Enabling Modularity in the Littoral Combat Ship

Gordon Hunt
TRG Systems

Frank Crow
Progeny
LCS CSA Prime

Paul Pazandak, Ph.D.
Stan Schneider, Ph.D., CEO
Real-Time Innovations

The views expressed in this paper are those of the author and do not reflect the official policy or position of the Department of the Navy, the Department of Defense, or the U.S. Government.

Abstract

The US Navy began extensive efforts to examine the best way to structure future surface combatant forces in response to the May 2002 Defense Planning Guidance (DPG) and the Quadrennial Defense Review (QDR). The Secretary of the Navy and the Chief of Naval Operations (CNO) concurred on a program requirement that became the Littoral Combat Ship (LCS) program in Spring 2002.

The Program Executive Office (PEO) LCS was created to align several program offices into a single consolidated PEO which focused entirely on delivering the LCS program in July 2011. The program focuses on three mission areas within the Littoral Surface Warfare operations which emphasize the prosecution of Surface Warfare (SUW), Mine Countermeasures (MCM) and Anti-Submarine Warfare (ASW).

The design of the LCS permits the use of reconfigurable modular mission packages[1] in support of these missions. Thus, these ships are essentially "Seaframes" that provide the integrated systems required to support the various mission packages. These systems include command and control, computers, and intelligence (C4I) along with an integrated unmanned vehicle control system and common resources such as power, compressed air and water, etc.

It is essential to provide an open technical architecture and a modular design that provides flexibility and ease of upgrade in order to support the rapid installation and integration of these mission packages. Standards-based interfaces between mission packages and the Seaframe are also necessary to allow these systems to be developed independently.

This paper focuses on the justification for, and tenets of, an open architecture in order to provide the modularity required within LCS. We then delve into the details of the architectural design, decisions, and lessons learned building the Common Software Architecture (CSA) for LCS. Finally, we provide an introduction to the underlying technology that provides the foundation for the open architecture of LCS CSA, and many other DoD programs – the Object Management Group (OMG) Data Distribution Service for Real-Time Systems[2] (DDS), a DoD-mandated standard.

1 Navy Open Architecture

1.1 Problems & Motivation

There is an ongoing and dramatic change in defense procurement acquisition; in lieu of systems integrators, the DoD is starting to define and maintain flexible system architectures for the electronic and software systems they wish to procure. By taking *architectural management* of the integration infrastructure and the integrated system of systems architecture, the DoD seeks to evolve the acquisition of defense capabilities towards a strategy that promotes open competition, cost control, innovation, and the rapid replacement and upgrade of capabilities to address warfighter needs.

Implementing these system-of-systems while satisfying the combined system attributes of performance, scalability, and reliability is challenging.

Adding additional software requirements including interoperability, flexibility, modularity and portability makes the problem even more difficult. The Navy is not undertaking these challenges for the sake of the software architecture alone, but rather to enable fiscally-constrained fielding of dominant capabilities that leverage a large industry base for technical innovation[3]. The Navy has a long history of open architecture efforts[4] so it is important to understand what is different.

To date, it has been the various integrators that have realized many of the benefits of open architecture approaches. However, this has resulted in vendor-specific product lines and platform-specific solutions which dramatically limit the Government's ability to bring multiple vendors and new capabilities together affordably. This is especially problematic for the LCS platform: it has two distinct Seaframe variants, and multiple mission packages that should be interchangeable between them. This presents a significant configuration and integration challenge. Certainly brute force could work, but what about the mission modules yet to be identified or capabilities that have already been fielded on other platforms? Current open architecture approaches do not put enough specificity into the Government's hands to get ahead of this integration challenge and certainly do not address the cost constraints of integration. It is a problem larger than interface specifications, data rights, COTS standards, or source code.

1.2 Approach & Solutions

The Navy has employed an Open Systems Architecture[5] (OSA) strategy which covers the complete life-cycle of technology and capability acquisition. Up to now, defense procurement agencies have been asking for open architecture solutions from their supply chains, and the result has been their adoption of open standards, modular system frameworks, and Commercial-Off-The-Shelf (COTS) standards-based technologies. All of these are *absolutely* components of OSA, but unless they are brought together within an open infrastructure architecture, an open data model, appropriate data rights, properly incentivized contracting, and metrics-based programmatic oversight, realizing the benefits of OSA principles will be a long road.

To be meaningfully interoperable, different systems built at different times, with different hardware, different software architectures, different technologies and different uses of the data and system information must be cost-effectively integratable. This does not mean, nor imply, interoperability by commonality, plug-and-play integration, or any other over-used promise of open architecture. Furthermore, this is not implying that all integration must be government-led. OSA is about integration without ambiguity, and the ability to achieve interoperability between systems – at scale, repeatedly, across system ownership boundaries.

The question and the solution lies in the definition (the technical specification as well as the IP and data rights strategy) of these system ownership boundaries. At these boundaries one can address each specific system's software architecture non-functional requirements (Table 1) and define known, separable, testable, and independently acquirable system functions.

OSA defines several immediate key actions and steps.

- Implement the coordinated set of business changes that improve competition, incentivize better performance, and deliver capability more rapidly;
- Construct a limited number of technical reference frameworks to immediately support improved competition and ultimately enable enterprise re-use;
- Develop an Execution Guidebook for this strategy; and,
- Lead and guide training the workforce on OSA implementation.

Table 1. Critical non-functional system software-architecture requirements.

Non-Functional Requirement	Definition
Interchangeability	To put each of (two things) in the place of another, or to be used in place of each other.
Integratability	To form, coordinate, or blend into a functioning or unified whole. To incorporate into a larger, functioning or unified whole.
Replaceability	One thing or person taking the place of another especially as a substitute or successor.
Extensibility	The ability to add new components, subsystems, and capabilities to a system.
Interoperability	The ability of systems, units, or forces to provide services to and accept services from other systems, units, or forces, and to use the services so exchanged to enable them to operate effectively together.
Portability	Abstraction of application logic and system interfaces to effect the usability of the same software in different environments
Modularity	A logical partitioning of the software design that allows complex software to be manageable for the purpose of implementation and maintenance

OSA also supports the recently enacted National Defense Authorization Act which requires modular open systems approaches in acquisition programs. An excerpt of HR 3979 Section 801 MODULAR OPEN SYSTEMS APPROACHES IN ACQUISITION follows:

(2) OPEN SYSTEMS APPROACH.—The term ''open systems approach'' means, with respect to an information technology system, an integrated business and technical strategy that—

(A) employs a modular design and uses widely supported and consensus-based standards for key interfaces;

(B) is subjected to successful validation and verification tests to ensure key interfaces comply with widely supported and consensus-based standards; and,

(C) uses a system architecture that allows components to be added, modified, replaced, removed, or supported by different vendors throughout the lifecycle of the system to afford opportunities for enhanced competition and innovation while yielding—

(i) significant cost and schedule savings; and

(ii) increased interoperability.

What does this all mean? An Open Systems Architecture is an architecture derived from a set of architecture design decisions that provide architectural management to the organization acquiring the system in order to assure that the resulting architecture meets the business, technical, and regulatory requirements of the acquiring organization.

Practically, it means a system integrator still takes responsibility for integrating all the sub-systems together, but against an architectural specification and infrastructure governed by the acquisition program office. These specifications decouple applications from deployment specific technologies and platform specific concerns, provide common functions and capabilities, and most importantly, clearly specify the data in decoupled formats from system and software application implementations. The specifications are *data-centric* in that they promote the data as the primary point of integration and interoperability.

Interoperability, along with other system non-functional requirements, is being brought to the top line as a delivery requirement where interoper-

ability is more than a common infrastructure, common messages, or COTS and standard-based technologies. The DoD is mandating the data-centric architecture of systems they wish to procure with a common systems architecture that clearly and unambiguously describes and document the data, its structure, its context, and its behavior.

In the context of LCS and the Common Software Architecture (CSA), the construction of the technical reference framework, the capability decomposition, and the service interface specification, was conducted by a team of industry and government engineers resulting in the data-centric CSA infrastructure specification. The technical reference framework addresses the system non-functional software requirements and provides a software environment for long-term cost effective integration. The careful separation of infrastructure and data provide additional benefits for testing, and more importantly, provide a clear boundary for data rights and intellectual property separation.

1.3 Remaining Challenges

Of the challenges remaining in the implementation of OSA, the technical side is likely the easiest to address. There are many efforts underway that are separating the software platform (operating system, IO, hardware) from the applications through standard APIs and system interface specifications. Yet, several efforts have evolved beyond this by also focusing on the *data* architectures[i]. This includes specifying the syntax, semantics, and behavior of the data. Semantics is tricky as it is traditionally captured implicitly in application logic, unstated system-specific use of the data in messages, or domain-specific labels and words. In order to be able to integrate systems-of-systems at scale, ***the semantic content of the data-centric specifications must be fully captured***.

A bigger challenge confronting the implementation of OSA are the acquisition processes and contracting flexibility needed to effectively compete and integrate from a broader industry base. While completely understood, this is likely going to take more effort.

1.4 Conclusion

The LCS CSA architecture and its implementation have adopted the tenets of OSA. It provides a great reference example as to how to decompose an open infrastructure into a set of reusable applications, implement a portable technical reference platform based on open-standards, and integrate rigorous model-based definitions of services and their data. In the next section we describe LCS CSA in more detail.

2 LCS Common Software Architecture

The Common Software Architecture (CSA) provides an operating environment and a common set of services for the mission packages that are supported by the LCS Seaframe. The goals of the LCS CSA were reviewed and approved by the LCS Community at its outset and are documented by the CSA Architecture Design Description (ADD).

Generally speaking, the goals of CSA are to provide a common environment for the various mission packages that may be installed on the Seaframe, and to facilitate an open business model that supports innovation. In order to refine that broad vision into a particular set of goals, an in-depth analysis of existing approaches and systems engineering artifacts was conducted.

That analysis identified the limitations of existing approaches, and the benefits of alternative approaches, which resulted in a clearly defined set of high-level design goals. While many design goals

[i] E.g., Unmanned Aerial System Control Segment (UCS), Future Airborne Capability Environment (FACE)

were identified, the particular design goals which enable modularity and thereby support the broader goals of the LCS program itself are described in the following section.

It is important to remember while considering the following design goals, the fundamental tenets that most broad Department of Defense architectural efforts seek to achieve:

- Support the functional needs of the system. In the case of LCS, the tactical needs of Mission Package warfighters;
- Minimize costs across all aspects of the life cycle. In this case, from Mission Package capability design through training and sustainment;
- Foster innovation and technical advances. For LCS, this applies to both the Mission Packages as well as the services provided by CSA;
- Enable Open Acquisition and avoid vendor lock. In this case, by ensuring LCS acquisition authorities are provided with clearly articulated and vendor-neutral CSA service definitions.

2.1 High-level Design Goals

In order for CSA to provide infrastructure and application services that would be useful and flexible enough to be used by the mission modules, it was determined that a component-based *Service Oriented Architecture* (SOA) should be implemented as a high-level design goal. To avoid the limitations inherent in certain SOA approaches it was necessary to adopt the tenets of the *Modular Open System Approach* (MOSA).

Large scale integration of diverse software systems often suffers from a measurable increase in integration complexity. This is due to the way that services and applications communicate with each other. A message-oriented communications backbone results in a web of connections which tends to expand exponentially as additional participants are added to the environment.

Message exchanges and request brokers have been used to mitigate these issues to some extent but ***the greatest reduction of integration complexity can only be realized by a data-centric publish/subscribe communication paradigm.*** Therefore, it was determined that the CSA services would utilize the OMG Data Distribution Service for Real-Time Systems (DDS) as its primary means of communication to more easily support large scale integration.

In order to fully support the open business model and reuse of CSA services, a *model-driven development* (MDD) approach was adopted. An MDD approach coupled with service-level requirements is used to clearly define interface and functional requirements in a conceptual manner. The Unified Modeling Language (UML) models that were created describe the services using conceptual data types and operations which can then be reused for any future implementations of CSA regardless of hardware or technology specifics.

In summary, the design goals for CSA included using:

- Component-Based Architectures
- A Modular Open System Approach
- Service-Oriented Architectures
- Data-Centric Publish-Subscribe Networking
- Model-Driven Development

The following subsections provide more details about these design goals, which are central to enabling modularity, and directly support the goals of the LCS Program.

2.1.1 Component-Based Architectures

A component-based architecture will allow Mission Package Application Software (MPAS) designers to select only those components needed to meet mission-diverse requirements. Component definitions include the data sent and received by the component, the services provided and required by the component, as well as the functional and performance requirements for the services.

Component definitions also include verification requirements, and associated test plans, data sets, procedures, harnesses/software development toolkits, and test results.

The granularity of CSA components evolve over time and are based on extensibility and reuse needs, and adopt coupling and cohesion principles so that a component provides a related set of functionality with consistent and stable interfaces.

2.1.2 Modular Open System Approach

The CSA follows a MOSA design, development, and integration strategy. MOSA is an integrated business and technical strategy that employs a modular design and, where appropriate, defines key interfaces using widely supported, consensus-based standards that are published and maintained by a recognized industry standards organization.

Standards can help to ensure that systems do not become locked into a single vendor. Standards should be widely accepted as well as formally recognized. The choice of a standard is predicated not only on the content of the standard but also on its applicability to the CSA component for which it is under consideration.

The CSA is modularized to include common components that will have openly published, non-proprietary interfaces to support extensibility and reuse. Interfaces are standardized and configuration managed. Commercial, open system interface standards are specified for components that are exposed for use by MPAS or other CSA components. De facto standards have been selected when open system standards do not exist *when* the de facto standards are supported by multiple suppliers. Custom interfaces have only been defined where open or de facto standards do not exist.

2.1.3 A Service-Oriented Architecture

Utilizing the broadly adopted SOA paradigm, the CSA exposes services via open, published, non-proprietary interfaces. Service-orientation promotes engineering principles of modularity, encapsulation, information hiding and interface-based design. In a SOA, warfighting operations are defined in terms of activities, which are mapped to services. Service definitions are not dependent on knowing which system(s) implement the services. Externally visible behavior of system components are defined in terms of the services they provide and require.

The benefits of using a service-oriented architecture include:

- **Interoperability**: SOAs enable significant interoperability between software, information, and processes
- **Integration**: Standards-based approaches to SOA enable one service to integrate multiple applications and information from disparate systems
- **Reuse**: More effective reuse of existing applications and systems, reducing (re)development costs
- **Composability**: The loosely-coupled, standards-based approach of SOA allows services to be easily connected and reconnected to create new processes and operational threads
- **Agility**: The loose coupling between operational activities and application functions offers the potential to increase operational agility
- **Risk Reduction**: SOAs allow for the seamless redundancy of critical functionality with varying levels of data fidelity.

In addition, the decoupled nature of the publish/subscribe interaction pattern allows chains of system functions to be executed to perform a service. While MPAS components won't be dynamically composed by operators in the field, software components will be designed such that system engineers and architects can create different applications from these reusable components to provide services in a composable manner.

2.1.4 Data-Centric Publish/Subscribe Communications

The CSA promotes the use of a data-centric publish/subscribe paradigm for the distribution of data and messages between CSA and MPAS components.

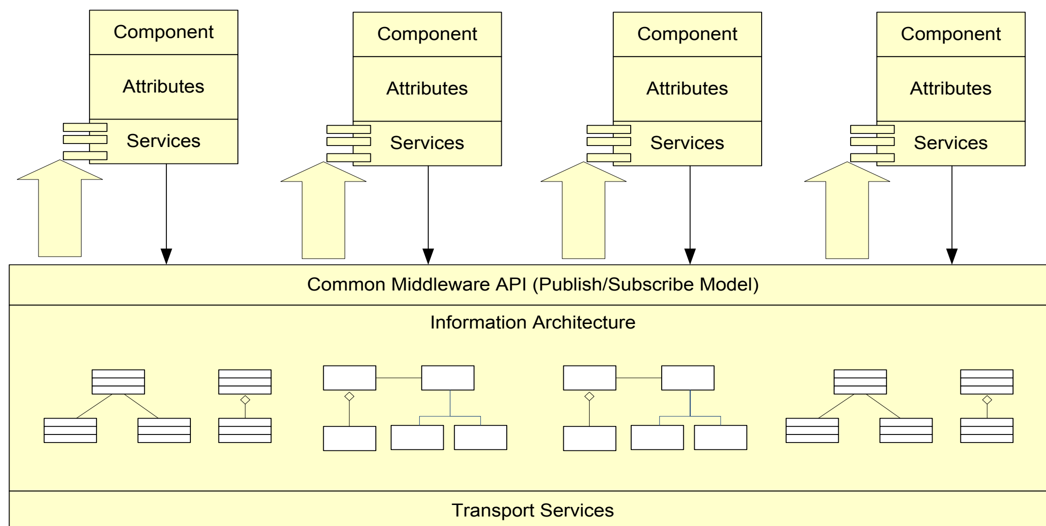

Figure 1. Component Interfaces and Data Flow

An essential part of a component's description is the information it produces and/or the information it consumes. Information (or changes in information) is both a stimulus and a response for event-driven processes. Based on subscriptions requested by consuming components, infrastructure services can optimize the distribution of data to where they are needed and at the necessary rate.

A *data-centric information backbone* has been integrated into the CSA. It supports data subscription by any authorized component; it supports extensibility; it provides a mechanism for timely responses to events; and, it minimizes synchronization points across components. This publish / subscribe model also allows for seamless redundancy of critical functionality with varying levels of data fidelity that could be of interest in risk reduction.

The publish/subscribe model supports the high performance data exchange needs of CSA and MPAS components, especially those components associated with Unmanned Vehicle Control. The model allows components to react immediately to changes in system data produced by other components. This "push" view is a very effective mechanism for supporting the high performance chaining of events within the environment to produce desired effects. But not everything within the environment is event-driven. A "pull" view is also supported by data servers which subscribe to messages and retain the information until requested by another component.

Figure 1 illustrates the information-driven nature of CSA component interfaces. The larger arrows in the figure illustrate data that is 'pushed' to MPAS components based on subscriptions established by those components. The smaller arrows indicate data pushed from those components to be made available to other MPAS components, based on those components' subscriptions. The larger arrows emphasize data flows to the components in order to stimulate event-driven responses to achieve desired effects.

Using publish/subscribe services allows components to not be connected by logical point-to-point interfaces. Publish/subscribe services support many-to-many data exchanges and allow addition of new components with minimized effort.

As illustrated in Figure 2, this information-oriented approach is well suited for evolutionary development of CSA and MPAS components. New components can be developed to publish and subscribe to existing data in accordance with the

established data models, as well as to publish new data and extend the data model where appropriate. Legacy components can be wrapped within a transformation layer that translates between the common system data / interface standard for that component and any internal component representation. The data models also define each component interface in a manner that automated software testing tools can process.

This approach also supports gradual migration from legacy MPAS components to smaller, more reusable components. A single large-grained component could produce a set of system-level data that is allocated to several smaller components in the CSA. As long as the large-grained module exposes all of the messages defined for the set of components it represented, it still conforms to the CSA service specification. Subscribing components would not be aware that one large-grained component was producing those messages instead of multiple smaller components.

The technology employed by CSA was chosen by examining the available message-oriented middleware systems using a weighted and subjective analysis. The criteria and weights for the study (known as the CSA High-Performance Middleware Evaluation) were determined and the results peer-reviewed by the LCS Community. The selected middleware, the Data Distribution Service for Real-Time Systems, is an open standard overseen by the Object Management Group (OMG), an international standards body. DDS will be described in greater detail in section 4.

2.1.5 Model-Driven Development

CSA employs a tailored Model Driven Development (MDD) approach to system definition. It leverages best practices and lessons learned from both PEO IWS Product Line Architecture (PLA) and OSD Unmanned Air System (UAS) Control Segment (UCS).

The CSA Model is captured in UML using conceptual data types and operations, with supporting text descriptions. The UML model provides con-

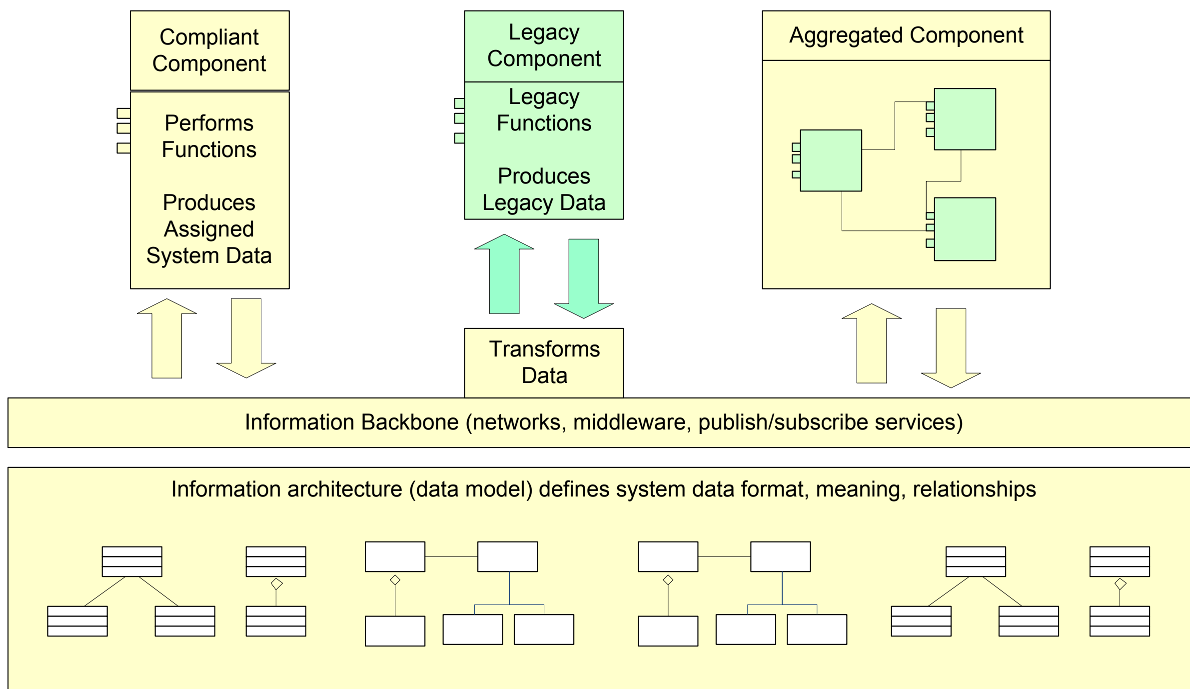

Figure 2. Evolution of Legacy Components with CSA Publish/Subscribe Data Distribution

text for conceptual types using projections against other types in the model. This gives a clearer understanding of any given conceptual type within any given service versus the data used by other services and operations.

A mapping from conceptual types to technology-specific types is also defined in the model. This enables the same service definitions to be mapped against other technologies or hardware platforms in the future, if necessary, without changing the basic definition of the services.

The fully defined CSA Model includes:

- UML depictions of the interfaces exposed by each CSA service
- Data models, as well as interface and message definitions which use these models
- Functional behavior descriptions for service interfaces (what should happen when an interface is stimulated)
- Sequence diagrams to depict interface relationships and expected usage of those interfaces
- Information to support multi-language (Java, C++, IDL, WSDL) code generation for interfaces
- Interface exposure (what the services need to provide)
- Interface usage (what service consumers need to do to use interfaces)
- Verification information to support automated testing of services

Apart from the ability to map against disparate platforms and technologies, the conceptual modeling approach gives a clear visual representation of how the conceptual functionality is mapped into the specifics of any given platform or technology solution. This decouples conceptual aspects of the services from any given implementation of them.

2.2 Benefits over Competing/Past Solutions

The initial analysis identified several areas that could benefit from alternative approaches. While the existing approaches were not entirely without merit, they were limited and vertically focused. For example, the Mission Package Services (MPS) and its Mission Package Operating Environment (MPOE) provided only low-level infrastructure services that were often not fully utilized by the mission packages that it supported. Many features were not flexible enough to truly support the diverse needs of the existing mission packages and stopped short of providing common application services.

Individual mission package features that were similar to features provided by MPS had been used preferentially as the MPS-equivalent services were determined to be insufficient for the vertical focus of the mission modules. The result was duplicative and overlapping features and associated engineering effort. The MPS Alerts services are a prime example of this situation – the MPS-provided solution did not fully meet the needs of the mission packages and has been underutilized.

The MPS also made no attempt to provide application-oriented services or functionality that was common between the various mission packages. The CSA approach was to identify all areas of common functionality that is present in the existing mission packages. These common areas of functionality such as infrastructure services, common business logic services, presentation services, and common operating environment services became the top-level functional domains of CSA.

Moreover, the MPS architecture was not particularly modular, in that it did not provide a flexible or open approach for 3rd parties to easily modify or replace its components as they were tightly coupled. The MPS fell short in providing a truly component-based modular architecture which minimized the value of fully integrating its features with the mission modules.

The CSA approach to service definition has resulted in a very modular design with clearly-defined and granular components. The functionality provided by each CSA domain is defined by high-level service descriptions. These high-level de-

scriptions are further refined into individual service definitions which are defined by use cases, requirements, UML models and specific service definitions which follow the modular open approach defined by CSA. These artifacts are reviewed extensively by stakeholders and MPAS developers.

Integration with the features provided by MPS was difficult because MPS was primarily based on libraries rather than discrete services. The MPS libraries use CORBA which is an aging object-centric communications middleware approach that has well-known limitations and vendor-specific integration issues. That approach to providing common software facilities does not support the realization of a service-oriented and modular architecture.

CSA employs a standards-based data-centric publish/subscribe mechanism using DDS for interfacing with its services. This results in discrete services which are loosely coupled and easily replaced or upgraded by 3rd parties as needed. CSA services can be upgraded or replaced independently from the MPAS that make use of the services.

2.3 DDS-Related Aspects

As stated above, CSA uses message-oriented middleware that is based on the OMG DDS standard. Unlike CORBA or message-exchange solutions, commercial implementations of the DDS standard are guaranteed to be interoperable down to the network protocol level. CSA uses only standard DDS features of the middleware to specifically avoid vendor-lock. Service providers or MPAS developers are free to use any conformant implementation of DDS.

Using standards-based middleware supports the open business model aspect of the LCS program, but more importantly, the features of DDS support a data-centric publish-subscribe architecture that is loosely coupled. Data-centric architectures provide an environment that is easily extensible and does not result in the same degree of integration complexity that is inherent with other message-centric approaches.

DDS provides a rich set of over 20 Quality of Service (QoS) policies that specifically support low-latency, high-bandwidth and high-volume data exchange. CSA takes advantage of the high-performance aspects of DDS to meet or exceed its requirements for performance. CSA also makes extensive use of the QoS policies that make use of middleware-provided connectivity management features which have traditionally been implemented using various ad-hoc approaches.

2.4 Lessons Learned

In the course of the development of CSA, several lessons have been learned. Everything from process improvement, to specific details of "look and feel" of presentation components, to better approaches to data-centric design have been identified. The particular lessons that support modular development and an open business model will be briefly discussed.

In order to minimize confusion and inconsistencies, and to ensure a consistent understanding of service functionality, it is necessary to document use cases prior to service modeling or System Requirements Specification (SRS) requirement refinements. The use cases should provide a plain English description of each service's functionality and identify key decisions affecting service behavior as well as outstanding issues and questions.

To provide consistent approaches to software development and unit testing, the CSA Software Development Plan (SDP) was expanded to specify software quality guidelines based on the associated SRS requirements. These requirements are often driven by Security Technical Implementation Guides (STIGs) and program office directives for safety. The approaches defined by the SDP must be applied consistently amongst all developers and organizations that contribute to CSA services.

Service interfaces must make use of concisely defined data structures and types. Data must not be

unnecessarily duplicated within messages when it can be provided using discrete topics and instances. Message-centric approaches typically result in duplication of data that is only valid during any given service interaction. A data-centric approach makes use of well-defined data states and relationships between that data. CSA services make use of data-centric architecture concepts to the greatest extent possible.

The operations provided by CSA services should avoid traditional message-centric request/response patterns whenever possible. Request/response interactions are best suited for message-oriented technologies that use point-to-point connections. Publish/subscribe interactions are typically one-to-many and many-to-one and are not characterized by point-to-point relationships between participants.

Data-centric services typically do not need to know which participants are requesting operations, and any data generated in response should reflect the new state of the system rather than provide a traditional success/failure message. CSA services make use of data-centric interaction patterns to the greatest extent possible and avoid request/response interactions.

Although DDS implementations are based on the OMG standard, they are not necessarily limited to the specification and often provide vendor-specific additions as well. Participants that make use of CSA services must agree with QoS policies specified on topics that are defined by the service. It is necessary to avoid vendor-specific functionality whenever possible and to provide a means of sharing service-provided QoS policy settings.

The CSA provides an API known as the DDS Encapsulation Layer (DDS-EL) that enforces a consistent use of DDS and provides a means of sharing QoS policy settings in a standard way, and providing the most valuable features of DDS in a simplified manner. While use of the DDS-EL is not currently mandatory, all existing services have been written using it. Its use is highly recommended by any group implementing services or consumers of the services in order to decrease development effort and to reduce integration issues.

3 Overview of OMG DDS

DDS is composed of two primary but distinct specifications – one for the application layer interfaces, and another that assures wire-level interoperability between vendor implementations. These layers ensure not only that a different vendor's DDS implementation[ii] can be swapped in without impacting application code, but that systems built using different implementations of DDS will interoperate.

DDS has been embraced by numerous DoD programs, and Prime contractors working on Open System Architecture initiatives. Users include virtually all major US Prime contractors, US defense research laboratories, and many commercial telecommunications, transportation, utilities, automotive, manufacturing, mining, and financial companies. With encouragement from the Navy, the first version of the OMG DDS specification was adopted in 2004.

In the previous sections, we discussed some aspects of this standard in the context of the work on CSA. Here we provide a general overview of DDS, and highlight more of the features that can be leveraged to support open system architectures.

3.1 General Design

At its core, DDS is data-centric and publish-subscribe. It is these two central architectural features that differentiate it from all other distributed system design approaches, and make it inherently ideal for the implementation of open systems. Moreover, the ramifications of adopting these features cascade deep into the design of your system, and result in significant long-term benefits includ-

[ii] Currently, the OMG website lists 11 implementations of the DDS specification (http://portals.omg.org/dds/category/web-links/vendors)

ing lowering costs, and improving evolvability, interoperability, scalability, and more.

DDS Systems are data-centric. At a very basic level, in order to build a distributed system, the various applications and processes need to communicate. The prevalent distributed system design methodology is to focus on the methods or procedures and component interaction (e.g., object-oriented programming). It is a natural extension of the way standalone applications have been designed for decades, and it works quite well. However, distributed systems using this interaction paradigm tend to be brittle, more difficult to evolve or extend, and they can be nearly impossible to evolve by anyone but the original developers. These systems require potentially significant code changes when new components are introduced in the distributed program logic. This will result in significant maintenance and upgrade costs for the customer.

A data-centric or data-oriented methodology, on the other hand, focuses on the flow of data through the system. Data-centricity elevates data to "first class citizenship" within the system, and not simply the byproduct of a remote method invocation of other paradigms: *the data is relevant independent of the producer or consumer*. The producer and consumer are no longer inextricably tied together. Conversely, method-centric approaches tend to seed brittle, difficult to scale solutions.

Data-centricity is the first step toward building open system architectures. However, the shift in paradigms from object-centric to data-centric can be challenging for system architects and programmers to adjust to, similar to the introduction of object-oriented thinking to the procedural programmers of the 1990's.

A comparison between object-oriented and data-oriented programming is shown in Figure 3.

Object-oriented programming	Data-oriented programming
Hide the data (encapsulation)	Expose the data
Expose methods – code	Hide the code
Intermix data and code	Separate data and code
Define object interfaces	Agree on data labels and schemas
Invoke/Implement operations on objects	Send/Receive messages
Combined processing, no restrictions	Strict separation of parser, validator, transformer, and logic
Changes: Read and change code	Changes: Change declarative data definition
Tightly coupled	Loosely coupled

Figure 3. A comparison of data-oriented programming with object-oriented programming. The data-oriented approach enforces attention on the data rather than on the processes that manipulate the data.

DDS Systems are loosely coupled. Rather than sending information from one specific process to another, DDS uses the notion of a data bus. Applications *publish* data to the bus, and other applications interested in the data receive it. So, applications are not tightly bound to each other, resulting in systems that are no longer brittle.

In DDS terms, processes that produce data are called *publishers*, and those that the consume data are called *subscribers*. Subscribers declare their interest in the data by using a well-known *Topic* name, which is mapped to a well-known data structure. An application can have any number of publishers, subscribers, and topics. Overly simplified, one can think of the data bus as numerous telephone party lines, one for each topic, that applications can talk on and listen to. As subscribers

and publishers come online, they announce their interest in specific topics. This enables the subscribers and publishers to discover each other, and to negotiate delivery agreements based upon numerous quality of service parameters. These parameters are needed to ensure that a publisher can meet the delivery requirements of the subscriber. For example, some subscribers may only want to receive data from publishers that can emit data at 60 samples per second, or that provide reliable delivery of the data (versus best effort).

DDS also provides the concept of *domains* that enable an enforceable partitioning of the data bus, limiting a publisher or subscriber to only participate in one domain. This allows the data, for example, to flow in different domains in order to separate concerns, or to limit data access.

This decoupling of applications from each other has significant benefits. For example, it facilitates scalability as it allows any number of new data producers and consumers to be added, *without* coding changes or bringing the system down. Since only data interactions are specified (rather than interfaces), devices or processes can be upgraded or added without the need to change code and exhaustively retest every configuration. It also enables DDS to provide fault tolerance, where if one or more processes fail (publisher or subscriber), DDS can detect this and allow a replacement process to immediately take over for it.

It is worth noting that DDS also supports other interaction paradigms such as request-reply, for situations when its required. Being built upon DDS, they inherit all of the benefits of data-centric publish-subscribe.

DDS systems are data-aware. For open systems built using DDS, the structure of the data being sent over the network is understood by the middleware, and it is not simply treated as a “blob” of data that needs to be moved from point A to point B. In DDS, the structure of the data is declared using an interface definition language (IDL). This IDL is shared, along with the quality of service parameters discussed above, during a discovery phase (when a new publisher or subscriber joins the system) to match publishers and subscribers. This *data (or structure) awareness* enables DDS to reason about the data, optimize the management of it, and offer data services to the applications.

A core feature that DDS provides is a logical global data space across the distributed system, where all processes have the same shared view of the data they subscribe to. DDS will automatically manage and update the state of data instances – such as the value of a stock, or the position of a sensor – as the data arrives, locally managing this complexity for each process. A process can ask to be alerted when any data arrives, or only when specific changes in the data occurs. A useful analogy is to think of DDS as managing data in motion, while a database manages data at rest.

Moreover, since DDS is data-aware, an application can tell DDS what data within a topic it is interested in by using SQL data filters. Only data that satisfies the filters are forwarded to the application. For example, an application may only want data for a specific targeting region, or for sensor state that changes in value by more than 5%. This filtering is generally done on the receiving (subscriber) side, however some advanced implementations of DDS may also forward a filter to the publisher as an optimization in order to reduce network traffic (e.g., if all existing subscribers are only interested in satellite data over Maine, then the publishers can stop publishing imagery for the rest of the US).

Since DDS knows the structure of all of the data[iii], it can reason about the data. This allows the DDS middleware to build in common data-oriented features that applications can reuse, thereby signifi-

---

[iii] Application designers can implement data structures as binary sequences that DDS cannot filter on, in conflict with the open system tenets. However, this is necessary for some types of data such as raw sensor data.

cantly reducing the complexity, size, development and lifecycle maintenance costs of the application code (see Figure 4). Feature-limited middleware may tout their compact size, but the trade-off is that the application developers will be burdened with correctly designing, implementing, testing and maintaining these needed features in their own code. Programs may spend millions, or tens of millions, trying to get distributed system communication right, but often fail. Leveraging proven implementations of DDS significantly reduce these costs.

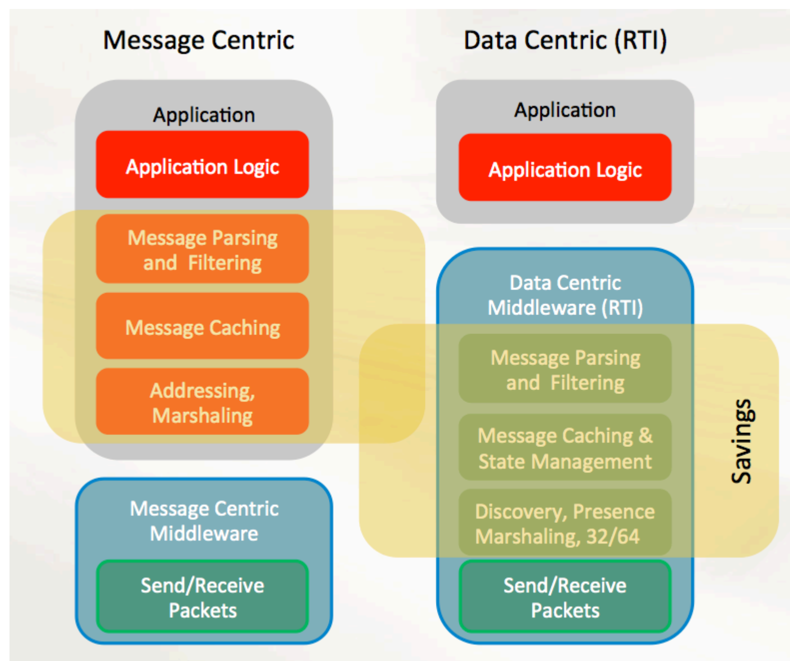

Figure 4. DDS reduces development and lifecycle costs.

Another advantage of exposing the structure of the data to DDS is that the middleware can provide language & platform independence, and even protocol independence. DDS naturally maps the data into a language neutral form that can be ingested by multiple programming languages running on any number of operating systems and hardware platforms.

3.2 DDS Security

RTI recently completed the creation of an OMG-supported security specification for DDS[6], funded in large part by a Navy IWS Phase II SBIR. RTI's implementation of this standard has been tested at the Quantico Cybersecurity Range. The DDS Security specification was adopted by OMG in December. A detailed description can be found in our companion IIS 2015 paper entitled "Next-generation Cybersecurity for Advanced Real-time Distributed Systems".

4 Conclusions / Summary

Implementations of the OMG DDS specification simplify application and integration logic by offering more capability than traditional messaging solutions. Instead of exchanging messages, software components communicate by sharing first-class data objects. Applications operate directly on these objects (create, read, update and delete). Developers do not have to deal with low-level messaging or networking interfaces, which in turn, significantly reduces application development time and cost.

DDS handles the details of data distribution and management, including serialization and lifecycle

management. It also provides for data in motion what a database provides for data at rest:

- Decoupling. Data producers are agnostic to the number of consumers and the type of processing they do. This allows components to be added and changed without affecting those that are already deployed.
- Easy integration. The interfaces in a system—as defined by the data model—are explicit and discoverable. Integration requires no knowledge of a component's implementation and you do not need to reverse engineer protocols and messages.
- Robustness. DDS maintains a system's shared state, providing a single source of truth. Late and re-joining applications automatically synchronize with the current state. This ensures applications have a consistent world view even in dynamic and large-scale environments.

Progeny has taken advantage of these features for the design and implementation of LCS CSA, leading to a system that is easier to scale, evolve, and maintain.

5 Acronyms

ADD Architecture Design Description
ASW Anti-Submarine Warfare

C4I Command And Control, Computers, And Intelligence
CNO Chief of Naval Operations
CORBA Common Object Request Broker Architecture
COTS Commercial-Off-The-Shelf
CSA Common Software Architecture

DDS Data Distribution Service
DDS-EL DDS Encapsulation Layer
DPG Defense Planning Guidance

FACE Future Airborne Capability Environment
IDL Interface Definition Language

LCS Littoral Combat Ship

MCM Mine Countermeasures
MDD Model-Driven Design
MPAS Mission Package Application Software
MPOE Mission Package Operating Environment
MPS Mission Package Services
MOSA Modular Open System Approach

OMG Object Management Group
OSA Open Systems Architecture
PEO Program Executive Office
PLA Product Line Architecture

QDR Quadrennial Defense Review
QoS Quality of Service

SDP Software Development Plan
SOA Service Oriented Architecture
SRS System Requirements Specification
STIGs Security Technical Implementation Guides
SUW Surface Warfare

UAS Unmanned Air System
UCS UAS Control Segment
UML Unified Modeling Language

6 References

1. Payloads over Platforms: Charting a New Course" Proceedings Magazine, July 2012 Vol 138/7/1,313 http://www.usni.org/magazines/proceedings/2012-07/payloads-over-platforms-charting-new-course

2. Data Distribution Service (DDS), http://portals.omg.org/dds/

3. Better Buying Power, http://bbp.dau.mil/

4. Towards Affordable DoD Combat Systems in the Age of Sequestration, http://blog.sei.cmu.edu/post.cfm/ towards-affordable-dod-combat-systems-in-the-age-of-sequestration

5. Open Systems Architecture (OSA) Brochure, https://acc.dau.mil/adl/en-US/695451/file/75899/OSABrochure.pdf

6. DDS Security Specification, http://www.omg.org/spec/DDS-SECURITY/