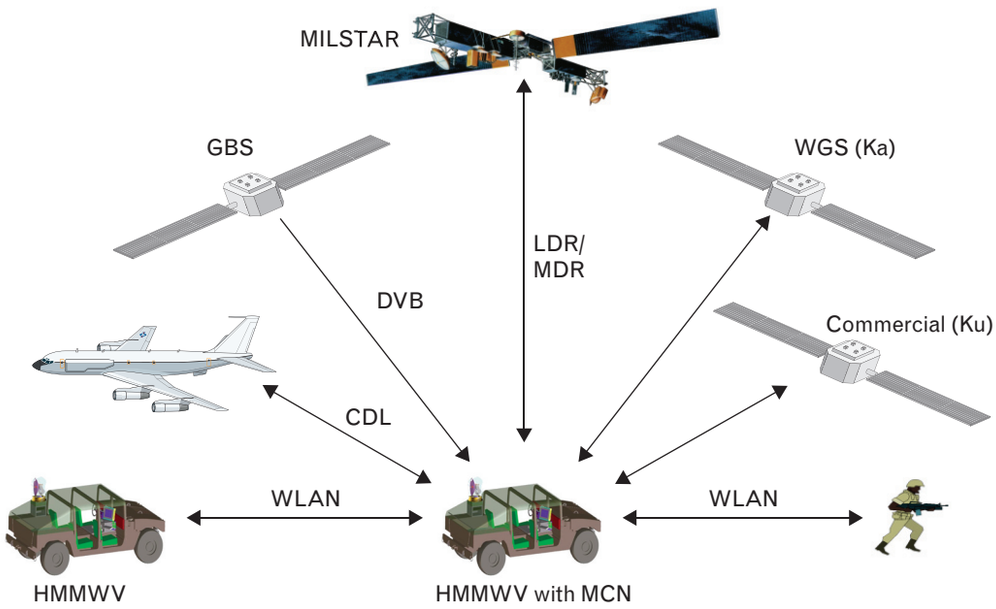


FIGURE 8. Mobile communication node (MCN) on a high-mobility multipurpose wheeled vehicle (HMMWV)–system content, showing the communications links between the MCN and other systems. MILSTAR stands for military strategic tactical and relay satellite connected through low and medium data rate (LDR/MDR) links; GBS is a global broadcast system communicating through the DVB digital video broadcast link; WGS is a wideband gapfiller satellite; CDL is a converged data link; and WLAN is a wireless local area network.

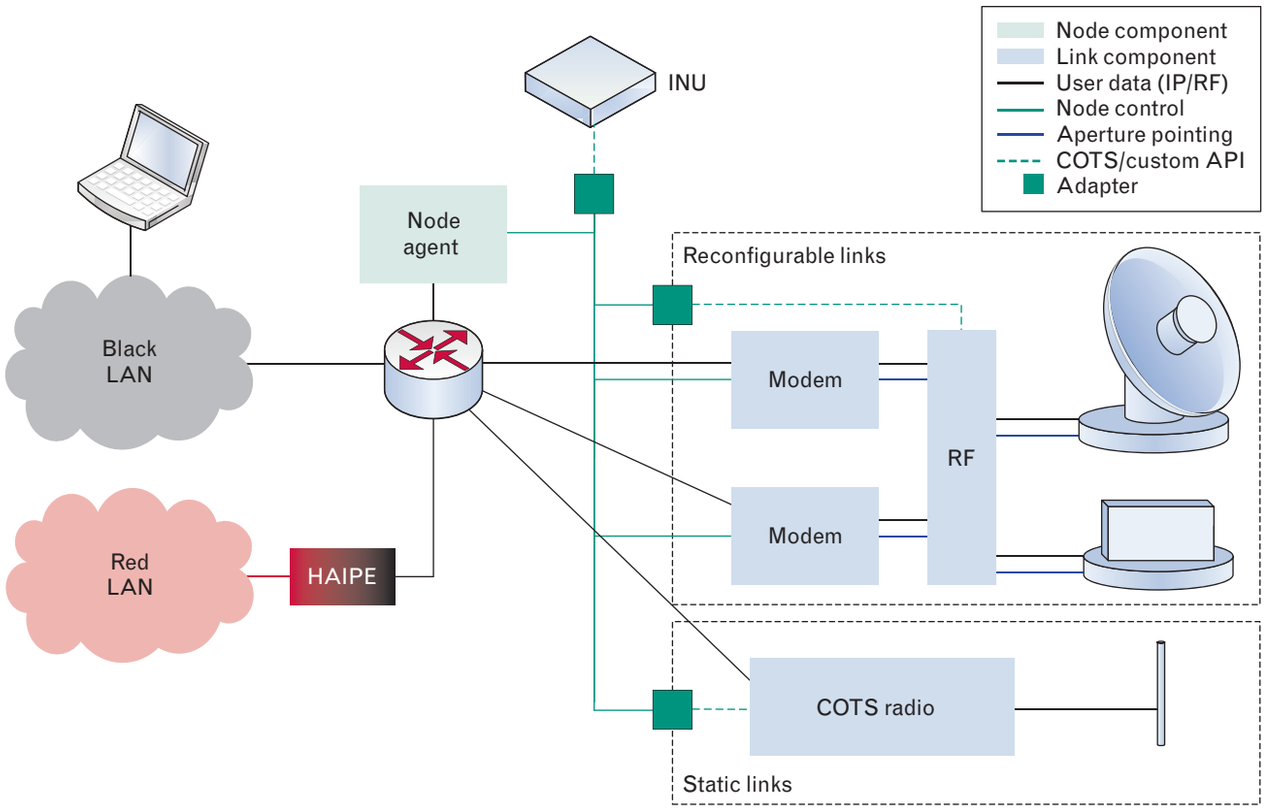


FIGURE 9. MCN system architecture, showing the top-level notational components required to realize the MCN. HAIPE is a high-assurance internet protocol encryption device. COTS/custom API stands for a customized commercial off-the-shelf application programming interface, LAN stands for local area network, and INU is an inertial navigation unit

seem conducive to the needs of DRE systems composed of cooperating processes such as the system we are building. On the basis of these considerations, we selected NDDS as the middleware for our software control plane.

System Context

The mobile communications node, shown in Figure 8, is a prototype vehicle-mounted communications system built to demonstrate how exploitation of an ensemble of networks containing links of various types (e.g., ground-to-space, ground-to-air, ground-to-ground) can provide for reliable, wideband, on-the-move communications. The prototype currently has some real-time requirements that are not particularly strict. There is some uncertainty as to which links the system will need to support in the future. Therefore, the system must accommodate the insertion of new COTS and/or custom link components, which may or may not have real-time control requirements. The system must make some basic services available to the different links, such as configuration, vehicle position, and spacecraft tracking.

System Architecture

The system architecture concept for our prototype mobile communications node is illustrated in Figure 9. The currently supported links are low and medium data rate (LDR/MDR), global broadcast system (GBS) receive-only, and wireless local area network (WLAN). While only three links are currently implemented (two reconfigurable and one static), the system supports the insertion of additional link components or COTS radios. The specific components of the system architecture are

1. *Node Agent.* This control component is responsible for configuring the other components of the node and monitoring their status as necessary. It may also respond to changes in status by reconfiguring or notifying other components. It may also reconfigure the router to attempt to reroute traffic destined for failed links.
2. *Reconfigurable Links.* These are links that can be

composed from available link components. Such links have three elements: a modem, a radio frequency (RF) component, and an antenna. In the prototype, either a GBS or a military strategic tactical and relay satellite (MILSTAR) link can be created by using the same RF and antenna (but different modems).

3. *Static Links.* These links are not dynamically reconfigurable by the Node Agent in real time. A node may contain multiple such links. Each link transmits and receives data by using its own dedicated radio containing both an RF module and an antenna.

4. *Inertial Navigation Unit.* This package is used by the node to determine its location and orientation in inertial space.

5. *Router.* The router in the node performs store-and-forward routing of IP-encapsulated data packets. The router is connected to each modem via an Ethernet cable. We made the assumption that COTS modems or radios would have an Ethernet data port. We accepted the constraint that custom modems would provide an Ethernet data port.

6. *High-Assurance Internet Protocol Encryption*. This device, also called HAIPE, performs the encryption and decryption required to support connection of a classified local area network (LAN) to the node.

7. *Unclassified and Classified LANs.* Users connect hosts to these networks to run various applications.

Our hardware team made several key hardware and platform decisions that enabled us to consider the use of middleware for this system. For the node agent and other node components we selected a CompactPCI backplane with Ethernet support and an Intel x86 single-board component (SBC) running the Linux operating system. Decisions on future modem applications included using PowerPC and VxWorks, which are supported by several middleware vendors, and using a general-purpose protocol as a modem controller to hide special-purpose protocol modem components from the rest of the node architecture. The first two decisions are realized in the existing prototype. The second two are design constraints on the custom modems we may build in the future. Obviously, there may be requirements for some future modem that invalidates either or both of these decisions, and so we are gambling to some extent.

Software Architecture

The software control architecture for our mobile-communications-node prototype is illustrated in Figure 10. NDDS is running over the CompactPCI backplane. There are currently two Intel x86 SBCs plugged into the backplane: the node control processor and the space tracking processor. Empty slots are available for additional SBCs or custom boards to support the insertion of additional modems.

Several legacy subsystems from our previous work have been integrated into the prototype. The MILSTAR-on-the-move (MOTM) terminal was developed by using the custom remote-invocation framework mentioned previously. This is a completed terminal for receive-only MDR and LDR MILSTAR protected satcom that is plugged via Ethernet cable into the node backplane. It uses remote invocation to interact with the physics package unit over this Ethernet link. It is also connected to the node controller via serial cable, to facilitate the operation of the LDR adapter [17], which publishes signal-strength metrics retrieved from the LDR modem.

The second link selected for insertion into the prototype was a COTS GBS receive capability, which uses the same RF and antenna positioner as the MOTM terminal. The inertial navigation unit (INU), RF, and antenna subsystem from that system were integrated as a separate subsystem into the prototype to allow it to be shared with GBS. The antenna control processor is connected via serial interface to the space-tracking processor. The INU device is controlled by the physics package unit, which is adapted to NDDS by the Attitude Heading Reference System (AHRS) adapter. The legacy antenna subsystem is controlled by the physics package unit via the antenna adapter. The RF interface module was also built as an adapter. The tracker is responsible for spacial tracking of the GBS satellite. We designed a digital video broadcasting (DVB) adapter for control of the GBS DVB receiver, but found that GBS DVB had a convenient web interface for setting modem parameters. Once set, these parameters become the default. Therefore, the DVB adapter is not currently necessary.

The network agent, router manager, and wireless adapter, illustrated in Figure 10, have not been built yet. They will be required for our experiments with fast rerouting, but are not currently necessary for the operation of the prototype. The network agent will receive signal-strength metrics from the links (e.g., the wireless adapter). It will then use the router manager to reroute traffic destined for failed links. Until we begin experimentation with fast rerouting, static routes and links are

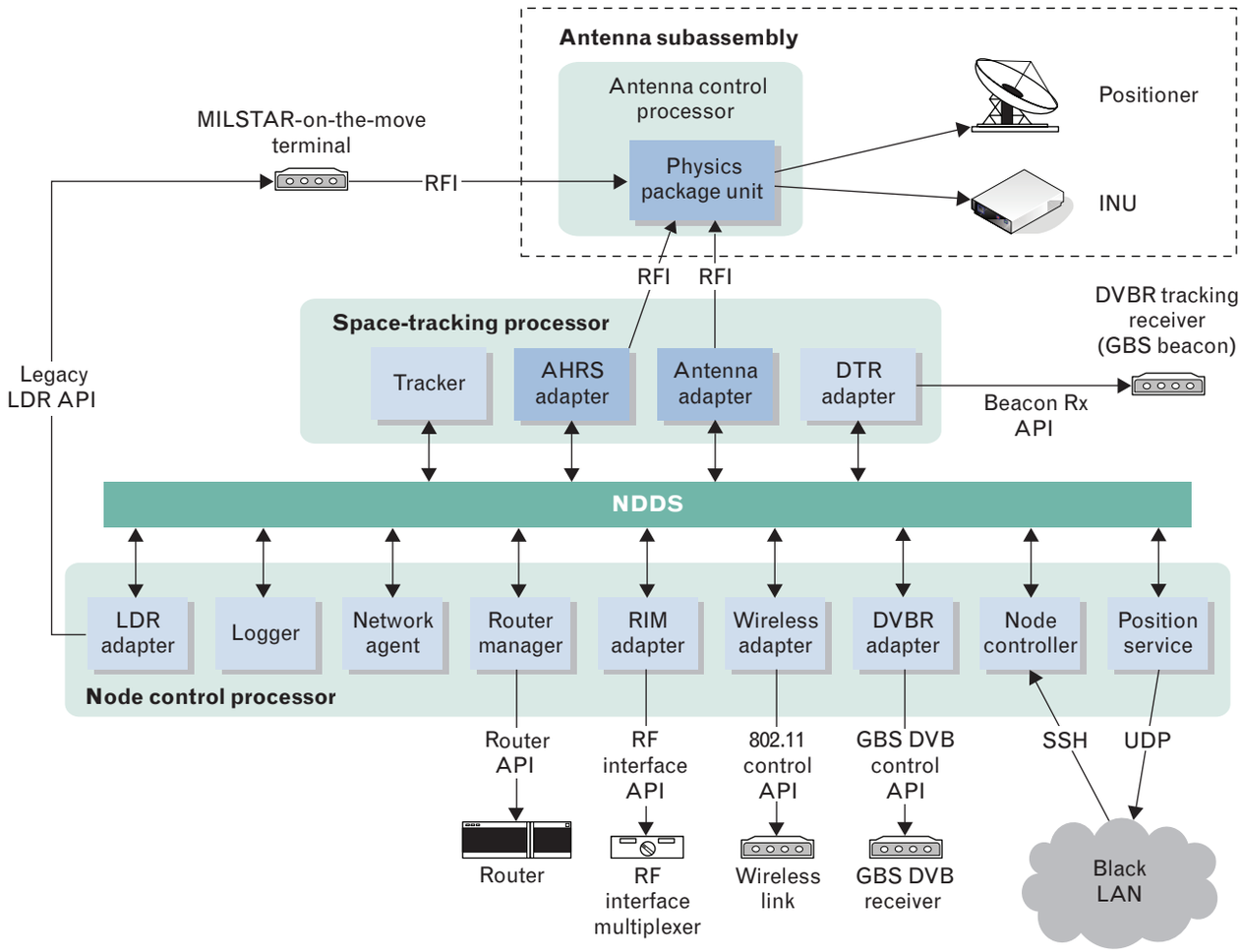


FIGURE 10. MCN software architecture. This is a run-time view illustrating how the various top-level components of the MCN software are connected via the Network Data Distribution Service (NDDS) to peripheral devices. AHRS stands for attitude and heading reference system; DTR is data terminal ready, RIM is an RF interface module; SSH and UDP are secure-shell and user-datagram protocols; DVB is digital video broadcasting; and DVBR is a DVB receiver.

sufficient. This simplification of the problem demonstrates how we are benefiting from one of the advantages of publish-subscribe, the ease of system evolution.

Software Design

Component design with publish-subscribe requires a different approach than its current object-based invocation. In object-based invocation, one of the consequences of distribution transparency is that a component is typically structured as an object. With publish-subscribe, it makes more sense to structure components as state machines that change state according to events. Object-oriented design still plays a vital role in building the elements that make up a component.

High-level design using NDDS is mostly about partitioning the data to be exchanged into topics. No remote methods are invoked or object-oriented interfaces specified in IDL, only topics that consist of a name and a type. A topic type is defined in IDL, and code generation is used to create the appropriate structure (or class) definition and marshalling code for the target language. The topic name is used to differentiate topics of the same type. For example, the topic in Figure 7 has a type position, and a name VehiclePosition. We could also create another topic named SatellitePosition, which has the same type.

Figure 11 illustrates how the topics currently implemented in the node prototype map to publishers and subscribers. The individual topics are described in Tables 2 and 3 and in the following sections. The logger, which

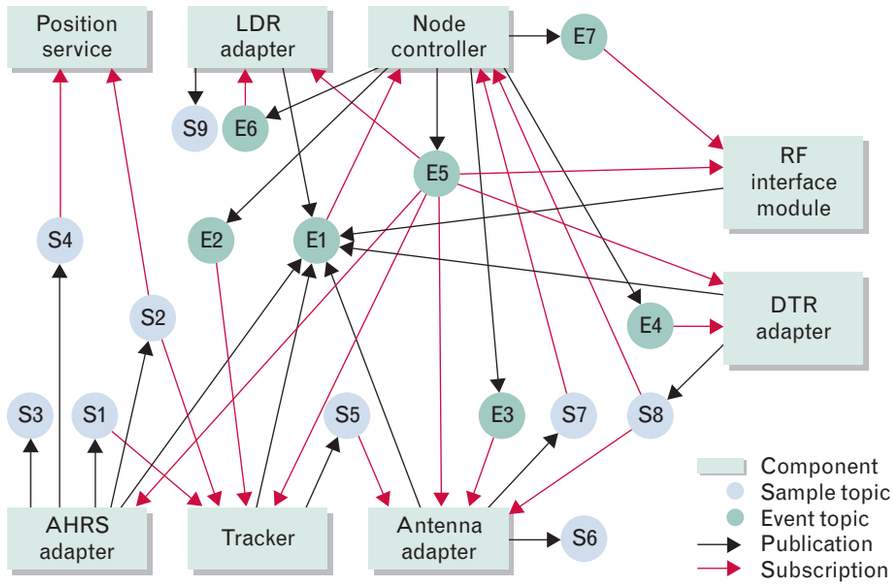


FIGURE 11. MCN software topic mapping, showing the mapping of publishers and subscribers to topics. Sample topics are listed in Table 2, and Event topics are listed in Table 3.

is not shown in the figure, can potentially subscribe to all topics.

Just as interface methods in object-based invocation can be notionally divided into commands and queries, topics types in publish-subscribe can be divided into two subtypes: samples and events.

Samples. Samples are periodic, typically representing measurements of the environment. Because of their periodic nature, samples can be sent *best effort*; reliable delivery is not necessary. The consequences of losing a single update are relative to the rate of publication, and since samples are typically fairly high rate, the loss of one is not a serious problem. Refer to Figure 11 for a mapping of publishers and subscribers to the sample topics listed in Table 2.

Events. Events are aperiodic, representing unique changes in component or system state. Events can be commands, parameter updates, status updates, or exception notifications. Because they represent unique changes in component or system state, the loss of a single event could cause a serious error. Therefore, events are sent by using the reliable mode provided by the middleware. Status messages are also set with a time-to-keep quality-of-service value. This information enables late-joining subscribers to get the complete sequence of status events for all components in the system. Refer to Figure 11 for a mapping of publishers and subscribers to the event topics listed in Table 3.

Within the node prototype, several topics appear to have no current subscribers. These topics may be designed for functionality that is to be added, like fast rerouting, or for debugging purposes. An update that is published when there are no subscribers to that topic gets dropped by the middleware, and no network traffic is generated. It is also possible for a component to subscribe to a topic for which there are no publishers. It simply means that the callback for that subscription will never get called.

Some topics are being published to more than one subscriber. These are currently being sent unicast, because the difference in end-to-end latency between one-to-one unicast and one-to-four unicast is on the order of 80 μsec [26]. Granted, these are vendor performance metrics, but they would need to be off by several orders of magnitude to be a concern. If it became a timeliness concern, we could switch to multicast to optimize the publications.

Some command publications have a device ID field, which represents a command directed at a particular object. How is this any different than invoking an object? For one thing, invocation contains an implicit assumption that the object exists. Sending an event with a device ID is like saying "if a device with this ID exists, it should perform this command." There is no presumption of existence. This decoupling is what contributes to the ease of system evolution.

Table 2. Sample Topics*

UtcTime (S1)	Universal Time Coordinated (UTC) time, published at 10 Hz for time synchronization.
AhrsLocation (S2)	Longitude, latitude, and altitude of the vehicle published at 1 Hz.
AhrsDisplacement (S3)	Displacement of the vehicle from a known point published at 1 Hz.
AhrsVelocity (S4)	Three vectors indicating the vehicle-position rate of change, published at 1 Hz.
AntennaReferenceAngle (S5)	Pointing angle for the antenna subsystem.
AcquisitionMetric (S6)	Energy measurement to aid in debugging acquisition problems, published as the antenna scans a particular pattern.
AntennaAngles (S7)	Current reference, dither, offset, and bias angles for the antenna.
DtrSamples (S8)	Received power and carrier-to-noise ratio published at 10 Hz.
LdrEnergyMetric (S9)	Measurement of LDR energy from the MILSTAR terminal.

* Labels S1 through S9 are coded to locations in Figure 11.

Client Services

We have recognized the need for three client services so far. A client service is defined as some service relating to the control of the node prototype that may be utilized by clients on the unclassified LAN.

Node Controller. The node controller provides a simple command-line user interface to the node, with shell features such as tab completion. Clients can use secure shell (SSH) for a secure remote login into the node control processor and launch the node controller process to send commands to devices and check their status.

Logger. As mentioned previously, the logger is used primarily for system monitoring and debugging purposes. It is capable of subscribing to one or more topics, which is a little like using a tool to snoop network

Table 3. Event Topics†

DeviceStatus (E1)	Typically, devices that are commanded will send a notification of update in device status. This notification includes a device ID (to distinguish the device, since many devices publish to this topic), a status code or error code, and an optional string (for reporting any details to the user).
TrackCommand (E2)	Issues commands to the Tracker. A TrackCommand update contains a device ID, a command, and a satellite name. Commands to start, stop, suspend, or resume.
AntennaCommand (E3)	Issues commands to the Antenna subsystem. An AntennaCommand consists of a device ID, a command, and a bias angle used for acquisition. Commands to start and stop acquisition, and set bias angle.
DtrParams (E4)	This topic is used to update various parameters on the DVB tracking receiver. Publishing an update to the topic sets the new parameters on the device.
DeviceCommand (E5)	This topic is used for basic operations common to all devices, like reset and kill. A DeviceCommand consists of a device ID and a command.
LdrCommand (E6)	Commands to enable/disable energy metrics, antenna commands, encoding, and interleaving, as well as commands to set downlink mode and interleaver size.
RimCommand (E7)	Commands to switch RF Interface Multiplexer to LDR/MDR or GBS.

† Labels E1 through E7 are coded to locations in Figure 11.

traffic. The logger allows a user to see what one or more components in the system are publishing at any given time. Several instances of the logger can be launched and set to log different sets of topics.

Position Service. The position service subscribes to the AHRS adapter's publications for the vehicle's position and velocity, and can be configured upon launch to send these data in a custom user-datagram protocol (UDP) format to designated client IP addresses on the user data network. These clients can run Precision Lightweight Global Positioning System Receiver (PLGR) simulator software called PLGRsim, which was developed by Mark Smith in the Laboratory's Wideband Technical Networking group. PLGRsim enables clients on the LAN to run applications that require vehicle position via PLGR, such as FBCB2.

Node Controller—Drivers and Adapters

The Node Controller is the node user's interface for commanding the various elements of the node. It can be used for such node functions as switching between LDR/MDR and GBS receive, activating GBS satellite tracking, and querying the status of any device. The node controller has a collection of drivers, which act as proxies for the adapters in the system.

Because of the loosely coupled design, the node controller process can be killed without any negative impact to the operation of the node prototype. When the node controller is relaunched, it retrieves the cached status of each device in the system, and picks up exactly where it left off. Figure 12(a) illustrates how the node controller component uses a driver object to communicate with a device through its associated adapter component.

Drivers are used to send command and parameter updates to devices. Drivers also represent a cache of an active device's last status, exception, and sample publications. The user can retrieve this information with a simple command. Drivers are also used by a test case in our automated test suites to verify the correct behavior of the adapter, as shown in Figure 12(b).

Adapters are used to adapt the native interface of a device to published events and samples. Adapters in test cases are run in a dummy mode, which does not actually invoke the device. This facilitates testing of the device-adapter interface for each device, even when the device is not available. Integration tests between adapters and devices are performed manually.

Logging Notification Yields Behavioral Decoupling

The logging of events during the execution of software applications has become an area of increasing interest and activity. Over the past ten years, flexible and con-

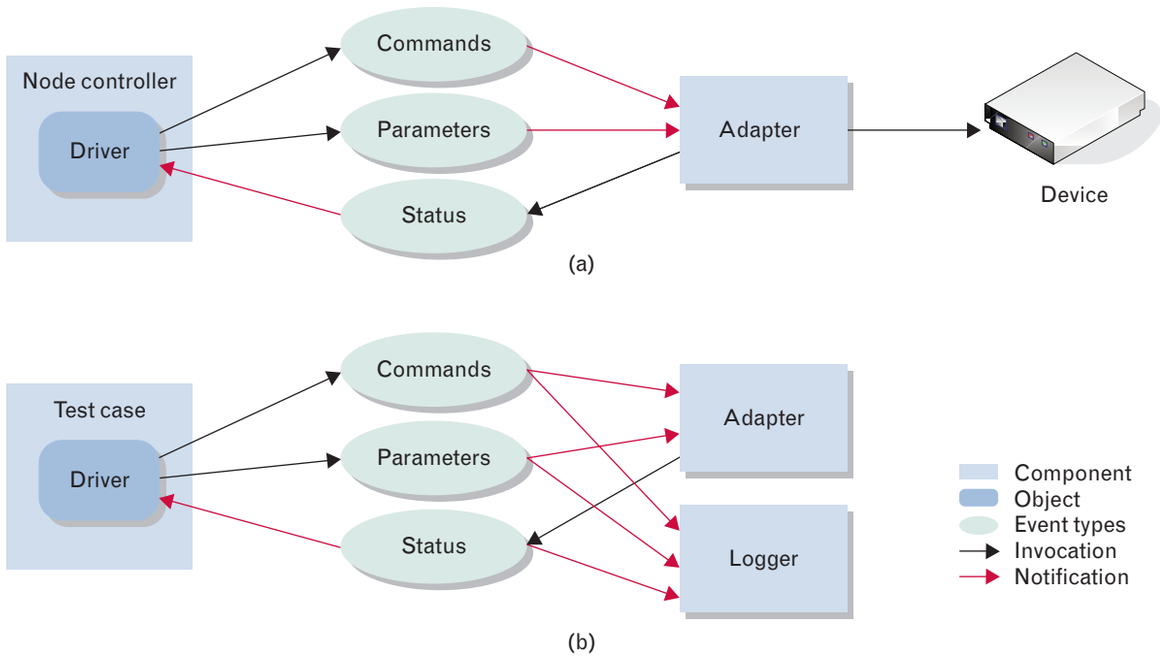


FIGURE 12. Drivers and adapters. (a) Enabling the node controller to manage devices; (b) test-harness facilitating the testing of events.

trollable freeware logging packages have been developed and used in the construction of large software projects [27]. Modern loggers are flexible enough to allow event data to be logged to any combination of files, databases, or custom listeners. They are controllable enough that their output can be selectively enabled and/or redirected at run time on the basis of parameters of each individual events, such as program component or debug level.

The benefits of easier debugging and data gathering are not lessened in the case of distributed applications, but they can be more difficult to realize. Consider a system consisting of processes A and B executing on separate machines and communicating via invocation. A run-time logging service that accepts logging invocations from both A and B is, in effect, an additional distributed component C. This logging service adds to the complexity of the system's interactions and to the risk of deadlocks, bottlenecks, and priority inversions. Furthermore, executing such a system under different logging configurations can result in different behaviors when the interactions between A and B are sensitive to the temporal effects of interactions with C. Bugs that appear or disappear on the basis of logging or debugging efforts are known as heisenbugs (after Heisenberg's Uncertainty Principal), and are notoriously difficult to isolate [28].

The decoupling of A and B from C removes this added risk by using publish-subscribe instead of invocation. Conceptually, A and B always publish all of their events. The fact that C subscribes or does not subscribe to a given set of logging events does not affect A or B's behavior. Furthermore, it becomes possible to log all of the communications between distributed components in the system by configuring the logger to record every publication of any sort. It is also possible to have any number of loggers running at once, on any number of attached systems, recording any subset of the events and communications in the system, with no effect on the behavior of the other components. Certainly this multiplicity of functions could have a significant impact on timeliness, if sufficient subscribers were added. However, this concern is mitigated by optimizing publications with multicast.

Our architecture always executes one logger to record all events in human-readable format. Users wishing to use the system to gather data may launch additional logger instances themselves to record specific publications in human- or machine-readable format.

C++ Exceptions: Stack Traces and Notification

We augment the standard C++ exception behavior on Linux in the following four ways.

1. *Stack traces.* Our exceptions generate stack-trace strings containing the symbol name and address associated with each unwound stack frame, by using the GNU C++ Compiler's `(gcc)` `backtrace` function. We translate these addresses when available, to source file and line numbers by spawning an external `addr2line` process, which is included on most Linux systems, and having it parse the running binary. The resulting string included in each exception is similar in appearance to a Java stack trace.

2. *Signals.* We instantiate signal handlers for a set of signals, which upon execution throw exceptions. On modern Linux systems, this instantiation correctly unwinds the stack as the signal handler stack is situated upon the regular application stack frame. For example, a null pointer dereference will generate a full stack trace with line numbers right up to the receipt of a segmentation violation signal.

3. *Uncaught exceptions.* We install handlers to display the textual information for all uncaught exceptions.

4. *Notification.* We provide convenient macros so that adapter-derived objects can automatically publish any exception generated by the contained code fragment.

Our code was inspired by an IBM developer Works article and does much to ease the C++ development process [29]. By tying notification into our exception handling, we made it possible that both the user at the console at the time of failures and the developer reading log files after the fact receive sensible, context-rich information.

Future Work

We want to consider the problem of dynamic routing on shorter time scales than are currently supported by typical routing protocols such as OSPF. Adding a custom modem with more stringent hard real-time control requirements using publish-subscribe would be an interesting evolution of the system architecture. We would like to consider a software design that implements components as hierarchical state machines, as described in Reference 30.

Obviously, as we add more links to the prototype, the fast rerouting problem gets more interesting. If we add directional links, then topology decision making would

be interesting. Also, we're interested in the capability of reconfiguring links to take full advantage of available resources. We would like to spend some time modeling link state and decision making, and developing and implementing a link abstraction to create a unified control interface for link components.

Conclusion

Military communications systems can be constructed by using many different software technologies. The question we have considered is which architectural approach for middleware best supports the goals of DRE systems. Not all DRE systems have the same requirements or quality attribute goals. However, some qualities are essential to the definition of a DRE system. We have considered two architectural approaches in the context of these qualities.

We observe that invocation makes component programming slightly easier and system integration much harder, while publish-subscribe makes component programming slightly harder and system integration much easier. Since RPC was developed, many advances, such as languages and library support, have made programming easier. Many modern-day software projects struggle in the integration phase. It therefore makes sense to choose an architectural approach that trades off some component simplicity for integration simplicity. Because components cannot be integrated until they are built, integration naturally falls after component construction. Selecting an approach that increases integration complexity shifts uncertainty and risk to the latter stages of a project. On the other hand, choosing an approach that complicates construction but simplifies integration front-loads the risk and uncertainty. This architectural approach should lead to a progressive reduction of uncertainty throughout the project schedule.

We observe that publish-subscribe, a distribution system implementation of the implicit-invocation architectural style, promotes reuse and extensibility. By decoupling communicating components, this approach insulates them from one another's behavior, timeliness, and predictability concerns. This decoupling also removes deadlock as an integration problem, improving the composability of components developed for publish-subscribe.

While systems can certainly be built by using a number of architectural approaches, we must consider some inherent trade-offs. We have shown, and we believe, that publish-subscribe demonstrates some very attractive qualities as a middleware for DRE systems.

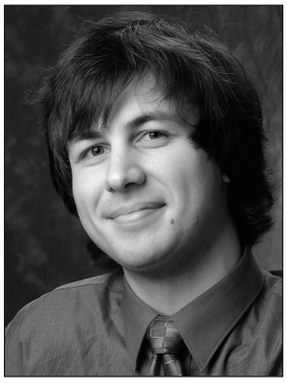
REFERENCES

1. Software Communications Architecture Specification Version 2.2, Joint Tactical Radio Systems Joint Program Executive Office, jtrs.spawar.navy.mil/sca/downloads.asp?ID=2.2.
2. Free On-Line Dictionary of Computing, www.foldoc.org.
3. A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Comput. Syst.* **2** (1), 1984, pp. 39–59.
4. R. Guerraoui and M.E. Fayad, "OO Distributed Programming Is Not Distributed OO Programming," *Commun. ACM* **42** (4), 1994, pp. 101–104.
5. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline* (Prentice Hall, Upper Saddle River, N.J., 1996), p. 165.
6. Common Object Request Broker Architecture: Core Specification Version 3.0.3 (see Section 11.3.8.1, Thread Policy), Object Management Group, Mar. 2004, www.omg.org/docs/formal/04-03-12.pdf.
7. www.dacs.dtic.mil/topics/edcs/demodays.shtml.
8. N. Kaveh and W. Emmerich, "Deadlock Detection in Distributed Object Systems," *Proc. 8th European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)/9th ACM Special Interest Group on Software Engineering (SIGSOFT) Symp., Vienna, 11 Sept. 2001,* pp. 44–51.
9. N. Kaveh, "Using Model Checking to Detect Deadlocks in Distributed Object Systems," *Engineering Distributed Objects,* W. Emmerich and S. Tai, eds. (Wiley, Chichester, U.K., 2000).
10. P. Inverardi and S. Uchitel, "Proving Deadlock Freedom in Component-Based Programming," *Fundamental Approaches to Software Engineering: 4th International Conference, FASE 2001: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001: Genova, Italy, April 2–6 2001: Lecture Notes in Computer Science (LNCS)* **2029,** pp. 60–75.
11. F.P. Brooks, "No Silver Bullet—Essence and Accidents of Software Engineering," *Proc. Int. Federation for Information Processing (IFIP) Tenth World Computing Conf., Dublin, 1–5 Sept. 1986,* pp. 1069–1076.
12. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch or Why It's Hard to Build Software from Existing Parts," *Proc. 17th Int. Conf. on Software Engineering (ICSE), Seattle, 23–30 Apr. 1995,* pp. 179–185.
13. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice,* chaps. 4 and 5 (Addison Wesley, Reading, Mass., 1998), pp. 71–130.
14. M. Shaw and D. Garlan, *Software Architecture*, p. 172.
15. Ibid., p. 173.
16. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns,* vol. 1 (Wiley, Chichester, U.K., 1996).
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, Reading, Mass., 1995).
18. Real-Time CORBA Specification Version 2.0, Object Management Group, Nov. 2003, www.omg.org/cgi-bin/doc?formal/03-11-01.pdf.
19. Real-Time CORBA Specification Version 1.2, Object Management Group, Jan. 2005, www.omg.org/cgi-bin/apps/doc?formal/05-01-04.pdf.
20. Minimum CORBA Specification Version 1.0, Object Management Group, Aug. 2002, www.omg.org/docs/formal/02-08-01.pdf.
21. T.H. Harrison, D.L. Levine, and D.C. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service," *Proc. 12th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, Atlanta, 5–9 Oct. 1997,* pp. 184–200.
22. P. Gore, I. Pyarali, C.D. Gill, and D.C. Schmidt, "The Design and Performance of a Real-Time Notification Service," *Proc. 10th IEEE Real-Time and Embedded Technology and Applications Symp., Toronto, 25–28 May 2004,* pp. 112–120.
23. Real-Time Publish Subscribe Protocol (RTPS) Wire Protocol Specification Version 1.0, IEC/PAS 62030 International Electrotechnical Commission, domino.iec.ch/webstore/webstore.nsf/artnum/033452.
24. C.O. Ryan, D.C. Schmidt, and J.R. Noseworthy, "Patterns and Performance of a CORBA Event Service for Large-Scale Distributed Interactive Simulations," *Int. J. Comput. Syst. Sci. Eng.* **17,** Mar. 2002.
25. Data Distribution Service for Real-Time Systems Specification Version 1.1, Object Management Group, Dec. 2005, www.omg.org/docs/formal/05-12-04.pdf.
26. NDDS version 3.0i Data Delivery Performance, Real-Time Innovations, Mar. 2002.
27. Log for C++, Open Source Technology Group, log4cpp.sourceforge.net.
28. Heisenbug, Wikipedia, en.wikipedia.org/wiki/Heisenbug.
29. S. Agrawal, "C++ Exception-Handling Tricks for Linux," IBM developerWorks, 23 Feb. 2005, www-128.ibm.com/developerworks/library/l-cppexcep.html?ca=dnt-68.
30. M. Samek, *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems* (CMP Books, San Francisco, 2002).

J. ĐARBY MITCHELL
is an associate staff member in the Wideband Tactical Networking group. His current research includes software architectures, design patterns, and middleware for distributed real-time embedded systems. He received a B.S. degree in computer science at the University of North Carolina at Wilmington, and an M.S.E. degree in software engineering from Carnegie Mellon University.

MARC L. SIEGEL
is a software engineer at Liquid Machines, Inc.; he was formerly an assistant staff member of the Wideband Tactical Networking group. His current research interests include static analysis of concurrency in distributed systems, expert systems, and advanced C++ techniques. He earned a B.S. degree in computer science from Brown University.

M. CURRAN F. SCHIEFELBEIN
is an associate staff member in the Biodefense Systems group; she was formerly a member of the Wideband Tactical Networking group. Her current research studies the performance of distributed wireless sensor networks for advanced warning of biological or chemical attacks. She earned her A.B. and Sc.M. degrees in computer science at Brown University.



ARMEN P. BABIKYAN
is an associate staff member in the Wideband Tactical Networking group. His current research interests broadly span topics in computer networks, distributed systems, wireless networks, grid computing, and asynchronous networking. He received B.S. (with honors) and M.S. degrees in computer science from the University of Massachusetts at Amherst.