Downloaded from https://www.everyspec.com

# Open Architecture (OA) Computing Environment Design Guidance

## Version 1.0

### 23 August 2004

**Prepared for:**

**PROGRAM EXECUTIVE OFFICE, INTEGRATED WARFARE SYSTEMS**
**Program Manager, Open Architecture (OA)**
**1333 Isaac Hull Avenue SE**
**Washington Navy Yard, DC 20376-2301**

**Prepared by:**

**Naval Surface Warfare Center Dahlgren Division (NSWCDD)**
**17320 Dahlgren Road**
**Dahlgren, VA 22448-5100**

**"Statement A: Document approved for Public Release: Distribution is unlimited."**

Downloaded from https://www.everyspec.com

OACE Design Guide 1.0
23 August 2004

# Open Architecture Program: An Enterprise Approach to Introducing Open Architecture Into Navy Warfare Systems…and Beyond

**"Statement A: Document approved for Public Release: Distribution is unlimited."**

Downloaded from https://www.everyspec.com



# FOREWORD

The Assistant Secretary of the Navy for Research, Development and Acquisition (ASN (RDA)) assigned the Program Executive Office for Integrated Warfare Systems (PEO IWS) with responsibility for coordinating the introduction of open architectures into the Navy's warfare systems. The Open Architecture (OA) initiative is a multi-faceted strategy providing a framework for developing Joint interoperable systems that adapt and exploit open-system design principles and architectures.

The strategy calls, in part, for establishment of OA Computing Environments (OACE) through the desemination of guidance and standards that descibe types of computing systems that will exploit open system design principles. The initial implementation guidance for OA addresses a family of hardware and software standards as well as guidance on developing software based warfighting functions that will perform well in an extensible architecture. The extensible characteristics of these types of open architectures will permit the use and implementation of a wide variety of products, including reusable software components, requirements analysis, contract language, process descriptions, test cases and scenarios, models, simulations, designs and architectures and human expertise across Naval Air, Land, and Undersea platforms. This initial implementation reflects a shift in focus from a platform-centered warfare system development approach to a more integrated, Battle Force (BF)-centered approach.

The introduction of open architectures into Naval Warfare Systems will be formally instituted through the Open Architecture Enterprise Team (OAET) established by ASN RDA in August of 2004. However, this OA documentation is intended for use in the surface ship combat systems domain and has been released by PEO IWS. In the future, all documentation related to the OA initiative will be released by the OAET. The OACE Technologies and Standards document provides a core set of technologies and standards that apply to the OACE technology base. The OACE Design Guidance document (this document) provides guidance concerning design aspects of the standards-based computing environment that is to be used in OA warfighting systems.

.

Downloaded from https://www.everyspec.com



## EXECUTIVE SUMMARY

Computing technology is a key part of the OA initiative. A unified standards-based set of computing resources that is to be used in OA warfighting systems is called the Open Architecture Computing Environment (OACE). This document, *Open Architecture Computing Environment Design Guidance,* provides guidance concerning OACE technologies and application design using those technologies. A companion document*, Open Architecture Computing Environment Technologies and Standards,* provides an enumeration of the standards and product selection criteria that apply to the OACE technology base.

This document is intended to provide overall guidance for the design and implementation of warfighting-capable software that, when coupled with OACE, will meet mission requirements for Naval warfighting systems. Initial review indicates that the design guidance as written has applicability in both the warfighting system domain and the Command, Control, Communication, Computers, and Intelligence (C4I) domains. Therefore, it is anticipated that parts of the guidance are extensible to selected C4I systems, the specifics of which are left to the system developers and government program offices.

This document contains two major technical guidance sections. The first, Section 3.2.1, Component Design Guidance, provides guidelines useful for constructing real-time distributed warfighting application computer programs. The second, Section 3.2.2, Open Architecture Computing Environment, provides a description of the technology base to be employed in conjunction with OA warfighting systems.

Downloaded from https://www.everyspec.com



**TABLE OF CONTENTS**

**Section** **Page**

1 INTRODUCTION ... 1-1
1.1 Purpose ... 1-1
1.2 Scope ... 1-1
1.3 Technical Approach ... 1-2
1.3.1 Open-System Characteristics ... 1-2
1.3.2 Open Architectures ... 1-4
1.3.3 Benefits of Open Architecture ... 1-6
1.3.4 Managing Change with Standards and Middleware ... 1-8
1.3.5 Non-Standard Products ... 1-11
1.3.6 OACE Change Management ... 1-11
1.4 Document Overview ... 1-12

2 APPLICABLE DOCUMENTS ... 2-1

3 TECHNICAL ARCHITECTURE GUIDANCE ... 3-1
3.1 Technical Overview ... 3-1
3.1.1 Functional Partitioning ... 3-2
3.1.2 Computing Characteristics ... 3-3
3.1.3 Federated vs. Integrated Architectures ... 3-4
3.1.3.1 Federated Approach ... 3-4
3.1.3.2 Integrated Approach ... 3-4
3.1.3.3 OA Guidelines ... 3-5
3.1.4 Implementation Constraints ... 3-5
3.1.5 OACE Compliance Categories ... 3-6
3.1.6 Development Processes and Tools ... 3-7
3.2 Technical Architecture Guidance ... 3-9
3.2.1 Component Design Guidance ... 3-10
3.2.1.1 Architecture and Design Patterns ... 3-12
3.2.1.2 Component Design ... 3-14
3.2.1.3 Portability ... 3-22
3.2.1.4 Location Transparency ... 3-25
3.2.1.5 Client-Server ... 3-26
3.2.1.6 Data Distribution ... 3-31
3.2.1.7 State Data Coherency ... 3-34
3.2.1.8 Computational Flow ... 3-36
3.2.1.9 Fault Tolerance ... 3-37
3.2.1.10 Scalability ... 3-43
3.2.1.11 Real-time Performance ... 3-49
3.2.1.12 Process, Thread, and Memory Management ... 3-55
3.2.1.13 Data Brokers ... 3-57

Downloaded from https://www.everyspec.com



# TABLE OF CONTENTS

**Section** **Page**

3.2.2 Open Architecture Computing Environment (OACE) ........................ 3-58
3.2.2.1 Cabling and Cabinets ........................ 3-63
3.2.2.2 Information Transfer ........................ 3-64
3.2.2.3 Computing Resources ........................ 3-66
3.2.2.4 Peripherals ........................ 3-70
3.2.2.5 Operating Systems ........................ 3-70
3.2.2.6 Adaptation Middleware ........................ 3-71
3.2.2.7 Distribution Middleware ........................ 3-72
3.2.2.8 Design Patterns, Frameworks, and Wrappers ........................ 3-77
3.2.2.9 Resource Management (RM) ........................ 3-79
3.2.2.10 Instrumentation, Recording, and Assessment ........................ 3-83
3.2.2.11 Failure Management ........................ 3-85
3.2.2.12 Information Assurance ........................ 3-86
3.2.2.13 Time Service ........................ 3-87
3.2.2.14 Programming Language Facilities ........................ 3-88
3.2.3 System Composition ........................ 3-89
3.2.4 Displays ........................ 3-89
3.2.4.1 Description ........................ 3-89
3.2.4.2 Guidance ........................ 3-90
3.2.5 System Test and Certification ........................ 3-90
3.2.5.1 Description ........................ 3-90
3.2.5.2 Guidance ........................ 3-91
3.2.6 Selection of Standards ........................ 3-92
3.2.6.1 Applicable Standards ........................ 3-93
3.2.6.2 Open-Source Products ........................ 3-94

**Appendix**

A ACRONYMS ........................ 1

Downloaded from https://www.everyspec.com



## ILLUSTRATIONS

**Figure** **Page**

Figure 1-1. Open-System Model ... 1-3
Figure 1-2. Managing Change with Middleware ... 1-10
Figure 3-1. Functional Partitioning ... 3-2
Figure 3-2. Client-Server Model ... 3-29
Figure 3-3. Fault Tolerance Models ... 3-39
Figure 3-4. Partitioning Scalability Patterns ... 3-45
Figure 3-5. Load Sharing by Replication ... 3-47
Figure 3-6. Comparison of Design Models ... 3-53
Figure 3-7. Open Architecture Computing Environment ... 3-60
Figure 3-8. Notional Physical Architecture ... 3-61
Figure 3-9. Resource Management Architecture ... 3-80

## TABLES

**Table** **Page**

Table 1-1. Open Architecture Concepts and Benefits ... 1-7
Table 1-2. Application Portability Strategies ... 1-9
Table 3-1. OA Constraints ... 3-6
Table 3-2. Technical Architecture Guidance Index ... 3-9
Table 3-3. Units of Computing Performance Parameters ... 3-69
Table 3-4. Types of Standards ... 3-93

Downloaded from https://www.everyspec.com



# SECTION 1

# INTRODUCTION

Computing technology is a key part of the Open Architecture (OA) effort. The open standards-based computing technologies specified as part of the Open Architecture Computing Environment (OACE) can be assembled into a system in a variety of ways. Care must be taken, however, in how this is done. Otherwise, the OA goal to achieve a common computing base might not be reached. To achieve this goal, guidance is needed on common ways to implement systems using the standards and technologies specified for the OACE.

## 1.1 PURPOSE

The purpose of this guidance document is to provide technical principles and design guidelines for redesigning and reimplementing Naval Warfare Systems (NWSs) in architectures suitable for meeting the performance, long-term maintenance, and upgrade goals described above. The overall set of computing resources used in OA systems is called the OACE. The guidance contained herein describes overall computing system architecture, system-wide design principles, computing equipment and support software infrastructure technologies, standards, application functional partitioning principles, computer program characteristics, and development strategy. The intent of the guidance is to characterize technologies and design techniques that have proven successful in other systems and prototypes comparable in complexity and/or stressing requirements to NWSs, e.g., commercial, business, financial, medical, telecommunications, and process control applications. The guidance also focuses attention of the technology base needed to sustain performance for NWS applications.

No initial change in NWS functional capabilities are anticipated that would impact system requirement specifications such as the System Specification (A-Specification) and Prime Item Development Specifications (PIDS) as a result of the guidance contained in this document. Rather, this document is intended to foster development of a robust and enduring computer program design and equipment utilization approaches.

## 1.2 SCOPE

The technical guidance contained herein and the standards invoked in the companion volume, *Open Architecture Computing Environment Technologies and Standards,* apply to a variety of systems, primarily but not limited to Program Executive Office for Integrated Warfare Systems (PEO IWS) warfighting systems. PEO IWS systems are components of larger "systems of systems." For this reason, the OA standards may be applied to non-PEO IWS systems as well. Such scope is delineated in conjunction with the promulgation of this document and its companion, *Open Architecture Computing Environment Technologies and Standards*. The OA effort is an enterprise-wide undertaking that, with maturity, will govern all Naval warfare systems from development through the life cycle.

Downloaded from https://www.everyspec.com



The technical guidance contained in this document is presented for descriptive and informational purposes and is not to be construed as contractually binding unless so designated, in whole or in part, by separate mechanism. Such mechanisms may include PEO or program office directives and instructions, clauses in Requests for Proposal (RFPs), requirements statements in production contracts, etc. Specific compliance statements designed for formal invocation are contained in the Standards document mentioned above.

## 1.3 TECHNICAL APPROACH

Design guidance is provided to supplement the standards and technologies provided in the *OACE Technologies and Standards* document. It is intended to provide common ways to apply these standards and technologies to meet NWS requirements. Guidance is provided for an overall computing system architecture, system-wide design principles, computing equipment and support software infrastructure technologies, standards, application functional partitioning principles, computer program characteristics, and development strategy.

### 1.3.1 Open-System Characteristics

Open systems and architectures built to open-system principles possess a number of common characteristics. While not every open system possesses every possible characteristic, most open systems tend to possess most of these characteristics. Based on examination of various open-system definitions, the attributes of an open system include the following:

a. Use of public, consensus-based standards

b. Adoption of standard interfaces

c. Adoption of standard services (defined functions)

d. Use of product types supported by multiple vendors

e. Selection of stable vendor with broad customer base and large market share

f. Interoperability with minimal integration

g. Ease of scalability and upgradability

h. Portability of application(s)

i. Portability of users

A number of the points need amplification. First, while upgradability is a well-recognized characteristic of open-system designs, scalability may not be so widely accepted. Scalability has a number of possible meanings. In this case, we are talking about the ability to add new functionality and resources. Second, open specifications are those for which a

Downloaded from https://www.everyspec.com



consensus-based standards organization provides control and adjudication over content and evolution. Third, the unusual attribute, ***portability of users***, denotes the benefit whereby users can transition easily to a particular new system given experience with other similar systems.

Finally, the invocation of widely used public standards must be qualified. There are many standards from which to choose. The choice of a standard is predicated not only on the content of the standard but also on its applicability to the domain for which it is under consideration. Practically, this means that standards must fit their application domain. For Navy warfighting systems that have a substantial requirement for mission critical real-time performance, standards must be chosen from standards families that contain a real-time aspect. This does not mean that all aspects of the chosen standard are real-time nor that real-time performance is necessarily invoked for all applications using the standard. But, it does mean that the real-time requirement is a necessary condition for OA standards, particularly in the operating system and middleware areas. This philosophy is reflected in the *Open Architecture Computing Environment Technologies and Standards* document, a companion volume to this document.

The relationships among and characteristics of key components in open computing systems are shown below. In the diagram, two computer-based devices are shown. Each has a hardware interface with which to communicate with other devices via an information transfer mechanism across a media, e.g., copper, optical fiber, wireless, etc. In this case, a request-response or "pull" situation, an invoking segment in one device, typically an application computer program, requests information from a providing segment in another device via a software Application Programmer Interface (API). In this representation, both the software API and the hardware interface are based on publicly accepted standards. The formats and protocols are defined by the standard, as is the expected response of the providing segment. The possible use of multicast is optional but is sometimes useful, especially for situations where data is to be "pushed" to multiple receivers simultaneously.

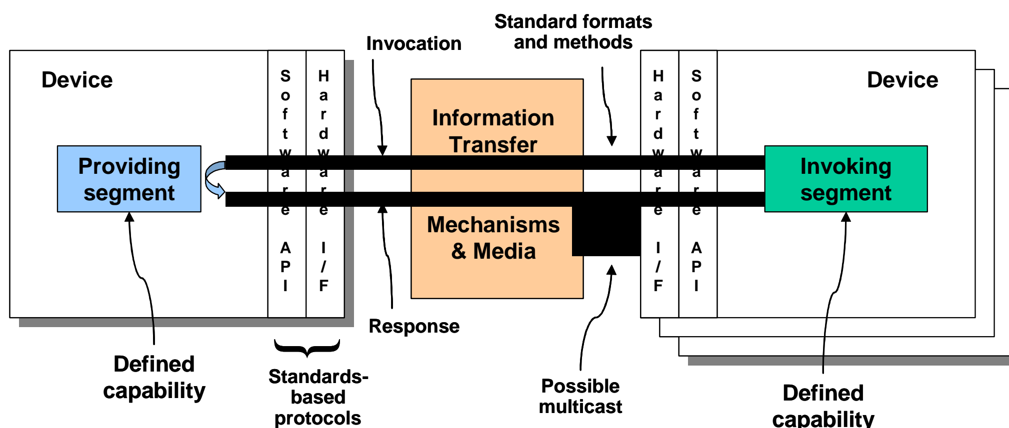


**Figure 1-1. Open-System Model**

Downloaded from https://www.everyspec.com



Standards play an essential role in open systems. They are the mechanisms that ensure that systems do not become locked into a single vendor. Note that for this principle to be effective, the standard should be widely accepted as well as formally recognized. The result of incorporating open-system principles into system architecture is that such systems fulfill the following relationship:

Open Systems = Standard Interfaces + Defined Capability → Interoperability + Ease of Change

Finally, in addition to upgrading engineering practices to design NWSs according to open-system principles, it is necessary to define suitable metrics that provide quantitative measures of the productivity and effectiveness of the open-systems approach. For example, standards compliance may be approximated by first-order estimation of the ratio of standards-based APIs to total APIs. Time and effort needed to incorporate new functions may be tracked and compared to historical data about similar activities. Numbers of interfaces, especially multi-client server interfaces, may be compared to previous implementations.

### 1.3.2 Open Architectures

Section 1.3.1 introduced a number of attributes of commercial open systems. While helpful as organizing principles, these attributes do not, by themselves, constitute an architecture. Architectures are specific, albeit high-level designs that satisfy not only a system's functional and performance requirements but also any relevant non-functional goals, such as open-systems principles. The choice of architecture and associated principles and patterns often determines how well or even if nonfunctional goals can be met. The following definition from the Open Systems Joint Task Force (OSJTF) is adopted:

> **Open Architecture:** "*An architecture that employs open standards for key interfaces within a system*."
>
> — DoD OSJTF

For the OA program, open architecture is the high-level technical structure of the NWS as designed in accordance with the principles of open systems to achieve both mission requirements (e.g., real-time performance) and life-cycle supportability goals. (Note that this definition does not address the concept of functional commonality, a key additional cornerstone of the OA effort.) Any NWS architecture, whether open or otherwise, should first and foremost meet the requirements of the warfighting system. Fortunately, the state of computing technology has advanced to the point where satisfying both is now possible. Key NWS domain attributes and performance requirements include:

a. Driven by external world events

b. Wide, dynamic range of processing loads

Downloaded from https://www.everyspec.com



c. High-volume throughput of continuously refreshed data

d. Low latency responsiveness to asynchronous events

e. Hard real-time deadlines associated with periodics

f. Computational paths traversing multiple components

g. Soft real-time processing requirements

h. High availability and survivability requirements

i. Operator display and control requirements

j. Information assurance requirements

k. Safety and certification requirements

For mission-critical systems, real-time capability is often a dominant requirement. Computing characteristics, including hard real-time and soft real-time viewpoints, are also of interest to OA. Other technical characteristics of OA include:

a. Distribution of processing and the separation of presentation, application logic, and data in order to minimize the coupling between them

b. Widespread use of standards-based Commercial Off-the-Shelf (COTS) computing technologies

c. Functional capabilities implemented as partitioned and functionally pure components

d. Use of object-oriented (OO) programming and/or structured programming constructs within components and middleware technologies for interconnection of and interoperation among components

e. Use of design mechanisms such as client-server to maximize isolation of implementation details from publicly visible services and APIs

f. Portability and transparency of application components with respect to physical location and network, processor and operating system types, etc.

g. The appropriate use of architecture and design patterns and frameworks to facilitate repeatable high-quality implementations

h. Use of appropriate development tools during the design, implementation, and test phases of the project

Downloaded from https://www.everyspec.com



The corresponding goals of the OACE are to provide to the NWS the benefits not only of assured technical performance but also of reduced life-cycle cost, affordable technology refresh, and reduced upgrade cycle time. Expected benefits include:

a. Scalable, load invariant performance

b. Enhanced information access and interoperability

c. Enhanced system flexibility for accomplishment of mission and operational objectives

d. Enhanced survivability and availability

e. Reduced life-cycle cost and affordable COTS technology refresh

f. Reduced cycle time for changes and upgrades

An additional point should be noted regarding survivability. Although current systems meet availability requirements, the use of shared Resource Management (RM) offers additional survivability benefits over those already present. Just as important to virtually all combat systems-related computing is that the shared computing resources approach will not require additional dedicated hardware for each increment of added functionality. In an OA computing system, computing is a basic commodity that can be effectively managed to meet dynamically changing computing needs.

### 1.3.3 Benefits of Open Architecture

Much of the information contained herein is the outgrowth of more than a decade of shared technology and architecture evaluation and risk mitigation by the Navy, the Defense Advanced Research Projects Agency (DARPA), academia, and industry. Research and Development (R&D) efforts have focused on certain architectural concepts intended to foster life-cycle cost benefits, as well as technical performance benefits. These concepts include:

a. Open architectures

b. Distributed processing

c. Portability

d. Scalability

e. Modularity

f. Fault tolerance

g. Shared resource management

Downloaded from https://www.everyspec.com



h. Self-instrumentation

These concepts and their benefits are summarized below in Table 1-1.

**Table 1-1. Open Architecture Concepts and Benefits**

| BENEFIT/CONCEPT | LOAD INVARIANT PERFORMANCE | INFORMATION ACCESS | MISSION FLEXIBILITY | AVAILABILITY | REDUCED COST | RAPID UPGRADE |
|---|---|---|---|---|---|---|
| Open Architecture | | X | | | X | X |
| Distributed Processing | X | X | X | X | | X |
| Portability | | | | | X | X |
| Scalability | X | | X | | X | X |
| Modularity | | | | | X | X |
| Fault Tolerance | X | | | X | | |
| Shared Resource Management | X | | X | X | | |
| Self-Instrumentation | X | | X | X | | |

Open architectures provide benefits with respect to lowered development and ownership cost, rapid upgrades, and increased information accessibility. Distributed processing, as well as the ability to apply it in a scalable manner, provides the computing power needed to deliver required mission performance. Portability, one of the key aspects of open systems, reduces costs by making upgrades and technology refreshes more nearly independent of the underlying computing equipment and support software. As with any military system, fault tolerance is a vital component of system availability and survivability.

Most of the architectural characteristics listed above have sufficiently widespread antecedents to be obvious in their meanings. One possible exception is RM, the collection of services and capabilities needed to manage the computing system from a resource consumption perspective. This capability, along with the instrumentation needed to diagnose system health and to assess corrective strategies, contributes to increased performance, mission flexibility, and availability. In effect, it constitutes a closed-loop control system that dynamically adjusts application use of computing resources to conform to system performance and availability objectives in response to changing tactical stimuli and mission priorities.

Although not explicitly addressed in the discussion above, software reuse merits strong consideration during definition of the OA plan. Reusability may be achieved by a variety of methods. One method is by creation of reusable libraries of application code segments designed for use in multiple larger structures. A contrasting approach is reuse at the application program

Downloaded from https://www.everyspec.com



level. OO technology (where real-time efficiency requirements permit) provides a capable framework for such reuse, although OO techniques are by no means mandatory for the achievement of reuse goals. While reuse of low-level code segments has been employed successfully on occasion, it generally requires a breadth of management control that is typically difficult to sustain. Reuse of entire programs is thus advised except in special cases.

### 1.3.4 Managing Change with Standards and Middleware

Application functions change paced by threats and missions. Computing technology also evolves driven by technological innovation and market pressures. Even standards change, albeit at a slower pace. Unless applications are in some manner isolated from this change, system upgrades and COTS refreshes will be expensive and time-consuming. Managing this change and isolating its effects is the key to successful OA implementation. A number of strategies are available for ensuring application portability with a minimum of change, see Table 1-2. Although all of these techniques may be used at times, the primary OA technique for attaining isolation from change is use of standards. In addition, the OA approach makes use of a category of support software known as ***middleware*** to manage change.

Downloaded from https://www.everyspec.com



**Table 1-2. Application Portability Strategies**

| Method | Definition | Usage | Benefits | Comments |
|---|---|---|---|---|
| Standards | Use of widely recognized standards (e.g. international) to ensure application portability | Basis for many systems | Many vendors and products<br>OSJTF endorsed | Standards do change slowly, must be tracked |
| Widely used products | Use of commonly available non-standard products, e.g. MicroSoft | Widespread in practice, often business-based | Readily available products<br>OSJTF 2nd choice | Vendor lock<br>Market captive<br>Must upgrade |
| Porting | Moving applications from one set of technology products to another (e.g. OS, M/W) by changing APIs within applications | Often used to move from non-standard basis to standards | Low or no cost if porting within standards family | Not all products exactly match standards |
| Virtual machine | Run-time interpreter of machine independent version of source code (e.g. Java byte code) | Widely used with popular Java language | Portable code | Not good for real-time apps |
| Wrappers | Software layer that hides a variety of products below and exports a common interface to applications | Used to hide non-standard products | Low cost within domain of use | Vendor lock<br>Slows migration to standards |
| Emulation | Software layer that makes one type of computer appear to be another type by "emulating" the missing computer's instructions | May be used for legacy and obsolescent systems | Software does not have to change to new computer type | Overhead<br>Architectural obsolescence |
| Model Driven Architecture | Use of highly abstracted modeling tools (e.g. UML) for design; source code generated automatically | Not yet widely used but holds great promise | Hides details<br>May enable auto code generation | Still maturing |

* Primarily addresses portability across operating system and middleware products; modern processors are largely (but not completely) abstracted by OS & M/W. Thus, portability at OS & M/W level supports processor & hardware independence.

Middleware comes in two widely used types, adaptation middleware and distribution middleware. Technical details of these technologies are discussed in Sections 3.2.2.6 and 3.2.2.7, respectively. A third type of support software, frameworks (see Section 3.2.2.8), provides higher-level functions such as thread management, commonly used components, etc. A fourth type, known as RM, provides system management functions that are not strictly middleware in nature but that, like adaptation and distribution middleware, provide vital services in the composition of large systems. Dynamic RM is discussed in Section 3.2.2.9.

Together, these four types of support software provide a synthesizing function that allows applications in a distributed system to interface with each other and with the underlying computing equipment and operating system. Figure 1-2 illustrates the relationship of these software services to applications and to computing equipment and operating systems.

Downloaded from https://www.everyspec.com



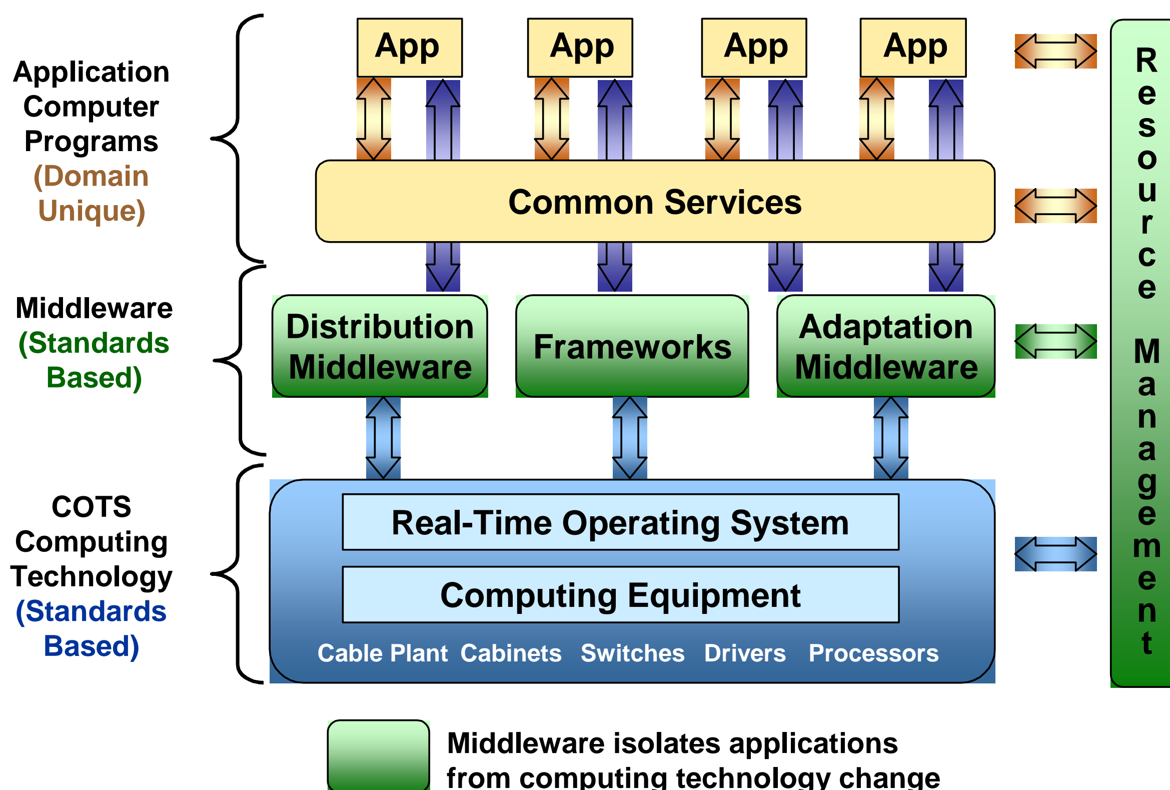


**Figure 1-2. Managing Change with Middleware**

Each of the categories in Figure 1-2 will, at times, undergo change. Application programs, even those designated as common services, change as missions, threats, warfighting capabilities, and warfighting system technologies change. Computing equipment and operating systems change as computing technologies evolve and improve. Furthermore, even though middleware performs an isolation function, it too will change as the underlying computing equipment and operating systems change. In fact, the middleware's interface to the technology base *should* change if applications are to be isolated from changes in the technology base.

Since middleware must change to keep pace with evolving computing technology, its implementation and support are equally as important to overall life-cycle support considerations as are the application computer programs and the computing technology base. The same ground rules that apply to the computing technology base should apply to middleware. Middleware should conform closely to standards whenever possible. Business considerations such as market share, broad customer base, and corporate stability are all valid considerations when selecting middleware products and vendors.

In fact, it is far more effective from a life-cycle supportability perspective if middleware is provided by the commercial computing industry (that also provides other computing technologies) rather than by application development companies. In the former case, the cost

Downloaded from https://www.everyspec.com



associated with middleware change and verification is shared among all customers using the technology. Otherwise, the cost of change should be borne by the developer of the system. Furthermore, since the middleware developer in this latter case is tied closely to a particular application domain, the likelihood is high of the middleware evolving toward a specialized and non-standard set of functions. Note that it is also highly desirable to have multiple standards-compliant implementations from which to select. In the case of the standards chosen for OA, this is generally the case (see *Open Architecture Computing Environment Technologies and Standards*).

### 1.3.5 Non-Standard Products

Special discussion is appropriate for certain classes of products that support design and development of easily modified complex systems but that do not adhere to the requirement for public consensus-based (or ***de jure)*** standards. Within the computing industry, a number of products exist that are not standards-based but that, nonetheless, have widespread market acceptance or that confer certain compelling advantages to system designers, despite their lack of standardization. Often such products constitute market-based (or ***de facto)*** standards, which have benefits that cannot be overlooked. Potential exceptions to use of industry standards include the following categories of products:

a. Products conforming to emerging standards that have not been finalized but that appear likely to do so

b. Widely used products with a high degree of market acceptance and stability

c. Widely used open-source products for which commercial support exists

d. Non-standard products providing unique or otherwise unavailable functionality, performance, flexibility, or cost benefit to the Navy

While such products frequently offer enticing advantages, it is important to note that use of these products may subsequently place system developers and acquisition managers in the position of having no competitive alternative to the products chosen. For this reason, a sound acquisition plan should require a thorough process for identification and justification of the need for non-standard products as well as management review and approval of the use of such products prior to initiation of development. When a standards-based product will serve equally as well as a non-standard product—both in performance and in a business-case sense—the standards-based product should be chosen.

### 1.3.6 OACE Change Management

Errata for this document are maintained at https://viewnet.nswc.navy.mil.

Downloaded from https://www.everyspec.com



Current Mainstream COTS computing technology meets many, but not all warfighting computing requirements. However, the pace of computing technology innovation has been very rapid for decades and shows little signs of slackening. Thus in the future, mainstream products may meet many requirements that currently are met only by special purpose solutions. Because of this rapid evolution, the boundaries between what is within OACE scope and what is not will require periodic reconsideration.

For this reason, an OACE change management process will be formally documented in the future through the Open Architecture Enterprise Team (OAET) established by ASN RDA August 2004. This formal process will provide for periodic review of the guidance contained in this document. This change process, cyclic in nature, will include mechanisms for incorporating the lessons learned and best practices of each program manager as well as inputs from industry.

## 1.4 DOCUMENT OVERVIEW

Section 2, Applicable Documents, provides applicable mandates and directives that trace down to the guidance addressed in this document. Section 3, Open Architecture Guidance, provides the characteristics of the system that will eventually be required for its acceptance.

Downloaded from https://www.everyspec.com



# SECTION 2

# APPLICABLE DOCUMENTS

The following documents are technical publications, directives, official correspondence, or instructions that affect the guidance put forward in this document.

a. ASN (RD&A). *Naval Open Architecture Scope and Responsibilities*, memorandum. 5 August 2004.

b. *Department of Defense Joint Technical Architecture*, Version 4.0. 17 July 2002.

c. ASN (RD&A). *ASN (RD&A) Commitment to Congress to Completely Phase Out AN/UYK-43 Computers by FY 99 Ships*, letter. 1 December 1999.

d. *Baseline 7 Ships Characteristics Improvement Board (SCIB) Report*. November 1992.

e. Bertrand Meyer. *What to Compose*. Software Development Magazine. March 2000.

f. Chief Naval Operations (CNO) LTR 3900 ser N865G/4U652928. 22 July 1994.

g. Christopher Alexander, Sara Ishikawa, Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press. 1977.

h. CNO LTR ser N864/6U656004. 21 November 1995.

i. *Aegis Combat System Baseline Plan: Annual Update*. CNO LTR ser N865G. 21 December 1996.

j. Defense Information Systems Agency (DISA). *Common Operating Environment (COE) Integration and Runtime Specification (I&RTS)*, Version 4.2. February 2002.

k. *The Defense Acquisition System*; Department of Defense Directive (DoDD) 5000.1. 12 May 2003.

l. *Mandatory Procedures for Major Defense Acquisition Programs (MDAP) and Major Automated Information Programs (MAIS) Acquisition Programs*; DoDD 5000.2-R. 10 June 2001.

m. Department of the Navy (DoN) Chief Information Officer (CIO) Memorandum. *DoN Policy on the Use of Extensible Markup Language (XML)*. 13 December 2002.

n. DoN CIO. *DoN XML Developer's Guide*. October 2001.

o. DoN CIO. *Information Technology Infrastructure Architecture (ITIA)*, Version 1. 16 March 1999.

Downloaded from https://www.everyspec.com



p. DoN CIO. *Information Technology Systems Guidance (ITSG)*, Version 1. 1999.

q. Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. 2000.

r. Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1995.

s. Extensible Markup Language (XML) 1.0. *W3C Recommendation*. 6 October 2000.

t. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. New York, New York: John Wiley & Sons Ltd. 1996.

u. *Guidance Document for Aegis Baseline 7 Phase 1 and II Specification Development: Information Architecture and Baseline Applicability*, Version 1.0. 20 March 1998.

v. *Base Definitions - Portable Operating System Interface (POSIX) Base Definitions*, Issue 6; Institute of Electrical and Electronics Engineers (IEEE) Std 1003.1-2001. 2001.

w. *Ada Programming Language*. International Standardization Organization/International Electrotechnical Commission (ISO/IEC) 8652:1995. 1995.

x. *C Programming Language*. ISO/IEC 9899:1999. 1999.

y. *C++ Programming Language*. ISO/IEC 14882:1998. 1998.

z. *Java™ 2 Platform Enterprise Edition (J2EE) Specification*, Version 1.3. July 2001.

aa. Object Management Group (OMG). *Common Object Request Broker Architecture (CORBA)*, Version 2.6.1. May 2002.

bb. OMG. *Data Distribution Service for Real-Time Systems Specification, Final Adopted Specification*. http://www.omg.org/docs/ptc/03-07-07.pdf. May 2003.

cc. OMG. *OMG Unified Modeling Language Specification*, Version 1.5. March 2003.

dd. *Commercial Item and Non-Developmental Item Selection, Acquisition, Integration, and Life-cycle Support Policy*. PEO Theater Surface Combatants (TSC) 4890.1.

ee. RADM Murphy Briefing. *Surface Warfare Road Map*. August 1996.

ff. *Acquisition Management Policies and Procedures for Computer Resources*. SECNAVINST 5200.32A.

Downloaded from https://www.everyspec.com



gg. Secretary of Defense Memorandum. *To meet future needs DOD should increase access to commercial state-of-the-art technology*. 29 June 1994.

hh. White Paper. *Forward From the Sea*. 19 September 1994.

ii. *Open Architecture Computing Environment Technologies and Standards*, August 23, 2004.

jj. ADM Vern Clark USN. Naval Institute Proceedings, Vol 128/10/1, 196. *Seapower 21: Projecting Decisive Joint Capabilities*. October 2002.

Downloaded from https://www.everyspec.com



# SECTION 3

# TECHNICAL ARCHITECTURE GUIDANCE

The Navy intends to implement warfare systems that meet operational performance requirements and are affordable. System affordability may be achieved in substantial measure by implementing application computer programs in an open fashion on computers that are in the mainstream commercial market. The mainstream commercial marketplace typically offers a diversity of vendors, produces large quantities of products, and offers the products at competitive prices. The word ***mainstream*** is critical. A niche market COTS product or one whose commercial future is uncertain lacks many of the desirable open-system characteristics stated in Section 1.3.1, even if it is driven by formal standards.

## 3.1 TECHNICAL OVERVIEW

The NWS will transition from a closed architecture to one based upon open-system design principles as outlined in Section 1.3. Requirements for increased processing, memory, performance, and interconnectivity to support cost-effective incorporation of these and future capabilities dictate this transition. OA also acts as an enabler, allowing extension of processor homogeneity (see Section 3.2.2.3) throughout the NWS, thus allowing for the pooling of processor resources to realize the benefits associated with fully distributed computer programs. These benefits are increased scalability, improved fault tolerance, and reduced logistics infrastructure.

In the case of new construction combined with new requirements, the transition to the highly modular open design will be direct; modern design principles will be employed from initial design onward. In the case of backfit and legacy system conversion, business case analysis will to a significant degree dictate what can be done and at what pace. Some systems, particularly at the periphery of the combat system (primarily sensor and weapon systems) may experience little if any change other than perhaps at the interface. Such systems might be called "legacy for life" as viewed by OA, though certainly their functional capability may be upgraded over time. In the case of core sensor control, command and control, and weapons control, the need to make more substantial modifications over time is greater in order to promote source code commonality across ship classes and platform types. Sensors and weapons provide less opportunity for code sharing and reuse; thus, there is considerably less of a business case to expend the funds necessary to incorporate modern modular design principles.

The scope of the redesigned NWS includes the underlying technology base, as well as the applications themselves. The OA design will be built on an infrastructure of technologies that include cable plant, cabinets, network components, processors, operating systems, adaptation and distribution middleware, frameworks, RM, and other services. This infrastructure is referred to as the Open Architecture Computing Environment (OACE). OACE (as described in Section 3.2.2) will allow the Navy to introduce and change out commercial technology to maximize affordability and performance goals.

Downloaded from https://www.everyspec.com



### 3.1.1 Functional Partitioning

The starting point for OA evolution is a partitioning of NWS functionality that will create a new component-based logical architecture no longer limited by design constraints arising from the previous generation of computing technology, as reflected in the current element-based architecture. The selection of partitioning boundaries will be accomplished by examining the entire NWS functionality set during the design stage. However, implementation may be accomplished in an evolutionary and phased manner consistent with funding availability. Due to constraints imposed by the legacy architecture, selection of implementation phases should necessarily take current element boundaries into consideration (see Figure 3-1. The figure provides a notional representation of this distinction. This diagram represents a typical hierarchical functional view of system design using a traditional structured analysis approach.

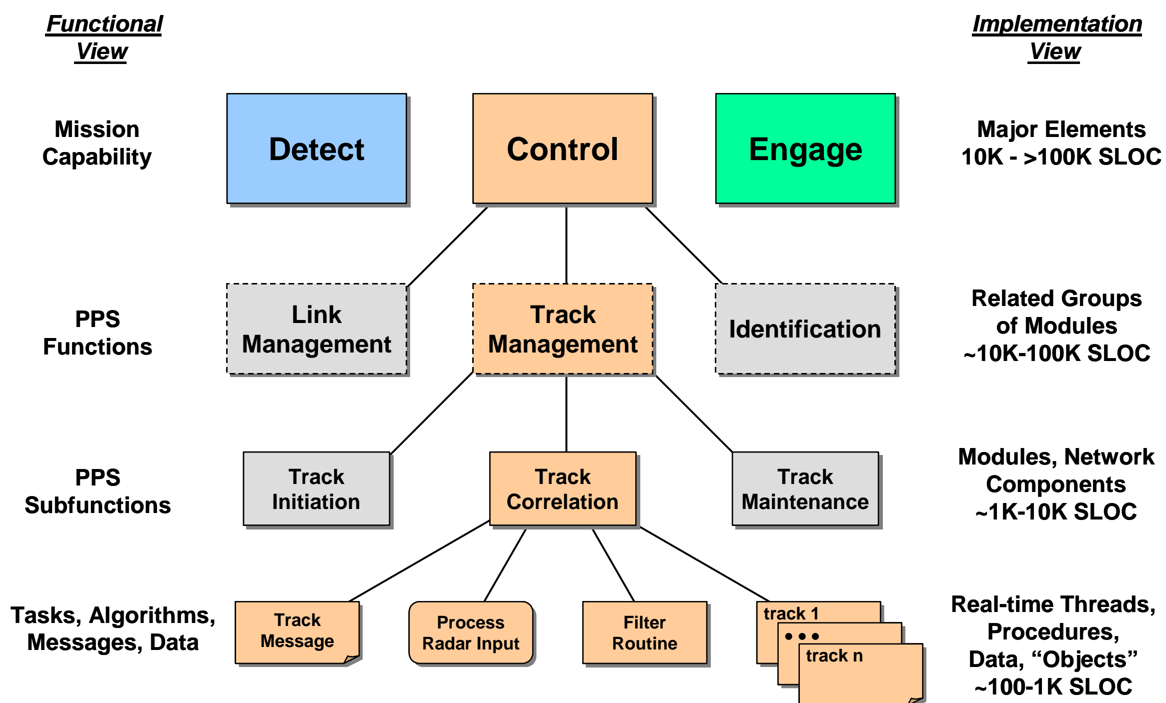


**Figure 3-1. Functional Partitioning**

The logical architecture will be defined so as to incorporate all specified NWS capabilities and requirements. These capabilities are defined by the appropriate ORDs and by the relevant combat system requirements and specifications. The details of these requirements and specifications are the responsibility of individual Programs of Record (PORs) and are thus not exhaustively cited here.

The functional architecture should be as technology-neutral as possible. The purpose of this approach is to ensure that long-term supportability goals are met. Only after the functional

Downloaded from https://www.everyspec.com



architecture is defined can definition of areas where code reuse is possible be performed with certainty. Implementation will utilize best commercial design and development practices and state-of-the-practice technologies consistent with NWS mission requirements.

Component boundaries and interaction requirements will be established in conjunction with and consideration of timing and performance characteristics. The implementation and deployment architecture will be established by finding the best technology that satisfies overall design objectives. The implementation architecture deals with technology choices and grouping of components to minimize delays or latency based on the technology constraints. Subsequent new construction and backfit efforts will use the same architectural framework to select new technologies. This allows the components to be reallocated as technology constraints change.

### 3.1.2 Computing Characteristics

The OA functional partitioning should support insertion of new capabilities not previously affordable, in part because it will be possible to add new capabilities on a shorter and lower cost upgrade cycle than in the past. This is, in substantial measure, true because information within an open NWS implementation will be much more readily available than in current deployed systems. The use of networks, client-server implementations, multicast-based middleware, general purpose computing equipment and services, etc. allows convenient access to the information and computing resources needed to incorporate new functionality. This, in turn, has the potential to provide a computational base for increased automation in support of manning reduction.

NWS communications will be based on Local Area Network (LAN) technologies. Limited Navy Tactical Data System (NTDS) legacy interfaces will continue to be supported. However, where feasible, NTDS legacy interfaces should be transitioned to the OA LAN-based architecture. This transition may not be cost effective in many cases. But in some cases, the use of hardware adapters that convert NTDS channels to LAN connections may be possible. In such cases, the NTDS device effectively becomes part of the network.

OA does not relieve NWS integrators from meeting all ship environmental requirements. Mission-critical enclosures will house commercial-based advanced processors, network, and conversion equipment. The legacy equipment footprint may be retained for backfit platforms where a cost benefit case cannot be made for ship design changes. In that case, the number and placement of enclosures as well as inter-compartment cabling will remain fixed. Where it is cost effective to change cabinet and cable configurations and is beneficial to do so, such changes should be considered. In any case, the NWS computing capability should be capable of uninterrupted operation under battle conditions that include but are not limited to shock, fire, and salt water spray. This should be accomplished in a manner that is survivable, maintainable, and affordable.

Downloaded from https://www.everyspec.com



### 3.1.3 Federated vs. Integrated Architectures

It should be acknowledged that the OA goal of commonality is, to some degree, in tension with providing maximum flexibility of choice to acquisition managers. Traditionally, warfighting systems have been assembled as collections of systems and subsystems, each separately and independently developed using its own choice of computing resources—i.e., a ***system of systems*** approach. However, a major new trend in computing system operations has emerged—enterprise management. Enterprise management allows an entire set of application computer programs to be managed as a whole on a unified set of computing resources—resources that are often shared interchangeably among applications. The notion of RM, perhaps dynamically and in near-real-time, is an integral part of enterprise management.

Given that the technology of enterprise management is relatively new, questions arise as to how much and how soon and in what circumstances it should be adopted for mission-critical and real-time systems. The following sections discuss the tradeoffs involved.

#### 3.1.3.1 Federated Approach

The term ***federated*** is used to describe an approach where selection of implementation technologies is minimally restricted for developers. Interface methods are enforced only at interfaces and not internally. In the federated approach, each system developer chooses processors, operating systems, middleware, and other infrastructure components without regard to choices made by other system and subsystem developers. In this case, resources are dedicated to the subsystem, and there is little if any opportunity for resource sharing. An oft-claimed advantage of the federated approach is that it allows maximum flexibility to meet stressing or system-unique requirements through selection of leading-edge technologies.

#### 3.1.3.2 Integrated Approach

The term ***integrated*** is used to describe an enterprise-managed approach that makes maximum use of commonality, both at the physical level and at the functional level. The integrated approach uses a common resource base to create a pool of computational resources that serves many, if not all, application needs. Applications that run within this pool of resources must be built so that they are not uniquely mapped to a particular processor or network Internet Protocol (IP) address. (Most modern middleware implementations support this ***location transparency*** feature via a name service.)

For new construction (and backfit where possible), the ***pool-of-computers*** approach allows any computer program to run on any available processor, subject to limitations imposed by high-performance real-time requirements. Fault tolerance and survivability is based on an M+N redundancy scheme, where M is the number of processors required to perform mission computing tasks and N is the number of spare processors.

Furthermore, the use of a ***pool-of-computers*** architecture for the resource base combined with the application of RM enables mission flexibility through warfare system-wide operational resource sharing and enhanced failure recovery through a high degree of redundancy. This

Downloaded from https://www.everyspec.com



service is available to all systems that are able to participate in the integrated approach, but it does not preclude employment of the federated approach, even within an integrated system, for subsystems whose requirements suggest otherwise. The introduction of OA as an enterprise effort across naval systems allows for the development of both integrated systems and federated systems. With due care standards, interfaces and functional components can be used simultaneously by both integrated and federated systems. The architecture also has the potential to change the approach to computer system maintenance in warfare systems. It may be possible to substantially reduce shipboard computer maintenance by capitalizing on the fact that application components are not bound to computer locality but instead are free to migrate to available processors under RM control. Reduction of shipboard maintenance should lead to reduced manning.

#### 3.1.3.3 OA Guidelines

Individual acquisition programs have frequently enforced the integrated approach within their own scope of control but have objected to its application to them from without. In part, this state of affairs reflects the difficulties of making "one size fit all," notwithstanding the benefits of commonality. The approach chosen for OACE is to encourage standards-based family similarity of computing resources across all warfare systems by making commonality beneficial without requiring absolute conformity to exact product selections.

Finally, note that the terms integrated and open are not synonymous. Traditionally, the term ***open*** has been more likely to invoke the expectation of a federated approach rather than an integrated one. However, an integrated approach is consistent with the principles of open-system design so long as a framework of established standards is used. For integrated systems, the adoption of a shared resource base is a design decision made in the interests of enhanced mission flexibility, survivability, and maintenance-free deployments.

### 3.1.4 Implementation Constraints

OA represents an opportunity to reinvigorate the performance and maintainability of NWS with a flexible functional allocation, a distributed computing architecture and mainstream standards-based technologies. Likewise, the investment in physical design and construction of existing assets should be considered. Thus, while it is possible to change computer programs, and even network and processor components, without impacting combatant construction, it is considerably more difficult to change cabling and cabinetry. For this reason, once the best technical design is identified, cost-benefit tradeoffs will evaluate any proposed physical design changes that may impact physical ship construction and modernization costs and schedules.

With the above caveat in mind, Table 3-1 lists initial OA design constraints. In the table, the term ***mainstream*** captures an important concept that contributes to achieving OA objectives. Generally, it connotes the idea of product lines with large market shares, extensive user base, readily available support, and vendor stability. Standards bodies embraced for the OA initiative include but are not limited to Telecommunications Industry Association (TIA), Internet

Downloaded from https://www.everyspec.com



Engineering Task Force (IETF), Object Management Group (OMG), and the IEEE's Portable Operating System Interface (POSIX) standard group.

**Table 3-1. OA Constraints**

| SYSTEM DOMAIN | CONSTRAINTS |
|---|---|
| Cable Plant | Use TIA standards-based COTS cabling technology when possible. |
| Enclosures | Use enclosures capable of providing MIL SPEC quality environmental conditioning for mainstream non-MIL SPEC COTS components. |
| User interface devices | Use consoles, displays, and other user interface devices consistent with Navy acquisition processes. |
| Networks | Use mainstream IETF standards-based COTS network technology. |
| Computers | Use mainstream COTS products, e.g., single-board computers, servers, blades, storage systems, etc. |
| Operating Systems | Use mainstream POSIX standards-compliant operating systems. |
| Middleware | Use mainstream OMG standards-based middleware. |
| Resource Mgmt | Implement NWS-wide static and dynamic capabilities. |
| Common Services | Implement common support services as system server applications (e.g., time services, data extraction, navigation data distribution, web servers, application servers, Relational Data Base Management System (RDBMS) servers, etc.). |
| Application Design | Redesign and/or refactor tactical applications (as required) as components in accordance with open-systems concepts outlined in this guidance document. Use mainstream COTS and open-source development tools. |

### 3.1.5 OACE Compliance Categories

The guidance in this document applies to both new construction and backfit. In the case of backfit ships and warfighting systems, some additional constraints may necessarily apply on a case-by-case basis. For instance, the *Open Architecture Computing Environment Technologies and Standards* document defines four OACE compliance categories:

a. Category 1 – Hardware adapters

b. Category 2 – OACE interfaces

c. Category 3 – OACE standards

d. Category 4 – OA common services and functions

Downloaded from https://www.everyspec.com



Category 1 reflects non-OACE-based technologies and is thus not OACE compliant. Category 1 is appropriate for a system that is in an extended legacy state and for which no functional commonality is envisioned. Such systems are often found on the periphery of a warfighting system rather than in the core segment, and they tend to have only a limited number of interfaces to the combat system. In this case, there is little business case justification for migrating them to Category 3. However, if a Category 1 system is in the core of the combat system, and particularly if it is a likely candidate to become a common function, its migration to Category 3 is a virtual necessity.

Category 2 uses adaptation layers to isolate existing software built on proprietary or legacy operating systems and middleware from OACE standards-based operating systems and from middleware interfaces to OACE-based external systems. Thus, Category 2-compliant systems interface with other systems via OACE-based middleware but may use other middleware internally. Where there is no requirement for functional commonality, there is no urgent business case reason to migrate Category 2 systems to a higher category unless the system is undergoing upgrade for other reasons, in which case it should be considered for migration if the business case supports it. However, when the functionality embedded in the Category 2 system is a candidate for adoption as a common function, its migration to Category 3 is necessary.

Categories 3 and 4 are fully OACE compliant. Category 3 compliant systems use OACE standards throughout, both internally and externally. Category 4 systems are both Category 3 compliant and based on OA common functions and services. Also, internal subsystem interfaces to embedded common services or common functions should use OACE standards at the interface. A minimum of Category 3 OACE standards compliance is required as a prelude to making services and functions common to enable their reuse across ship classes and warfighting systems.

As previously stated in Section 1.2, the guidance contained in Section 3 is presented for descriptive and informational purposes and is not to be construed as contractually binding unless so designated, in whole or in part, by separate mechanism. Specific compliance statements regarding individual standards have been formulated and are contained in the *Open Architecture Computing Environment Technologies and Standards* document.

### 3.1.6 Development Processes and Tools

Effective processes, methodologies, and tools are critical to the success of large-scale engineering projects. Key characteristics of the incumbent process include a strong emphasis on firm definition of requirements prior to design and coding, as well as a formal design review process involving Navy buy-off at every stage of the development. When required, warfighting changes were extensive in scope; Engineering Development Models (EDMs) usually preceded production development as a means of risk mitigation. For systems migrating to OACE Category 3 or Category 4 compliance and for common functions, the Open Architecture Test Facility (OATF) provides a vehicle for pre-production evaluation.

Downloaded from https://www.everyspec.com



The Object-Oriented Programming (OOP) method did not originate in the real-time community, but it has matured to the point where the performance characteristics of OOP computer programs are, in some cases, adequate to meet real-time requirements. The OOP approach not only supports critical OA objectives, such as affordability and time-to-market, but also represents state-of-the-practices among trained engineers within the pool of likely OA implementers. Where OOP technology does not support real-time requirements, its use should be avoided.

Given this situation, the time has come to update processes, methodologies, and tools to the current generation as a part of OA development. The new spiral approach is more iterative than previous approaches, due in no small measure to the degree of automated and collaborative design that is now possible. Unified Modeling Language (UML) and OOP allow engineers to express and capture requirements and design in a readily transportable and interchangeable form. Web-based Integrated Data Environments (IDEs) provide immediate access to project artifacts. When coupled with a management philosophy that promotes early and broad access to intermediate products by IPTs via the IDE, this approach allows maximum visibility to implementers and customers alike.

Not withstanding the advantages of new tools and methods, perhaps their most important advantage is the ability they convey to overlap requirements setting, system engineering, design, and implementation successfully. In the past, substantial parallelism across these stages was considered off process and was often not formally allowed. This resulted in delaying program direction changes in response to problems uncovered during development, with consequent schedule and cost disruption when such problems were inevitably encountered. However, where tools permit and where traditional engineering rigor and discipline are sustained, such overlap provides a useful mechanism for identifying and addressing problems early in the development cycle.

Having stated the preceding benefits of new tools, understanding how the process should be applied is important if these benefits are to be achieved. Two criteria for selection of candidates for early definition and prototype implementation include high-risk capabilities and foundation functionality. Each has its merits. In the case of high-risk functions, the benefit is obvious. The more time and effort devoted to resolving risks, the more likely it is that the overall project will be a success. Likewise, foundation functionality is the capability upon which subsequent system operations should depend. Examples include technology-based infrastructure components, such as networks, operating systems, middleware management, and RM; and core system functionality, such as track formation, common tactical picture, and display.

The traditional structured approach to system engineering and design might reasonably be characterized as ***top-down*** and ***breadth-first***, with a major focus on a sequential customer/developer review and approval process exercised on a step-by-step basis. By contrast, the new approach is simultaneously top-down but with ***depth-first*** or even ***bottom-up*** excursions in critical risk and foundation areas. The new process also includes a major focus on system architecture as well as technical architecture. In summary, the new methods and tools enable a more fine-grained and risk-aware development process without implying or demanding

Downloaded from https://www.everyspec.com



compromise of the disciplined top-down approach that is integral to successful large-scale system engineering endeavors.

## 3.2 TECHNICAL ARCHITECTURE GUIDANCE

The technical guidance points discussed herein describe certain critical technical application design characteristics that support OA; therefore, they are a primary content of this guidance document. However, they may differ in certain aspects from a more general description of open systems since these attributes might be applicable to a broad base of non-real-time and non-mission-critical applications. Where appropriate, these differences reflect the more stringent requirements of real-time, mission-critical systems.

The level of detail provided in the following sections is, in general, below that associated with requirements. The intent of the guidance is to characterize design techniques that have proven successful in other systems and prototypes comparable in complexity and/or stressing requirements to NWS, e.g., commercial, business, financial, medical, telecommunications, and process control applications, to name a few. Unless otherwise noted, guidance alternatives are not necessarily all inclusive. Thus, additional implementation possibilities may exist.

Technical guidance for OA is divided into two categories: the attributes of application computer programs and the attributes of the underlying technology base. Section 3.2.1 provides a detailed discussion of component design characteristics and constraints. Section 3.2.1.1 introduces the concept of architecture patterns; i.e., ways of building software that are useful in many different forms across a wide variety of applications. Section 3.2.2 covers the characteristics of the underlying technology base. Table 3-2 below contains an index to all guidance sections, both application design related and OACE infrastructure related.

**Table 3-2. Technical Architecture Guidance Index**

| CATEGORY | PARAGRAPH | PAGE |
|---|---|---|
| Component design | 3.2.1.1 | 3-12 |
| Portability | 3.2.1.3 | 3-22 |
| Location transparency | 3.2.1.4 | 3-25 |
| Client server | 3.2.1.5 | 3-26 |
| Data distribution | 3.2.1.6 | 3-31 |
| State data coherency | 3.2.1.7 | 3-34 |
| Computational flow | 3.2.1.8 | 3-36 |
| Fault tolerance | 3.2.1.9 | 3-37 |
| Scalability | 3.2.1.10 | 3-43 |

Downloaded from https://www.everyspec.com



| CATEGORY | PARAGRAPH | PAGE |
| --- | --- | --- |
| Real-time performance | 3.2.1.11 | 3-49 |
| Process, thread & memory management | 3.2.1.12 | 3-55 |
| Data Brokers | 3.2.1.13 | 3-57 |
| Cabling and Cabinets | 3.2.2.1 | 3-63 |
| Information Transfer | 3.2.2.2 | 3-64 |
| Computing Resources | 3.2.2.3 | 3-66 |
| Peripherals | 3.2.2.4 | 3-70 |
| Operating Systems | 3.2.2.5 | 3-70 |
| Adaptation Middleware | 3.2.2.6 | 3-71 |
| Distribution Middleware | 3.2.2.7 | 3-72 |
| Frameworks | 3.2.2.8 | 3-77 |
| Resource Management | 3.2.2.9 | 3-79 |
| Instrumentation | 3.2.2.10 | 3-83 |
| Failure Management | 3.2.2.11 | 3-85 |
| Information Assurance | 3.2.2.12 | 3-86 |
| Time Service | 3.2.2.13 | 3-87 |
| Programming Language Facilities | 3.2.2.14 | 3-88 |
| Displays | 3.2.4 | 3-89 |
| System Test and Certification | 3.2.5 | 3-90 |
| Selection of Standards | 3.2.6 | 3-92 |
| Open Source Products | 3.2.6.2 | 3-94 |

### 3.2.1 Component Design Guidance

The term ***component*** is broadly defined. It may refer to service libraries, widely applicable domain-specific classes, compiled and linked executable programs, or entire subsystems. In OA, those components that form application computer programs should be implemented as loosely coupled software components. Each such component is a separately compiled and linked executable computer program or process in POSIX operating system terminology. ***Loosely coupled*** means that these programs communicate with each other via an explicit message passing or distributed object invocation mechanisms. The contrasting case, ***tightly coupled***, means that communication takes place via explicit reference to shared memory. In a full OA implementation, a large-scale infrastructure of processors, networks, and services

Downloaded from https://www.everyspec.com



supports the distribution and execution of application components such that both scalable performance and system-fault tolerance are achieved.

It should be noted that the component design guidance contained in this section does not, in fact, constitute mandatory requirements but rather reflects to the greatest degree practical a description of the current state of the practice as far as designing distributed, mission-critical applications. Thus, this section is primarily tutorial in nature. The focus is necessarily on architecture patterns (design techniques and templates) that provide real-time performance, efficient use of computational resource, fault tolerance, etc. However, the use of these guidance principles is readily extensible to less than real-time requirements as well. It should be noted that in the latter case, new web service-based design techniques are being widely employed in commercial and business applications. In the case where non-real-time applications must interface with real-time, mission critical applications, guidance remains to be developed that will suggest suitable interface techniques.

Attributes of the OA application computer programs are listed below and described in the subsections that follow.

a. Functionally distinct self-contained applications or components, usually modest in size

b. Components loosely coupled in space and time with other components

c. Applications built for portability and location transparent allocation and operation

d. Client-server design patterns with push and/or pull interfaces

e. Mechanisms for distribution of continuously refreshed data

f. Maintenance of state data coherency in components with composite state

g. Computational flow and multi-component paths

h. Multiple fault tolerant design patterns (active, passive and hybrids)

i. Scalability via load-sharing peer-client design pattern

j. Quality of Service (QoS) mechanisms for support of both hard real-time and soft real-time requirements, possibly collocated

k. Multi-threaded applications (input/output [I/O], periodics, etc.) with threads not hard mapped to Central Processing Units (CPUs)

l. Pre-allocation of computing resources (memory, processes, threads, etc.) at application initialization

Downloaded from https://www.everyspec.com



m. Mechanisms for preservation of data integrity (correctness and resistance to disruption) across threads, processes, computers, and networks

n. Asynchronous processes with timed/fixed packet I/O buffering

o. Push distribution (forward caching) of track data to consumers

p. Data broker/adapter design patterns for legacy capture

q. Any-operator, any-console display architecture

In the above list, the term ***QoS*** describes various properties associated with providing varying degrees of assurance that requirements will be met. QoS properties are generally controlled via mechanisms embedded in either the operating system or the network protocol stack (including not only computer resources but also network equipment such as switches and routers). For real-time applications, this may include use of operating system priorities to ensure timely wakeup of periodic processing threads as well as use of network bandwidth reservation to ensure timely message delivery. Analogously, non-real-time applications may be run at low priority and without bandwidth reservation to ensure that such applications do not monopolize system computing resources at the expense of high-priority real-time applications. This is particularly important where both types of applications may be co-located on the same processor.

The attributes of the technology base needed to support these application component attributes are described in Section 3.2.2. For OA, the following global guidance should be incorporated into each tactical component, as applicable: all NWS functions should be implemented as distributed components designed specifically for the OA computing environment. The notion of architecture patterns is introduced in Section 3.2.1.1. Individual patterns are discussed in Sections 3.2.1.5 through 3.2.1.10 and 3.2.1.13. Hard real-time and soft real-time characteristics are discussed in Section 3.2.1.11.

#### 3.2.1.1 Architecture and Design Patterns

Patterns are widely useful reusable rules and methods for building components and systems, including open systems. Architecture patterns constitute high-level structures appropriate to design of major segments of software, such as computer programs and components. They may foster key open-system properties such as code reuse, flexibility of design, enhanced interoperability, etc. Design patterns are more detailed and discrete templates. The following definition of patterns comes from Alexander (1977). Although it deals with patterns in buildings and towns, nonetheless, its generality makes it highly applicable to computer systems as well.

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."*

Downloaded from https://www.everyspec.com



Extending this definition, designers Gamma, Helm, Johnson, and Vlissides state the following in their book, *Design Patterns: Elements of Reusable Object-Oriented Software*: "[A]t the core. . .of patterns is a solution to a problem in a context." They list the following four characteristics as the essential elements of a pattern:

> *"The pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary."*
>
> *"The problem describes when to apply the pattern. It explains the problem and its context."*
>
> *"The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations."*
>
> *"The consequences are the results and trade-offs of applying the pattern.*"

Certain important design patterns recur in open systems. The patterns listed below, as well as the tools required in their construction (largely middleware) have shown that they have value in the real-time weapon system problem domain. They will be discussed in more detail in subsequent sections. In many cases, application components may exhibit aspects of more than one pattern. The key patterns include:

a. Client-server patterns

b. Distribution patterns for continuously refreshed data

c. State data coherency maintaining patterns

d. Computational data flow patterns

e. Replication patterns for fault tolerance

f. Peer-client replication pattern for load-sharing and scalability

g. Inter-component adaptation patterns (e.g., brokers, translation layers)

Patterns describe a particular way of solving a problem. A pattern may be a high-level abstraction of a solution, such as a client-server, or it may be a very detailed recipe for solving certain classes of problems. Patterns may be presented in different forms. They may be described with detailed written specifications. UML class diagrams and object models are often used to convey patterns. Detailed constraint models sufficiently rigorous to describe all conditions, processing, and outputs have also been employed as pattern languages. In any case, the pattern is definitely a solution and not just a description of the desired result or the environmental constraints.

Downloaded from https://www.everyspec.com



#### 3.2.1.2 Component Design

This section provides rationale and requirements for partitioning NWS functional capability into medium-grain, functionally homogeneous, loosely coupled code units or components that are suitable for distributed processing.

##### 3.2.1.2.1 Description

A key concept in structuring application computer programs for incorporation into OA is the advantageous partitioning of system functionality into code units, or ***components***, that are easy to change and interoperate with over the life cycle of the system. In this characterization, a component is a physical, replaceable software entity or element that performs defined functions and encapsulates data. A component is a distinct, separable, and uniquely identified constituent part of a system, with well-defined boundaries, capabilities, and interface to other system components. Note that, conceptually, components may be either hardware or software. However, in this context, only software components are intended. Both processes and libraries may qualify as components.

Middleware plays a key role in designing modern software components, particularly for Distributed Object Computing (DOC). The standard chosen for OA DOC middleware is the OMG's Common Object Request Broker Architecture (CORBA); (see *Open Architecture Computing Environment Technologies and Standards*). One of the major features that makes the CORBA standard attractive for OA use is the fact that it supports application software portability between different middleware vendors.

###### 3.2.1.2.1.1 Utility of Components

Components provide access to encapsulated functions and data via well-defined interfaces, and they exhibit predictable behavior. Components are usable by multiple heterogeneous clients without modification. Usually, process-based components are defined along boundaries that facilitate loosely coupled communications with other components. Moreover, the scope of a component should usually represent a lowest common denominator of capabilities, such that little, if any, further decomposition can be performed on the functionality while still preserving the loosely coupled character of the component and without introducing significant adverse affects on the efficiency and performance of the system.

Components are used to compose larger aggregations. Loosely coupled components can be joined together to create composite services, even if they have been implemented using different technologies, as long as they have a clear, well-defined API and common mechanisms to invoke that API. The entire set of components aggregated for an overall purpose constitutes a ***system***. However, smaller collections of components often arise as well. These smaller collections may have a degree of functional unity and utility that merit separate identification and tracking. Such collections of components are referred to as ***subsystems***. One example would be a data extraction and recording subsystem, which itself may be composed of a number of separable components. Just as components may be part of multiple systems, so subsystems may also be constituent parts of multiple systems.

Downloaded from https://www.everyspec.com



If a software element is understood generally as any collection of software built for a specified purpose, the notion of components introduces constraints on the choice of boundaries and structure—constraints that are intended to foster ease of composition, reuse, and other supportability metrics. Also central to the notion of a component is the concept of ***durability***. The durability metric suggests that functionality, APIs and inter-component interfaces should change little over time and throughout repeated functional enhancements to other system components. The term ***stability*** is sometimes used synonymously with ***durability***, but since stability is also used to describe performance characteristics, the term durability is chosen in order to remove ambiguity.

***Granularity*** is defined as the relative size, scale, level of detail, or depth of penetration that characterizes a software object or method activity. Component granularity can be viewed in two ways:

a. A software component can be either coarse grained or fine grained depending upon how much functionality is included within the component structure. Sometimes component granularity is characterized by the number of tasks it performs, with fine grained performing a single task and course grained performing multiple tasks. Generally business components are more coarsely grained (e.g., databases), while infrastructure components that provide lower-level applications services (e.g., file access) are fine grained.

b. A software component can also be characterized as course grained or fine grained depending upon the information access with the amount of information supplied by a software component.

Component granularity is a key design issue. From a functionality viewpoint, the benefit of a finely grained component is the generally greater decoupling provided with potentially greater reuse. From a data access viewpoint, the benefit of a course-grained component is the performance improvements gained by reducing the amount of processing required for container interceptions, data marshalling/demarshalling, and the additional network traffic. However, this also potentially provides a client with too much access and a potential security problem since secure data/methods may be opened up with this level of granularity.

When properly designed, components have a number of key advantages. They facilitate construction of reusable code units that can be inserted easily into a wide variety of systems without requiring knowledge of the internal implementation of the component. Furthermore, the stable component interface facilitates the evolution of a system through specialized, extended, elaborated, or optimized versions of the component.

Downloaded from https://www.everyspec.com



3.2.1.2.1.2 Composing Systems with Components

According to Bertrand Meyer (2000), "*the two basic conditions for a software element to be considered a component are that it be:*

- *Usable by other software elements. This excludes a program in the traditional sense that is meant to be used by humans or non-software triggers—unless it has been componentized, meaning precisely adapted for use by other software.*
- *Usable by software elements, (whose) authors are unknown to the component's authors. This excludes the case of routines, classes and other software elements used by other parts of the same software. A component should be of interest to a broad range of 'clients' not directly connected to the original authors.*"

The preceding points have direct applicability to the OA initiative in that OA has as a major objective getting to a point where the Navy "buys once" and reuses components across multiple ship classes. In any case, according to the author, "*These requirements, modest as they may seem, immediately lead to several others*." Among them are the following:

a. A component should include a specification of all its dependencies: hardware and software platform, versions, and other components. Otherwise, new clients will not be able to make good use of the component without going back to the original author.

b. For the same reason, a component should provide a precise specification of the functionalities that it offers.

c. The component should be usable on the sole basis of that specification, without access to non-interface information (such as the source code even if it is available). This leads in particular to the information-hiding requirements.

d. Components should be composable with other components, since a single component is not very exciting and certainly does not justify talking about component-based development. **In practice, this means that a good component will usually be part of a more general component framework with a clear overall architecture, style, and standard design patterns.**

e. The process of integrating a component into systems should be fast and smooth.

With regard to the composability characteristic above, it should be noted that integration of a component that supplies a new service requires creation of new clients and will always take longer than integration of a new client or a server that takes over an existing interface. In any case, components are usually loosely coupled to other components (i.e., communicate via message passing); however, they may be tightly coupled internally via shared memory if system performance needs so require. Based on the preceding discussion, component requirements include the following:

Downloaded from https://www.everyspec.com



a. A well-defined behavior that is reflected through services offered the component.

b. Encapsulation of the data that is managed by the component.

c. A well-defined and durable interface through which services are obtained.

In OO design technology terminology, active components may be loosely thought of as large-scale executable objects designed to be easily replaceable as a unit in the context of a system. Especially beneficial is the situation when the interface is amenable to formal characterization, (e.g., by OMG modeling language specification or the UML). However, while the term ***object*** is often applied to such components, use of OO technology is not, in the strictest sense, a requirement for effective system partitioning.

3.2.1.2.1.3 System Partitioning for Components

Partitioning NWS functionality into components departs from earlier system development efforts. It tends to produce smaller, more functionally homogeneous, and loosely coupled program units rather than a few large non-homogenous shared-memory computer programs. These programs tend to be large or medium grain rather than fine grain, as might be found in data flow decomposition.

It is appropriate to consider design from both a functional and an OO perspective. Two points should be considered. First, a pure OO view is neither necessary nor desirable in defining components. In fact, components will likely take on characteristics of each. Second, it is entirely possible that real-time components with stressing performance requirements will be program units with OO interfaces rather than purely OO. Finally, even though many designs will be based on a traditional, structured approach at the "macro-" level, the OO principles of information hiding, inheritance, and polymorphism are often applicable at finer levels of granularity.

The terms ***functionally homogeneous*** and ***loosely coupled*** should be understood in general rather than precise terms. In this context, these terms imply a number of things about the structure and composition of the body of code that constitutes the component, as follows:

a. The boundaries of components result in clearly distinct entities separated by considerable functional distinctiveness and distance in a requirements sense. For example, there should be a distinct separation of User Interface (UI) components from non-UI components.

b. Components may rely on shared services provided by other components; in the example above, both data link processing and NWS control need access to track information. This in turn suggests yet another possible component boundary--in this case, one that provides service to multiple track-consuming components.

c. The operations and/or services provided by the component that other components rely on are logically closely interrelated and mutually supportive.

Downloaded from https://www.everyspec.com



Component boundaries are chosen to maximize internal binding strength while minimizing external binding strength. In this case, binding strength is related to the frequency, volume, and timeliness requirements of data access among code segments and is based upon component boundaries selected in accord with the binding strength criterion:

a. Intra-component data structures and operations are so closely related that direct visibility and tight coupling of processing and data access (e.g., via shared memory) is appropriate internally—the frequency, volume, or timeliness of data access or the synchronization of changes to the data would make a message-based solution impractical; at the same time, loose coupling (message passing and distributed object invocation) to all other components is practical in a performance sense.

b. Encapsulation, scoping, and visibility of externally referenced data structures are chosen so that an API-based non-shared memory, non-direct invocation style of system construction may be maintained without rendering component performance unsuitable and without exposing non-essential internal data to external visibility and access.

c. When senescence and bandwidth characteristics permit, tightly coupled (e.g., shared memory) access to data needed by multiple code segments is replaced by ***forward caching*** of data to consumers. This is done via an explicit multicast-like data distribution mechanism (i.e., distribution of data by a server-like entity in advance of its need by consuming clients). By this forward-caching mechanism, data references that were formerly accomplished by explicit access to memory shared are converted to local references within the component using the forward cached data.

d. Size of the component code base is a few hundred to many thousand lines of source code (i.e., ***medium grain***). Exceptions to the size guideline are twofold. The first case involves large, tightly coupled applications for which loosely coupled partition boundaries are not available. The second case involves small components, often elements of service libraries, e.g., math functions or domain specific object classes of common utility across applications.

e. Reuse is defined at the component level. Components may be service libraries, broadly applicable domain-specific classes, executable programs, or entire subsystems. There is no inherent size limit associated with reuse. Generally, making a component reusable is a business case and a programmatic decision as well as a technical decision. Reuse is often the result of a design decision by the developer of a particular subsystem in order to promote efficiency and competitiveness. In such cases, few if any intellectual property issues arise with reuse. This is not the case where the scope of reuse applies across multiple developers.

By the conventions used herein, components may be either active or passive. Thus, components may be processes (active), or they may be libraries (passive). Design considerations for active components include use of priorities, control of priority inversion, thread-safe synchronization and data access, and replication. Design considerations for passive components include re-entrance. In both cases, error handling should be considered.

Downloaded from https://www.everyspec.com



The most common view of an active component is that a component constitutes a separate process. However, a capability called ***collocation*** is available within some distributed object-computing models. With location transparency, objects may either be located on the same machine and within the same address space as other code segments, or they may be located remotely, accessible only via physical distributed computing mechanisms. Collocation allows the middleware to use more efficient means to access objects that are located on the same physical computer or within the same address space, even though the application invokes operations on those objects in the same manner as it would if they were located on a remote node. However, code built to depend on the efficiencies provided by collocation mechanisms may be less portable than non-collocation code; therefore, collocation is generally not recommended unless no other mechanism will meet application performance requirements.

Component technology such as Enterprise Java Beans (EJB) and the CORBA Component Model (CCM) may provide additional support for building reusable components and extensive services. However, EJB is only useful where Java is selected as the implementation language. The CCM has the advantage of providing component support in a multi-language environment.

3.2.1.2.2 Guidance

Distributed tactical computer programs generally consist of a set of network-resident components whose collective interactions provide the complete specified functionality of a required operational capability. Distributed components have available the underlying technology base specified throughout Section 3.2.2. Each member of a set of fully distributed software components should be capable of execution within a common unit of hardware. Exceptions to this rule may occur in a large embedded real-time system. Thus, some components may exhibit dependence on unique I/O or special computational resources, such as a group of closely coupled processors. Except as required because of limitations involving unique resource and/or connectivity requirements, components should make use of general-purpose OA computing and connectivity resources. Distributed components should exhibit the following characteristics:

a. Components should preserve NWS functionality and real-time performance.

b. Component functionality and interface definitions should be verified and documented via explicit and comprehensive use case analysis, as exemplified in the UML and associated design tools. Analysis should include real-time performance, latency and timing considerations using real-time UML extensions as appropriate.

c. Components should fall into one or more of several categories:

1. Compiled, linked, stand-alone computer programs

2. Library elements that can be loaded and instantiated by such linked components

Downloaded from https://www.everyspec.com



3. Nested components wherein simple components are incorporated into a more complex compound components

4. Reusable services and components

5. COTS packages

Compound components involving nesting may be created in a variety of ways, including linking of library services and instantiation of simple classes with inheritance of simple class properties.

d. Component boundaries should maximize intra-component cohesion, coupling, and inherent parallelism. In order to leverage the unit of computing concept (see Section 3.2.2.3) and within the constraints of required performance and internal cohesion requirements, encapsulated component functionality should be a small, manageable subset of the overall functionality of the particular tactical element from which it derives. As a rough guideline, simple components should be ***medium grain*** (i.e., sized on the order of a few hundred to a few thousand lines of source code). Where necessary due to the monolithic nature of certain processing algorithms, simple components may also be larger.

e. Partitioning of tactical application functionality should be based upon an OO and/or structured (control flow) decomposition method. This method not only minimizes data flow between components but also limits ripple effects into other components of internal component changes. Thus, it supports reuse and enhances maintainability.

f. Components should be as self-contained and functionally homogeneous as possible and should encapsulate closely related and logically coherent functionality and data.

g. Components should be designed for self-contained and independent operation under control of the host operating system. Components should be multi-threaded where necessary to preserve time and event scheduling properties.

h. Components should be designed to have low cohesion (i.e., a loosely coupled, message-passing, or distributed-objects interface) with other components in the system.

i. Components should be designed such that their reliance on data from other software components in the system is minimized. When data sharing among components is required, techniques such as distributed objects or message passing should be utilized to the greatest extent possible in order to retain a loosely coupled system.

j. Component interfaces should be well defined and durable as to content, function, and methods of implementation. Interface definition should be made at the logical level rather than the physical level using standards-based interface definition languages and automatic interface code generation tools. The various technologies that contribute to developing inter-component interfaces are discussed in Section 3.2.2.

Downloaded from https://www.everyspec.com



k. Component interfaces should be defined in an appropriate interface specification document or via an Interface Design Language (IDL). The API for this interface should be defined in an acceptable language or meta-language (e.g., Ada package specification, C or C++ header file, XML schema, Javadoc, Web Services Description Language [WSDL], or CORBA IDL).

l. Where efficiency considerations (e.g., data size, parsing time, etc.) permit, standards-based, self-defining interface methods may be employed. For example, XML might be used to provide a variable format status data to an operator for display, or it might be used as a common data representation mechanism for data transfers between various system components. Such a mechanism would be particularly useful in cases where the transferred data could dynamically assume variable formats. For example, a data item representing status information might sometimes be a simple boolean designating success or failure, but at other times it may contain several fields describing various parameter values relevant to the status. Use of XML would provide the flexibility of dynamic interpretation of the data according to its structure as delineated by XML tags.

m. Components should not use explicitly shared memory techniques to communicate data between components; however, they may use shared variables among multiple internal threads so long as proper data protection primitives are employed. This should not be construed to forbid use of shared memory between an application and the operating system or between application and co-resident support functions such as timekeeping, data extraction, etc. When shared memory use is absolutely unavoidable due to performance considerations, standard methods should be employed.

n. Explicit control flows between components should be minimized, especially where mechanisms for information accessibility (e.g., publish/subscribe) are available that do not require symmetric (i.e., two-way) binding of component logical identities. Such accessibility mechanisms promote server interfaces that may be shared by multiple clients rather than a pair-wise interface approach. In the pair-wise approach, a unique interface is built between every two components that should communicate, thus increasing cost and complexity. Such non-symmetrical information access mechanisms also promote component decoupling since the server need not know anything about the clients.

o. All OA components with time-critical performance and recovery requirements should be replicated for fault tolerance. Fault tolerance mechanisms are discussed in Section 3.2.1.9. Components with less critical recovery time requirements may be designed so that they can be restarted under RM control (see Section 3.2.1.8).

p. All components for which potential future performance requirements exceed single processor computing capacity or network bandwidth should be replicated for scalability. Scalable load sharing is discussed in Section 3.2.1.10.

q. Components subject to replication (i.e., for fault tolerance or scalability) should be designed so that several active instantiations (or replicas) of the component can execute simultaneously in physically different processors/locations.

Downloaded from https://www.everyspec.com



r. When multiple active instantiations of a component are configured for fault tolerance, the instantiations should provide a means for coordination of state information between the replicas and for deciding which copies fulfill defined roles such as primary, backup, etc.

s. When multiple active instantiations of a component are configured for load sharing, the RM capability (see Section 3.2.2.8) should redistribute load and instantiate or remove active replicas for purposes of achieving system load balancing, as well as fault tolerance. Therefore, load balancing application components should be designed in such a manner to facilitate RM execution of the requisite control action. Application replicas should implement and coordinate the load balancing strategy. RM will add and/or remove replicas according to that strategy. Further investigation is required to define specific mechanism.

t. When multiple components perform functionally related operations (e.g., similar processing functions for different sensors, weapons or warfare areas), an investigation should be conducted during component design to assess reuse possibilities. Alternatives to be considered include taking advantage of reuse possibilities inherent in the similarity of the components (both in the form of design pattern reuse as well as code reuse) as well as implementing the component as a common support service.

u. When multiple mature implementations of the CORBA Component model are available, then use of CCM should be considered.

v. Real-Time QoS metrics should be defined in terms of component performance requirements and multi-component execution paths determined by domain analysis. Real-Time QoS metrics and thresholds should be identified and quantified for each distributed tactical function hosted by OA. Real-Time QoS metrics should be utilized by the RM capability to dynamically balance the loading in the system as described in Section 3.2.2.8. Further investigation is required to define specific mechanisms.

w. Components should be robust with respect to exception conditions. Where appropriate, exceptions should be detected and handled internally to each component, preferably to the lowest level of invoking code that has sufficient information to properly handle the exception. When necessary, exceptions that cannot be handled locally should be defined as error conditions within the component's interface and conveyed to invoking code segments in other components.

x. The OA program manager will control the functional allocation of components in warfare systems. The intended level of commonality desired across multiple systems will determine the level of that control and approval.

#### 3.2.1.3 Portability

This section provides rationale and guidance for designing OA components to ensure portability of source code across networks, computers, operating systems, middleware, and other technology base components of the OA computing system.

Downloaded from https://www.everyspec.com



#### 3.2.1.3.1 Description

A major characteristic of today's computing industry is the fact that the technology base is changing and evolving rapidly. This potentially provides great benefit in terms of a steadily improving price/performance ratio. However, systems that are not designed to accommodate this rapid change of the technology base can be quite costly to maintain over the life cycle. Application source code not designed for portability should be modified, sometimes extensively, when it is ported to new networks, computers, operating systems, middleware, etc. In some cases, the changes can be so extensive that redesign of the system may be necessary.

Conversely, through proper use of standards and isolation layers that hide implementation details, it is possible to design components so that application source code can be ported across a wide variety of underlying computing technologies. Achieving this objective requires exercising sufficient design and implementation discipline to forego use of vendor- unique features. This is sometimes difficult since vendor-unique features may confer a modest or even substantial advantage in performance or in initial cost; however, the long-term cost of repeated use of non-portable source code to a succession of technology bases often eventually far exceeds any initial cost savings or performance gains.

Portability is not an easy goal to implement. Different types of portability are available, each resulting in a unique set of measures and metrics that is used to determine the portability effectiveness in the design. Portability types are as follows:

a. Product Line Portability. Vendor upgrades its process such as clock speed, additional memory, etc. but is nothing more than an extension of the vendor's product line in the same family (no changes to operating systems, compilers, linkers, debuggers, etc.).

b. Development Environment Portability. Vendor upgrades compilers, linkers, operating system, patches, tools, processes, IDEs, development frameworks, etc. With some new product releases, additional features and capabilities may be added, while current capabilities and features may be eliminated or deprecated.

c. Performance Portability. The performance in one computing and networking environment is the same (within thresholds that need to be defined).

d. Operating System Portability. Portability of applications between operating systems. As an example, the POSIX standard was designed to promote portability of applications across different operating system platforms. Software that is developed in close compliance with the POSIX specification can be ported with varying degrees of ease from one operating system environment to another.

e. Platform Portability. Portability of applications between different hardware platforms. This is both between a PowerPC from Vendor X to a PowerPC from Vendor Y and PowerPC from Vendor X or Y to Intel Pentium from Y or X. However, this portability is still in the basic platform style such as single-board computers (SBCs), i.e., SBC-to-SBC but not SBC-to-server.

Downloaded from https://www.everyspec.com



f. Computing Environment Portability. Portability of applications between different computing and networking environment. This involves porting the application from one type of environment to another style, e.g., SBC-to-server.

g. Language Portability. Portability of the application from one computer language to another language, e.g., via use of language neutral abstraction techniques such as Model Driven Architectures.

h. Architecture Portability. Port or convert the architecture style to a new architecture style.

i. Intellectual Portability. Portability of knowledge and training from one environment to another.

j. Language Platform Portability. Combination of operating system, computing environment, development environment, and platform portability. The Java language and Java Virtual Machine (JVM) were created to support this type of portability.

k. Complete Portability. Combination of the different portability types or all of the portability types.

A complete discussion of portability should address most if not all of these attributes. In many cases, metrics will involve thresholds for labor efforts, cost, schedule, and code changes to support the desired portability. Establishment of specific metrics for each of these is beyond the scope of this document and should be addressed early in an OA design by application developers.

3.2.1.3.2 Guidance

In accord with the objective of producing application components that can be ported quickly and affordably from one computing environment to another, applications should be built to be portable across a variety of standards-based platforms, networks, operating systems, and other technology components.

a. Application source code should require a minimum of compilation modifications due to changes in network configuration and technology, computers, operating systems, middleware, and other technology base components of the OA computing system.

b. In support of the above guidance, the changed computing environment components should adhere to the interface standards employed by the OA application software (see Section 3.2.2 for information on the OA computing environment; see Section 3.2.6 for guidance on standards selection).

c. There should be a minimum of dependence of application components on vendor-unique features. Where needed, tailoring to specific machine/operating system/vendor differences should be accomplished by isolation of changes in an adaptation layer. When

Downloaded from https://www.everyspec.com



possible, such mechanisms should utilize commonly available functions that can be emulated by other means (such as macros/stubs).

d. Portability may on occasion need to be sacrificed (e.g., via the use of vendor-unique features) to meet specific component performance requirements. The OA office will actively participate in program decisions regarding the level of required portability and the use of vendor unique features.

#### 3.2.1.4 Location Transparency

This section provides rationale and guidance for designing OA components to ensure location transparent allocation and operation.

##### 3.2.1.4.1 Description

A fundamental and recurring problem in developing and evolving large systems is the need for frequent change and growth. This means that task-to-resource bindings established during initial system design are likely to change over time—sometimes quite rapidly and extensively—as new requirements emerge and are incorporated. System designs that fix the task-to-resource bindings at the design stage are inherently difficult, expensive, and time-consuming to change.

Mechanisms exist to defer binding of application programs to system resources until system startup and, increasingly, at runtime. Whenever tactical performance requirements permit, use of these deferred binding mechanisms confers considerable flexibility in the design and maintenance of large, complex systems. Deferred binding permits incorporation of new computing resources to meet expanded requirements while preserving flexibility to modify system resource allocation, as well as to provide more flexible failure recovery schemes. These mechanisms sometimes add overhead, a fact that has weighed against their use in the past. However, the low cost and speed of modern processors and networks render this overhead affordable in all but the most stringent performance cases.

In accord with the objective of producing designs that permit or enable deferred bindings, applications should be built to be portable across a number of platforms, networks, and operating systems. They should take advantage of modern middleware and other technologies, including collocation facilities when necessary, so that they are location transparent (i.e., so that they communicate as logical entities rather than fixed-mapped entities located at predefined physical addresses). Included in the capabilities of such middleware products are name services that provide the translation from the logical addressing scheme of the middleware-based applications to the underlying physical addresses characteristic of the equipment base.

Location transparency has certain implications on system test and certification. In particular, it creates a situation rarely faced in the tactical community before, namely the fact that a particular functional capability may be achieved by multiple (identical) physical allocations. (In the more general case, even this physical isomorphism may be relaxed.) While these configurations are logically identical and physically comparable, they are not physically

Downloaded from https://www.everyspec.com



identical. The issue of acceptance testing and certification should be addressed in this context. Certification is discussed in Section 3.2.5.

3.2.1.4.2 Guidance

To achieve the objective of producing designs that permit or enable deferred bindings, the following guidelines are provided for component design:

a. Application components should be designed to be location transparent within OA network (i.e., there should be no application-level binding between components and physical addresses). This guidance should, at a minimum, apply to groups of compatible resource units (i.e., pools of resource that are effectively interchangeable from the perspective of component resource requirements).

b. Except where unique, performance-driven configuration requirements exist, addressing components within the OA implementation should be by means of logical names rather than physical locations. High-level middleware protocols (i.e., above the level of the Transmission Control Protocol/Internet Protocol (TCP/IP) and the Universal Datagram Protocol (UDP/IP) that provide logical naming services should be used for purposes of inter-component communications in all cases where performance requirements permit their use. These middleware protocols should provide a naming service. Recent combat system development experience across a wide selection of systems and developers has shown that system performance requirements are now achievable for most warfighting applications; exceptions generally lie in the area of signal processing, missile internal guidance algorithms, and other extremely high-performance domains.

c. Object collocation facilities should be used only where necessary as a means of increasing efficiency and performance.

d. Currently, complete separation of components from physical addresses is infeasible in most I/O device driver designs. To minimize the impact, one option is that the device driver should provide an interface to the system via a dedicated application component or components that present a named interface ***port*** or mechanism (e.g., Input/Output Processor [IOP] Gateway or storage-related nodes). Other implementations are possible and may be employed so long as location transparency is not compromised, subject to item b above.

3.2.1.5 Client-Server

This section provides rationale and guidance for designing OA components according to the widely used client-server design pattern.

3.2.1.5.1 Description

The client-server design pattern has gained widespread acceptance in commercial computing applications. Indeed, it has become almost ubiquitous because the model provides a very useful way of exporting services to a variety of users in a standardized manner. Servers, in

Downloaded from https://www.everyspec.com



particular, provide an excellent means of promoting reuse and standardization at the application level.

A server is a system component that supplies and controls access to a computational or data resource via a predefined interface. The resource controlled by a server may be a physical device, such as a display or external sensor; a persistent data store, such as a database; a state data repository; an algorithm; or a combination of these.

The interface provided by the server may include specification or guarantee of QoS parameters, such as client request priority or maximum response latency. A server exhibits the following characteristics:

a. A server and/or clients may consist of multiple cooperative or replicated processes.

b. Servers do not require any architectural or functional knowledge of the client application(s) that use it.

c. Clients require no functional knowledge about the server other than the server's API, nor do clients need to know the specific server replica or process with which they are communicating.

d. A server does not initiate interaction independently with its clients unless clients have registered for specific "*push*" services (i.e., server-initiated updates).

e. A client is any component that accesses the server via the defined API. Consistent with previous discussions, clients and servers may be physically collocated where use of such mechanisms is appropriate for performance. A server does not require knowledge of its clients in advance of receiving a request; thus the client should initiate any interaction between client and server. Similarly, clients do not need to know the internal workings of the server. Furthermore, since a server is not dependent upon the function or implementation of a client, it may provide services to multiple heterogeneous clients.

Because clients and servers may be mission-critical components and because they may have significant processing loads and/or throughput requirements, both clients and servers may have replication requirements in support of fault tolerance and/or scalability. Replication of any given client, for either reason, should require no modification of the predefined client-server interface. However, the interface can be designed to provide server support for client replication without negatively affecting the integrity of the client-server relationship. Examples of interface design choices that would support client replication, while still maintaining an appropriate decoupling between client and server include:

a. A registration mechanism in which the client specifies a publication group for the server to use in its distribution of data to the client. All replicas of the client can then benefit from the server's data distribution without each replica having to register separately.

Downloaded from https://www.everyspec.com



b. An interface by which a new replica can request current state data from the server. By anticipating the requirements of a replicated client, an appropriately designed server interface can work equally well for replicated and non-replicated clients.

c. Interface support that allows a client to register for only a selected portion of data (e.g., only certain tracks) for the purpose of load sharing with its peers.

The roles of client and server are not mutually exclusive. These designations refer to the relationship of two processes with regard to a set of interactions, e.g., access to a database. It is possible, and even likely, that a given process can be a client with respect to one set of interactions, and a server with respect to another. For example, a server that distributes kinematic sensor data may also be a client of a server that distributes non-kinematic sensor data, such as an identification server. In addition, both servers may be clients of a data-recording server.

Use of client-server architecture concepts allows for the development of both subsystem-wide and system-wide services. Typical services include but are not limited to data distribution; access to physical resources, such as sensors, actuators, or external devices; or state data protection and mediation. They may provide services required by the entire system, such as time synchronization or instrumentation, or that may be required by only a subset of the applications. Regardless, the availability of client-independent services greatly enhances the ease with which new elements and applications can integrate with the existing system.

A schematic of the interactions between clients and servers is shown in Figure 3-2. Three common models of client-server interaction are defined:

a. Synchronous: request-reply

b. Asynchronous: one-way

c. Asynchronous Response: client registration

Downloaded from https://www.everyspec.com



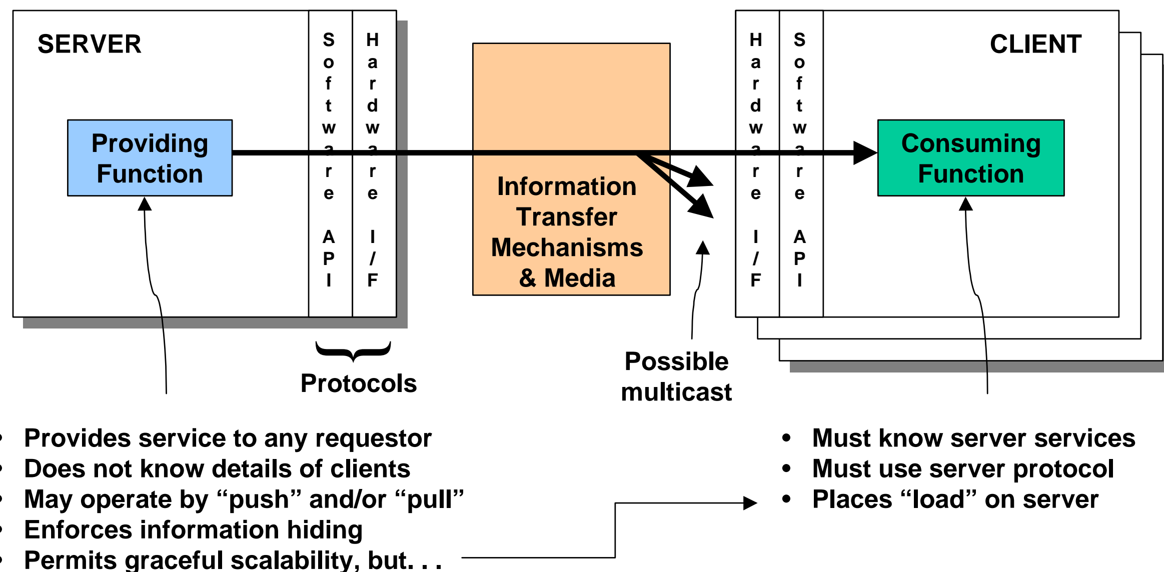


**Figure 3-2. Client-Server Model**

The first model, the synchronous model, is also known as the pull or request-reply model. This model requires that the server take a specific action in response to the client's request. A client will issue a message to a server requesting that a service be performed. For example, the client might request the display of data at an operator console or the initiation of some algorithmic processing. When the server completes action on the request, it will respond to the client with the results of the action, i.e., with an acknowledgment of completion or the results of the computation.

In some cases, a client may make a request to the server and no response is required. This type of interaction is the second common model, sometimes called the asynchronous one-way model, when the results of an asynchronous request to a server are implicit (e.g., a change of state data in the server). This model of client-server may be implemented using a multicast transport mechanism, effectively creating a diffusion of data, such as status data, to any client listening to data across that multicast channel.

The third common model of client-server interaction is the client registration or asynchronous response model, often referred to as the push model. This model requires that clients register for services that are to be delivered at a later point in time. For example, a client may register with a track data server for the update of track data at some client-specified frequency. Alternatively, a client may request notification upon a change in state of some resource. The server retains information about the specific requests of the registered clients as well as sufficient logical addressing information to allow the server to monitor the status of the clients and provide the information as appropriate. This model may also used in conjunction with a multicast transport mechanism, where the logical address is a multicast address. This

Downloaded from https://www.everyspec.com



allows a single client to register for information that is then delivered to any client with similar data requirements that is aware of and listening for data across that multicast channel.

Peer-to-peer relationships are typically used between process replicas or between related clients of a common server. Peers communicate to notify one another of state changing events or to coordinate a set of activities that culminate in a change in state data.

Servers control development and maintenance cost and complexity by providing software that serves many clients instead of providing separate and unique software functions for different consumers. Thus, the client-server model supports the concept of multi-purpose software. If unique customization were permitted, the client-server design pattern would not be helpful.

However, a critical system performance impact exists that should be accounted for during server design. By their very nature, clients place loads on the servers they access. Thus, servers should be designed carefully to ensure required system performance and efficiency. For this reason, design of client-server architectures, particularly for real-time applications, should be approached carefully. Furthermore, care should be taken that "utilization creep" does not occur; i.e., a growth in the number of calls on a server pushes the server towards the upper limits of its planned load capacity without sounding any alarms. In order to mitigate the impact of this client-imposed load, it is appropriate to consider a scalable design pattern for servers (see Section 3.2.1.10).

3.2.1.5.2 Guidance

The client-server model has been shown to be beneficial in implementing flexible and scalable systems. As such, it should incorporate the following guidance as appropriate:

a. To the maximum extent possible, servers should be designed such that they are independent of the implementation of their clients. Servers should be capable of providing services to multiple heterogeneous clients.

b. Clients and servers may have replication requirements in support of fault tolerance and/or scalability. Replication of any given client, for either reason, should require no modification of the predefined client-server interface. However, the client-server interface can be designed to provide server support for client replication without negatively affecting the integrity of the client-server relationship.

c. For clients and servers where replication is involved, the client-server API should be designed such that the clients are not directly dependent upon the replication mechanisms of the server. Additionally, the API should support or at least not interfere with potential replication of both the servers and the clients.

d. As with all component interfaces (see Section 3.2.1.1), the interface provided to the client by the server should be well defined in an appropriate interface specification document or via an IDL. The API for this interface should be defined in an acceptable language or meta-language (e.g., Ada package specification, C or C++ header file, XML schema, Javadoc, WSDL,

Downloaded from https://www.everyspec.com



or CORBA IDL). It should provide access to the server via a procedural or object class interface. When the API and server are separately maintained, documentation should be required of both the API and the public interface between the API and the server.

e. For protocol-based interfaces, (e.g., File Transfer Protocol [FTP] and Hyper Text Transfer Protocol [HTTP]), a rigorous specification of the protocol version, appropriate ports, expected inputs and outputs, and options (as makes sense for that protocol) should be provided in lieu of an API.

f. Servers should detect erroneous invocations and invocations for which sequence dependencies exist, and they should export exception conditions to invoking clients via the component's interface. Such erroneous invocations should not cause the component to operate incorrectly. Interface specifications should indicate the meaning of exceptions and the conditions under which they arise (e.g., the specification should define what happens if **open()** is called twice without an intervening **close()** call).

g. Components should adopt internal design safeguards (e.g., finite state machine structure) to eliminate the possibility that erroneous inputs and/or call sequences can result in erroneous operation, hung conditions, and crashes of the component.

h. Where appropriate, the server interface specification should include both explicit and implicit QoS parameters (see discussion in Section 3.2.1.11.1.2). An example of an explicit QoS parameter would be a mechanism for clients to specify response latency with their request. An example of an implicit QoS parameter would be a maximum response time associated with a particular type of request, regardless of the client initiating the action.

### 3.2.1.6 Data Distribution

This section provides rationale and guidance for ensuring the efficient flow or distribution of high-volume continuously refreshed data among OA components. Examples include distribution of track updates and navigation data to OA components. Distribution management techniques include use of a track data server to provide track distribution to registered track consumer client components, use of periodically refreshed push distribution (forward caching) of continuously refreshed data, and use of asynchronous processes with timed/fixed packet I/O buffering.

#### 3.2.1.6.1 Description

The nature of data flow within a large, complex system often dictates that certain components are distributors of data that is continuously refreshed, frequently in accord with a periodically scheduled thread that guarantees that no data are ever more stale than a defined value. Such components are excellent candidates for use of the publish/subscribe pattern.

The publish/subscribe pattern allows data consumers to express their interest in certain events or data, frequently referred to as topics. Data producers, or ***publishers***, publish data matching one or more of the topics. A logical software bus, often in the form of a distributed

Downloaded from https://www.everyspec.com



event manager, coordinates delivery of the data to the consumers based on the topics. This software bus allows the data producers and consumers to be logically decoupled. Publishers do not need any awareness of the subscribers that are registered for their data. Likewise, subscribers need only be aware of the availability of the data in the form of the topics that are available. They do not need to be aware of the data sources. Many commercial middleware products support the use of the publish/subscribe design pattern for distributed systems.

Components that participate in the distribution of mission-critical data, either as data producers or consumers, have the potential to affect the performance of other related data producers and consumers. In order that they perform as effectively and efficiently as possible, certain design characteristics should be implemented.

Many tightly timed real-time systems resort to a highly rigid synchronous periodic design in order to achieve efficiency. This is a good design for pure hard real-time systems that cannot meet timing requirements any other way. However, it is also an inflexible one, particularly when applied in a distributed processing environment. If timing and performance considerations permit, a far more flexible approach is available; i.e., one based on asynchronous, loosely coupled processes.

In such cases, workload flow variations are generally “cushioned” by use of message queues or buffers to smooth out backlogs and variations in arrival rates. This approach can be used in conjunction with the synchronous, hard real-time approach, thus fostering co-existence of hard and soft real-time processing within the same overall system.

Managing the flow of frequently updated information in such a real-time system is a critical design characteristic, particularly if the flow is high volume, as it so often is where sensor data distribution is part of the overall system design. This problem is compounded if the updates are irregular, as they are in the case of some sensor systems. Such variations are attributable to the fact that data flow is a function of what is in the environment in which the sensor operates.

In real-time systems, requirements generally tend to focus on latency considerations, despite the fact that–in view of the high volumes associated with these data streams–efficient use of available bandwidth is also an important consideration. The two simplest strategies for managing this data flow are (1) to send every update, which minimizes latency, or (2) to fill a fixed-size buffer prior to transmission, which maximizes channel throughput.

However, neither strategy provides a complete solution in many real-time system applications. In particular, the delivery time for the fixed-buffer method is potentially unbounded--an unacceptable situation for real-time performance. A good compromise is to employ both techniques simultaneously.

In the combat system, track distribution is a good example of the data distribution problem due to many consumers of track data. In previous system designs, processed track information (i.e., correlated and filtered data) was provided to consumers via custom-designed interfaces, with each interface being uniquely tailored to the requirements of the receiving

Downloaded from https://www.everyspec.com



element. This provides maximum functional capability, but it is also the most costly approach, and the one least susceptible to multi-purpose use.

Providing push-based distribution of the entire track file, appropriately time-tagged for local extrapolation (see Section 3.2.2.13 for guidance on system-wide time synchronization) to all track consumers, ensures that track information is always available to consumers when they need it. In effect, data is *forward cached* with track consumers against the certain eventuality that it will be needed. The volume of data and the frequency with which it should be distributed is well within the capacity of modern switch-based networks.

Benefits of using a push-distribution mechanism for track data include reduced processing in the track data server, reducing the possibility of widespread contention and performance bottlenecks. There are also the benefits of increased ease of integration of new functionality:

a. Ability to leverage common code to perform local caching and retrieval of track data, resulting in a reduction of application complexity.

b. Minimal performance and bandwidth impact with additional track data consumers if multicast is leveraged.

3.2.1.6.2 Guidance

The design guidance for OA components that perform distribution of continuously refreshed data is as follows:

a. The publish/subscribe pattern should be used for distribution of continuously refreshed data. While this would seem to drive up network bandwidth utilization, techniques such as multicast exist to ameliorate this possibility.

b. A common API and library of services should be provided to data consumers for receipt of events, data updates, and retrieval of locally cached data. This API may take the form of application library component APIs, standards-based product APIs (e.g., middleware), or a combination, as appropriate.

c. Data producers should provide a defined class of service with respect not only to functionality but also to performance. Thus data distribution periodicity, senescence, and maximum load should be defined as a component requirement for the data producer. The periodicity and other distribution characteristics of the data producers' transmissions should meet consumer requirements. Data producers should be designed to provide a limited but distinct range of distribution QoS options in order to service varying data consumer timeliness requirements.

d. Where throughput and latency requirements require it, continuously refreshed data flows among tactical components may be implemented using timed buffer transmissions. In this design, the buffer size is often fixed, but its transmission is tied to a periodic cycle that satisfies

Downloaded from https://www.everyspec.com



timeliness requirements associated with the data flow. If the buffer fills up before the period expires, it is transmitted as soon as it is full. In this way, efficient throughput and channel use is maintained. On the other hand, if the buffer is not full when the period expires, the partial buffer is transmitted at that time. This ensures that a defined worst-case latency is not exceeded.

e. To ensure efficient data producer operation, and when possible given performance requirements, underlying communication of data from data producers to the data consumers should be performed by middleware protocols that provide IP multicast functionality. This ensures efficient server operation.

f. Where applicable, data producers should be capable of guaranteeing first-in/first-out (FIFO) order of data flow to each data consumer. However, total ordering of the producer's data flow across consumers is not necessarily required. That is, unless specific applications have a total ordering requirement, completion of a particular update to all consumers is not necessary before some consumers begin receiving other updates.

g. Delays in consumption of distributed data on the part of a data consumer should not result in a system-wide performance-degrading backup of data distributed from the producer to that consumer or any other consumer, nor should the producer's operation be impacted. Where required, data consumers should be designed such that data may be discarded during overload conditions. This latter requirement ensures that the ordering requirement stated above does not force consumers to process stale data. The circumstances under which data consumers may discard data and the criteria for discarding the data should be clearly defined in an appropriate interface specification.

h. Mission-critical data producers should be fault tolerant. A number of useable fault-tolerance strategies are defined as architecture patterns in Section 3.2.1.9.

i. Where substantial growth is anticipated in the load a data producer is likely to experience over its life cycle, the producer should be designed using a scalable architecture pattern (see Section 3.2.1.10).

j. Where applicable, consider utilizing intelligent cache techniques and patterns, such as forward caching of data, in order to reduce the path traversed and the latency incurred by a component data request and the resulting response.

#### 3.2.1.7 State Data Coherency

This section provides rationale and guidance for ensuring that OA components that manage composite transient state data are designed to provide correct operation under all possible state transition situations. Included are complex state variables resulting from the run-time interaction of many components. Maintaining state data coherency is a part of the overall requirement to preserve data integrity in mission-critical and safety-critical systems. This section refers to the maintenance of transient state data rather than persistent state data. A more stringent degree of coherency of this transient state is required in some cases to achieve application fault tolerance beyond that required to achieve correct execution.

Downloaded from https://www.everyspec.com



#### 3.2.1.7.1 Description

Certain applications are inherently state intensive. Persistent state is generally maintained on persistent media in the form of databases and files. However, some applications need to manage transient state—the state that is critical to correct performance but is changing at a sufficiently frequent rate or with time-critical access needs that make storage on persistent media impractical. For such applications, the total set of transient state information constitutes a composite state vector comprised of state information contributed by many other components. Maintenance of this composite state information is often vitally important for correct system operation. Furthermore, maintenance and interchange of transient state information among replicated components for the purpose of failure recovery is a major consideration in designing state-intensive fault-tolerant components. For this reason, special consideration should be given to the design of components to ensure state data coherency.

Mechanisms by which state data coherency may be maintained include atomic multicast communication transactions with one or more components, guaranteed delivery of message traffic in a specific order and synchronized exchange of transient state data among participating components. Atomicity of multicast communications means that for each multicast, either all recipients receive the message or none receive it – and that the sender knows who received what. In the case of ordered messaging, various ordering properties are frequently defined (see Section 3.2.2.7.1.3).

Use of priorities can, at times, conflict with maintaining transient state consistency. Priorities should not be used to achieve message ordering, since the scheduling mechanisms between platforms vary and can result in different effects for applications running on different operating systems. Also, priorities can disrupt other ordering mechanisms, such as ordered messaging, potentially causing the order in which messages are handled to be unpredictable.

#### 3.2.1.7.2 Guidance

The design guidance for OA components that maintain composite state data is as follows:

a. Where application performance requirements permit, components that manage transient composite state data should utilize middleware and reusable services to enforce communications atomicity, message ordering and state data exchange. Impact on application source code should be minimized and limited to those steps for which appropriate middleware and service library functions are not available.

b. Components with transient composite state requirements should not utilize priorities to achieve ordering between replicas. To the degree possible, components that maintain composite state should not depend on use of priorities for correct operation. Where essential to meet performance requirements, care should be taken when using priorities in such components so that affects on portability and interoperability are minimized.

c. Components whose primary task is transient state data management should ensure by component design that computations associated with state data coherency maintenance do not

Downloaded from https://www.everyspec.com



adversely impact performance and timing characteristics of other system components. Maintenance of composite state arising from interactions with other components places a strong obligation on the designer to ensure responsiveness of the state data-maintaining component. Further investigation is required to define specific mechanisms.

d. Unlike data distribution servers (see Section 3.2.1.6), which have a requirement for non-coupled operation with their clients, state data intensive components should operate in synchrony with other components to which it interfaces to maintain its transient composite state.

e. Transient state data maintenance should be compatible with the component's fault tolerance design pattern (see Section 3.2.1.9).

#### 3.2.1.8 Computational Flow

This section provides rationale and guidance for ensuring those OA components that perform periodic or on-demand computations are designed to provide efficient and effective operation.

##### 3.2.1.8.1 Description

Many of the computations in large, complex systems are part of an overall data flow process. In this model, information arrives at a component, is processed, and is then forwarded to another component for additional processing. Such components are sometimes referred to as pipes or filters. The concatenation of several such components, coupled by interfaces and by component-spanning timelines requirements, constitutes a data flow path through the system.

Data flow paths can include both periodic processing and event-based processing. They may also be quasi-continuous or intermittent.

For periodic data flow paths, the nature of the processing is such that large quantities of data are processed on some predefined periodic frequency. In some instances, outputs may occur only infrequently (e.g., when certain conditions are detected in the incoming data). In other situations, periodic outputs of processed data may occur. Periodic data flow processes should also execute in a coordinated fashion with external event reaction time paths. Since event-based processing should execute with very low latency, periodic data flow processes may need to be preempted, requiring that special care be taken in their design.

Event-based processing is defined as processing that takes place in response to some pre-defined external event or condition. Sometimes the resultant processing is limited to either a discrete response to the event or to updating internal state information based upon the event. In other cases, a quasi-continuous sequence of processing may be initiated. In the quasi-continuous cases, a computational data flow may come into existence for a period, after which it returns to a quiescent state.

Downloaded from https://www.everyspec.com



#### 3.2.1.8.2 Guidance

The design guidance for OA components that perform computations as a part of an overall system data flow is as follows:

a. Components that are part of a computational data flow path should, where necessary, utilize priorities to ensure that reactive threads of operation requiring minimum latency response are able to preempt routine threads to gain control of the CPU.

b. For real-time components, where routine computational threads (whether periodic scheduled or externally stimulated) are coupled to physics-based system requirements, thread priorities and careful system design should be utilized to ensure that processing requirements and latencies are achieved.

c. When quasi-continuous processing requirements may occur, the system designer should ensure that adequate reserve computational capacity is designed in to ensure proper performance of both continuous and quasi-continuous operations. Use of an RM capability to ensure that component computing capacity requirements are met is discussed in Section 3.2.2.8.

d. To the maximum extent possible, computational data flow components should be constructed to minimize retained state information. Unless otherwise infeasible, state data maintenance for such components should be limited to state information that is needed to sustain fault tolerance and failure recovery operations.

e. For components that should maintain state data, every effort should be exerted to design such components so that state data updates consist of a simple update operation involving only the previous state operated on by new data. Interactions with multiple other components in the maintenance of state data should be avoided to the maximum extent possible. When these conditions cannot be met, then state data coherency preserving patterns are likely to be more appropriate (see Section 3.2.1.7). Further investigation is required to define specific mechanisms.

f. Computational data flow components for which computational demand is likely to exceed the capacity of a single CPU—either in bulk computational capacity or in required minimal computational senescence—should be designed to exercise computational load management among a number of components (see Section 3.2.1.10). These components may be scalable peer-clients, pipelined components, partitioned functional capabilities, etc.

### 3.2.1.9 Fault Tolerance

This section provides rationale and requirements for ensuring that OA components requiring continuous availability are designed to provide a fault-tolerant capability in accord with a defined set of fault tolerance models (e.g., active, passive, and selected hybrids).

Downloaded from https://www.everyspec.com



3.2.1.9.1 Description

Failure of a mission-critical application process means that the system can no longer carry out an assigned function. When rapid recovery is required, replication of mission-critical application processes can increase the tolerance of the system to such faults. This is known as replication for fault tolerance. When long recovery time specifications exist, replication for fault tolerance may not be necessary. In such cases, the system RM function may simply restart the failed application component to regain the functionality that was lost as a result of the failure.

In addition to replication for fault tolerance, components may also be replicated for load sharing (see Section 3.2.1.10). In general, load-sharing components also contain provisions for readjustment of loads and continuation of operations following faults.

Frequently, mission-critical processes retain memory-resident critical-system state data. As such, in the event that a replica fails, maintaining the integrity of that state data is a significant consideration in the design of an application that is replicated for fault tolerance.

Middleware plays a critical role in the implementation of any fault-tolerant component, especially group communication middleware. Group communication middleware, including communications endpoint considerations, is discussed in Section 3.2.2.7.1.3.

Several common models used to implement fault-tolerant replication solutions. Six models are illustrated conceptually in Figure 3-3, of which five are described in more detail in the following subsections: Active Replication, Passive Replication, Primary/Shadow Replication, N-Version Replication, and Check-pointing. The load-sharing peer-client model is discussed in Section 3.2.1.10.1.2. The items that are checked in Figure 3-3 (Load Sharing, Primary/Shadow, Check-Pointing, and Active Replication) represent fault tolerance models examined in the High-Performance Distributed Computing (HiPer-D) risk reduction program.

Downloaded from https://www.everyspec.com



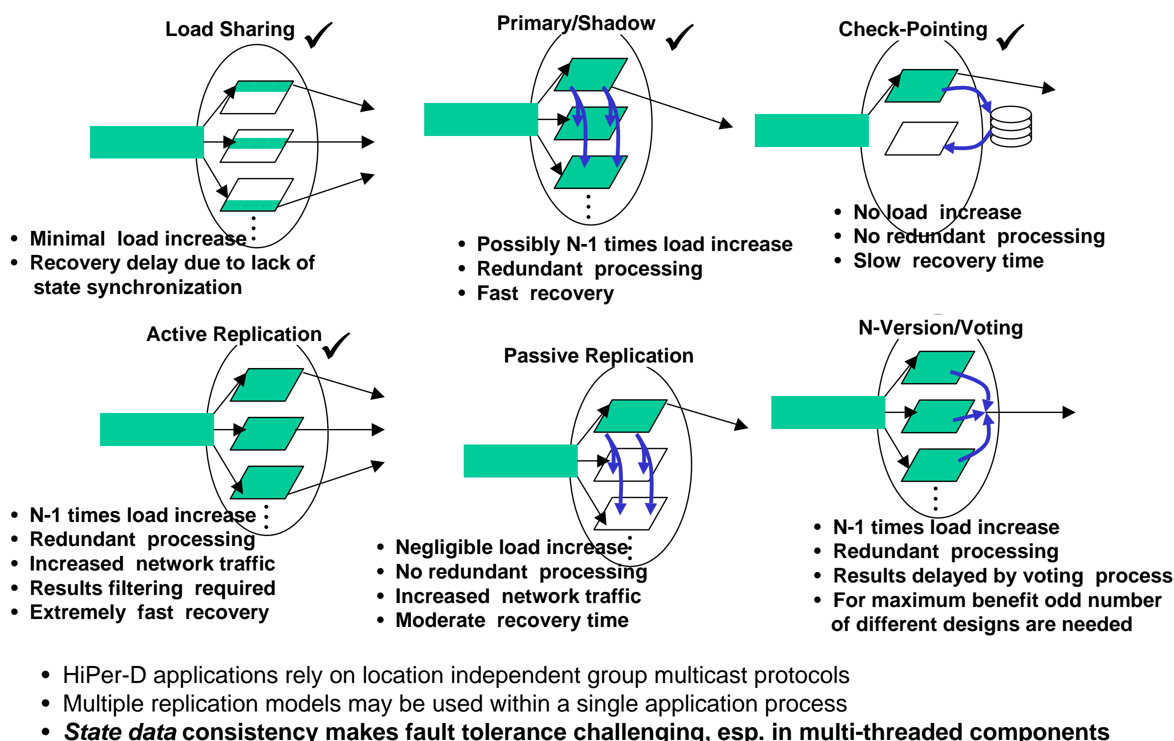


**Figure 3-3. Fault Tolerance Models**

### 3.2.1.9.1.1 Active Replication

When an application process has deterministic inputs and outputs, and deterministic behavior based on those inputs and outputs, active replication is a valid model for achieving fault tolerance. Active replication involves executing multiple identical copies of the same replicable process. Each replica receives all inputs, performs all processing, and provides responses. Receivers of responses from the replicas are responsible for filtering out duplicate data contained in the responses. The active replication model relies on deterministic input and behaviors for the replicas to remain in a consistent state. Generally, state synchronization between replicas is not performed except during replica initiation.

Two significant limitations are associated with this model. First, very few applications are completely deterministic. Sources of non-determinism may include very complex processing associated with reliable and atomic multicast, multi-source data inputs, memory management, thread scheduling, distributed time synchronization, time-based computations, and error handling. Group communications tools (discussed in detail in Section 3.2.2.7.1.3) can be very useful insofar that they can provide atomic, totally ordered multicast and ordered group membership change notifications. These attributes can reduce or eliminate potential non-determinism in the I/O streams to the replicas; however, such tools do not control other sources of non-determinism.

Downloaded from https://www.everyspec.com



The second limitation with active replication is its inability to handle Byzantine failures. Byzantine failures include unusual failure condition with a wide variety of causes, including data corruption, misbehaving applications, or even maliciously introduced errors. This class of failures is extremely difficult to manage with any model other than N-version replication, discussed in Section 3.2.1.9.1.4.

Single-threaded applications that perform non-time dependent calculations and applications that perform database retrievals from non-writable sources may be able to utilize an active replication model; however, its applicability is limited in the complex multi-threaded real-time systems domain.

3.2.1.9.1.2 Passive Replication

Passive replication is an appropriate replication strategy when the behavior of a process is non-deterministic based on its inputs and/or when processing resources are constrained. As in active replication, all passive replicas receive all inputs. Only the primary replica actually processes inputs and provides responses. Either on a periodic or an event basis, the primary replica transfers relevant application state data to the other replicas. This state data transfer is called checkpoint data. The other replicas, referred to as the secondary or passive replicas, queue the inputs until checkpoint data is received from the primary. The replicas use the checkpoint data from the primary to update state data as required, and the queued inputs associated with that checkpoint data are deleted. When a primary replica failure occurs, one of the passive replicas assumes primary responsibility and begins processing the inputs that were queued since the last checkpoint was received.

This method of replication requires associated applications be capable of handling duplicate data from the replicas. Recovery time may be long if significant processing is required for the queued inputs. To some extent this is controllable. Recovery time is determined by the relationship between queue growth and checkpoint interval. However, this model tolerates non-deterministic responses to input better than other replication methods.

Clearly, successful implementation of this replication model requires that associations be established between the inputs processed by the primary replica and the checkpoint data. By ensuring totally ordered message inputs to all replicas (both passive and active), group communications tools can play a significant role in establishing these associations.

3.2.1.9.1.3 Primary/Shadow Replication

A significant number of mission-critical applications do not meet the determinism criteria to use an active replication model, yet they have real-time requirements that cannot be fulfilled through passive replication. For these applications, a primary/shadow replication model may be more appropriate.

Although replicas processing the same inputs and outputs may arrive at different solutions, the primary/shadow model assumes that both solutions are potentially correct (in the absence of various subtle hardware and software errors). Multiple copies of a process are

Downloaded from https://www.everyspec.com



executed simultaneously, with one designated the primary replica and the remainder designated shadow replicas. All replicas receive and process all inputs with the primary replica providing control over the order in which the inputs are processed. When the primary arrives at either an intermediate or final solution, it provides that solution to the shadow replicas. The shadow replicas update their state with the primary replica's solution and do not issue independent responses, thus maintaining consistency with the primary replica. If the primary replica should fail, one of the shadow replicas assumes the role of the primary at the point in the I/O flow at which the failure occurred.

This replication model is based upon the following assumptions:

a. Failures are ordered with respect to message flow.

b. Messages will be issued via atomic multicast.

c. Messages will be processed in FIFO order. Since the primary replica controls the order in which the inputs are processed and does not require identical responses from the replicas, multi-threaded applications can be handled, and totally ordered messages are not required.

A detailed understanding of potential sources of non-determinism in the application is required to use this replication model. Since these sources vary with the specific application, primary/shadow replication cannot be implemented without awareness of the synchronization of the application state between the replicas.

#### 3.2.1.9.1.4 N-Version Replication

For some applications, correctness and/or uninterrupted (seamless) availability are critical. Safety-critical systems have unique correctness requirements. One method of providing greater assurance of algorithmic correctness is obtained through utilizing multiple implementations of critical algorithms. In N-version replication, N implementations of a process are executed, all receiving inputs and generating solutions. Through some arbitration, such as voting, a response is chosen, and potentially misbehaving processes are identified and excluded.

This method of replication shares some similarities with active replication. Care should be taken to exclude or accommodate sources of non-determinism, such as time dependencies. Some debate exists as to whether N-version replication provides significant benefit with respect to software/algorithmic correctness, since the source of many software and algorithm errors lies in the specification of the algorithms. Additionally, many software errors also result from incorrect handling of boundary and error conditions; thus they are more likely to occur in multiple implementations. Due to its high cost, use of N-version replication is rare.

#### 3.2.1.9.1.5 Check-Pointing

Check-pointing supports fault tolerance through the replication of data rather than replication of components. An application that implements a check-pointing pattern writes a

Downloaded from https://www.everyspec.com



snapshot of the state of an executing application, either to some form of persistent storage such as a disk or to another application that serves as a repository for the state. An application may write state data periodically or each time the state data is changed. If the application crashes, it can resume execution from the last checkpoint recorded by the original application.

This form of replication does not generally provide the precision of state retention that is potentially offered by other forms of fault tolerance. The recovery time can be slower, as it involves both restarting an application instance and recovering the state data from its storage medium. One advantage is that it affords some resilience to transient data errors that is not available from any other fault tolerance patterns. However, its primary advantage is the simplicity with which it can be understood and implemented.

3.2.1.9.1.6 Client-Server Interactions

Examples of how a client-server interface can support client replication include:

a. The client-server interface may be designed to allow the clients to readily subdivide data distribution among multiple peers, simplifying the ability to implement scalable peer-clients.

b. The client-server interface may allow the clients to specify a multicast address for data distribution or for request results, supporting replication techniques that provide full availability by instantiating duplicate running copies.

c. The client-server interface may allow a state transfer to occur when a new member subscribes, simplifying the initiation of new client replicates.

3.2.1.9.2 Guidance

The system should be resistant to faults in processors, network components, system software, and application computer programs. Component replication should be the primary mechanism for achieving fault tolerance. The guidance is as follows:

a. All OA time- and mission-critical components should be replicated for fault tolerance. In this context, time critical is defined as components for which reloading from a backup media is not adequately responsive. Components should be designed and implemented in accord with the appropriate replication strategy and fault tolerance design pattern.

b. OA components that are not time critical may be designed to be restartable under the control of the system RM capability. For such implementations, checkpointing may be a viable state data restoration strategy. Other strategies for fault tolerance, including those applicable to time-critical components, may also be used.

c. Under normal conditions, each load-sharing replicated component will actively share the processing load for the particular function. A minimum of two instantiations (or

Downloaded from https://www.everyspec.com



replicates) of any process on a mission critical path should be executing to support fault tolerance.

d. Metrics for assessing fault tolerance should be defined and performance thresholds should be established. The metrics will be used by the RM capability (see Section 3.2.2.8) to manage resources. System degradation should be measured in terms of task completion, timeliness and data integrity at a minimum. Further investigation is required to define specific mechanisms.

e. Detection of a fault in a processor, network component, or software component that results in loss of the component's capability (e.g., a fault that is not masked by network redundancy) should result in automatic shifting of the component's processing load to a replicated component. This may be a shifting of load to already active instantiations or may require that additional instantiations be replicated and activated to facilitate load redistribution. Detection and recovery times should be equal to or better than existing NWS requirements.

f. Each tactical component process should be independent of other components with regard to fault tolerance to the degree practicable. A failure in one process and the subsequent recovery actions associated with restoration of the capability should not require that other tactical components be replaced in unison except in limited situations when dependencies have been determined to be unavoidable. Each non-failed component should respond properly as the failed process recovers. Load sharing provisions are discussed in Section 3.2.1.10.

g. Multiple simultaneous process failures (e.g., loss of a cabinet containing multiple processors) should be treated as a series of independent failures. The failure management capability should queue all pending recovery actions and order them to ensure that, based upon recovery dependencies, multiple recoveries of the same component do not occur.

h. The fault-tolerant implementation should be designed to execute in a unified fashion across all OACE components. That is, for application components utilizing multiple OACE technologies, the varying characteristics of OACE shall be considered in component design. The use of fault tolerance features from multiple OACE technologies should not adversely impact application component fault tolerance performance with respect to fault tolerance metrics.

i. Due to its high cost, use of N-version replication should only be used when absolutely necessary.

#### 3.2.1.10 Scalability

This section provides rationale and guidance for ensuring that OA components with high-volume and large, dynamic ranges of computational demand are designed for scalability.

Downloaded from https://www.everyspec.com



3.2.1.10.1 Description

The term ***scalability*** has multiple meanings. In summary, these meanings of scalability are as follows:

a. Resource scalability

b. Functional scalability

c. Load scalability of load sharing

The first form of scalability denotes the ability of the system's design and implementation to accommodate incorporation of additional computing resources. Hence, networks promote scalability by making it easy to add additional computing resources.

The second type of scalability is the ability to add new functional capabilities without disrupting existing functional capabilities. The client-server model promotes functional scalability by allowing new client functions to access existing servers without disrupting other clients already using the server. However, adding new clients can potentially increase the need for increased load sharing for the server.

Finally, application components may themselves be made scalable to varying processing and I/O loads via architectural design patterns and load-sharing mechanisms. It is this last form of scalability that is discussed in this section. Two methods for implementing scalability are discussed: functional partitioning and replication.

3.2.1.10.1.1 Load Scalability by Functional Partitioning

Load scalability may be implemented by sufficiently partitioning a component's processing algorithms or application tasks so that the resulting executable processes can be spread across a number of processors and nodes. The partitions should be selected carefully so that the system can withstand, with a reasonable margin of safety, the maximum anticipated load. Common partitioning patterns include pipelining and paralleling. These patterns are shown in Figure 3-4. In the upper part of the figure, the boxes (p1, p2, etc.) represent pipelined components. Each process processes its inputs and passes outputs on to the next component in the pipeline. In the lower part of the figure, each of the parallel functions (f1, f2, f3) processes data of a particular type - different from data processed by the other functions.

Downloaded from https://www.everyspec.com



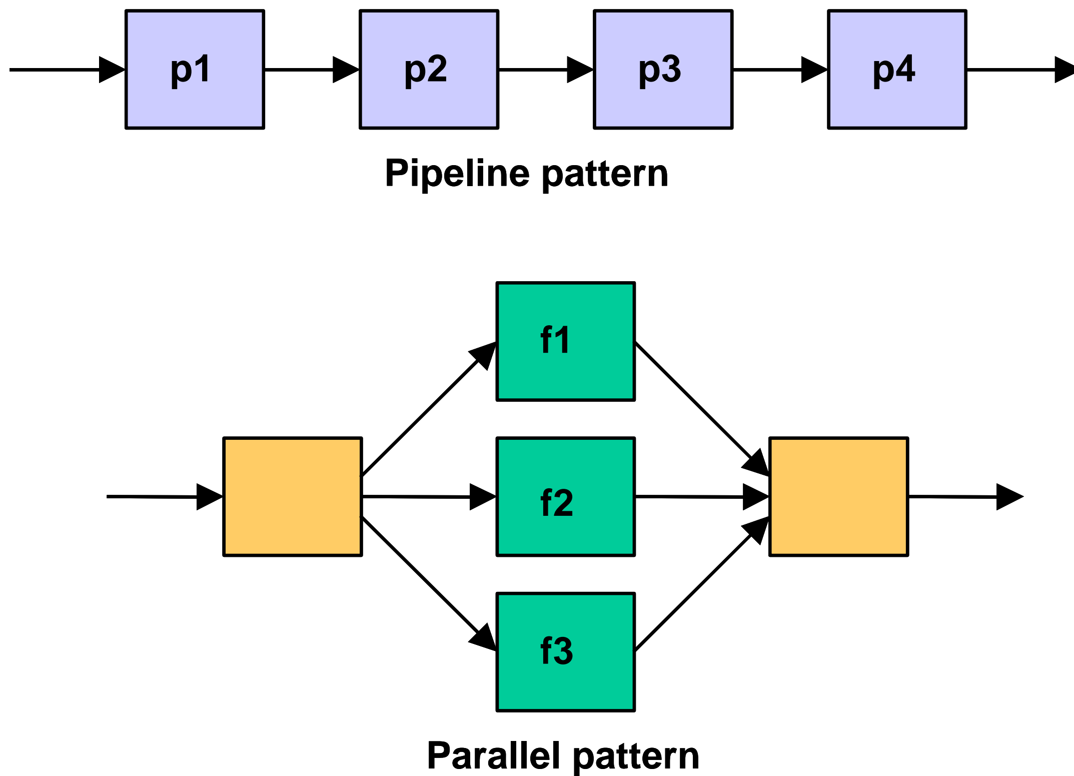


**Figure 3-4. Partitioning Scalability Patterns**

Pipelining is used when multiple separable computations should be performed serially on a set of continuous updated data. It is most effective when the computational and I/O load has a clear upper boundary. Typical applications include sensor data processing. In pipelining, computations are split between an appropriate number of applications. Ideally, each application requires an approximately equal time to process each set of data. This maximizes the utilization of the processors. If one computational application requires significantly longer than the other applications in the pipeline, it may introduce a performance bottleneck that interferes with the throughput of data. As each application finishes its computations on a given set of data, it passes its results to the next application in the pipeline and retrieves the data from the previous computation. The result is a relatively continuous, ordered flow of data and processing that can be spread across multiple processors.

Paralleling is applied for components that have computations that lend themselves to simultaneous execution. One application distributes the necessary data to multiple computational clients. When all computations are complete, a server that represents the endpoint of the processing may synchronize and synthesize the resulting data as required. As in pipelining, paralleling behaves optimally if all computations require roughly equal time, particularly if synchronization of the computations is required at the endpoint.

Downloaded from https://www.everyspec.com



An example of paralleling includes doctrine processing. In doctrine processing, multiple types of doctrine (e.g., weapons doctrine, display doctrine, etc.) may be in use at any time, and multiple doctrine statements for each type of doctrine may be active. A component may be allocated to each doctrine type and potentially for each active statement of a given type. Each track can then be compared against each of the active doctrine statements virtually simultaneously.

Combinations of architectural patterns may also be used to enhance scalability. For example, a component may use paralleling, while one of the parallel paths comprising that component uses pipelining. Additionally, each component in the partitioned component may use other architectural patterns that support other key design characteristics. For example, a component in a pipeline may use an active replication pattern for fault tolerance.

Scalability through partitioning provides good scalability within the limits of the intended range of load. However, it does not provide support for dynamically adapting to changing load. It is extremely useful where the load is known *a priori* to be bounded within a given range and where there is no significant advantage in being able to reclaim resources when the load is low.

###### 3.2.1.10.1.2 Load Scalability by Replication

Another method for providing scalability is through scalable process replication. With scalable replication, process replication is used to enable load sharing. Each replica of a load sharing performs a portion of the total workload. Different techniques for scalable replication may be required for clients and servers. Clients that are replicated for load sharing are called peer-clients. Servers that are replicated for load sharing are called scalable servers.

Figure 3-5 illustrates the use of replicated processes (R1, R2, and R3) to achieve load sharing. Each replica processes a subset of the incoming workload. Unlike the case of paralleled functions illustrated in Figure 3-4, where each function processed a different type of data, the replicas will process different instances of the same kinds of data.

A high-volume server is a server that has a significant likelihood of causing starvation of its clients due to the number of clients, the number of requests, or the quantity and frequency of data it is distributing. When a high-volume server becomes backlogged, it has the potential to interfere with mission critical processing, either directly or indirectly through some other inter-process dependency (similar to priority inversion).

Downloaded from https://www.everyspec.com



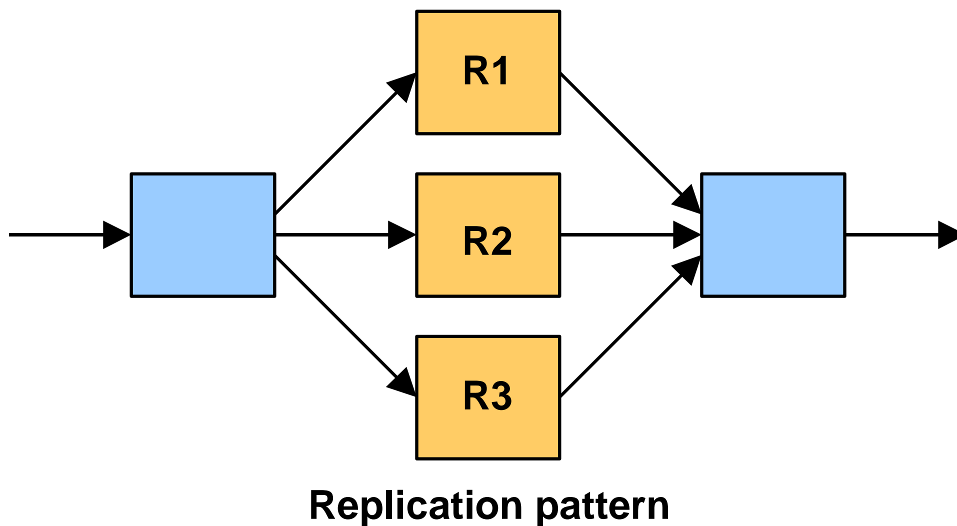


**Figure 3-5. Load Sharing by Replication**

The likelihood of encountering a high-volume server backlog can be reduced through the implementation of scalable servers. Scalable servers can distribute a load by a number of different methods. A method appropriate to the loading characteristics and synchronization requirements of each server should be chosen. Some methods for consideration include:

a. Assigning each scalable server a different set of clients.

b. Assigning each scalable server a different set of data/resources.

c. Providing a coordinator that assigns each new request in an unbiased manner (e.g., round robin or least load) to a different scalable server for processing.

Regardless of the method chosen, scalable servers should be capable of self-managing the load assigned to each replica. Typical methods for achieving this include the use of deterministic algorithms by which responsibility is determined, and assigning one replica as a coordinator for establishing/re-establishing load delegation. The method chosen often depends on the nature of the load.

Similar to high volume servers, client components can become overloaded with information that needs to be processed. Scalable replication in the form of load-sharing peer-clients is one way to overcome this situation. Each load-sharing peer executes as a separate process. The peers divide load through the use of a load-sharing algorithm contained within the replicas. This algorithm allocates a unique portion of the processing to each peer. The algorithm should account for the number of replicas in the system at any given point in time in addition to the type and quantity of the load. This information is used to divide the processing of data in an approximately even distribution across all peers. When the number of peer replicas changes, the processing load is readjusted at each peer. Other conditions that might require load readjustment

Downloaded from https://www.everyspec.com



may include changes in the type of load and or significant imbalance in the load distribution. In general, minimal coordination is required among the peers beyond the adjustment of load that occurs upon peer initialization or termination/failure.

The peer-client approach can potentially increase application performance by decreasing the overall load on a single computer. Such increases in performance need not double performance to provide benefits to the overall system. For example, the peer-client approach can be used to dampen the detrimental effects of a failure in the system by manipulating the number and placement of peers, thus distributing the impact of failures throughout the system. Peer-clients may also improve fault tolerance by increasing the number of processors that should fail before a function is lost. Finally, the ability to increase, migrate, or decrease peer components facilitates dynamic run-time load leveling as a means for maintaining optimum system-wide utilization.

Scalable replication provides the ability to adapt a system dynamically for a wide range of loads. Caution must be exercised in the design and implementation of scalable replication solutions, as changing the scalability of a component can potentially affect the overall system dynamics and may alter the performance of other components. The implementation of scalable replication can be significantly more complex than solutions that utilize computational partitioning. It can be especially complex when the component involved should maintain state data. However, scalable replication provides support for fault tolerance in addition to scalability, making it an excellent choice for mission critical components that should be fault tolerant and have significant or a varying range of load.

3.2.1.10.2 Guidance

OA should choose appropriate load sharing mechanisms to achieve scalability. In particular, the following guidance applies:

a. For components whose loads are predictable and have an upper bound, architecture patterns should be leveraged that divide computational responsibility across multiple executable processes. The partitioning should be selected such that the component is capable of handling the maximum possible load.

b. Where partitioning is used to support scalability, fault tolerance patterns should be considered to provide component availability.

c. Where the loads are highly variable, scalable replication should be utilized to maximize resource utilization.

d. Scalable servers should be designed to facilitate instantiation of one or more replicas across which load are distributed. An appropriate method for distributing load across these servers should be used. Replication patterns that support state data synchronization should be considered for servers that have state data maintenance requirements.

Downloaded from https://www.everyspec.com



e. To the degree practicable, scalable servers should be capable of self-managing the load distribution across the set of available replicas. They should be capable of the assignment of load and reallocation of load as the availability of replica change, without external intervention from a resource manager, except possibly for the notification of change in the number of replicas.

f. The peer-client design pattern should be employed for clients where a wide dynamic range of computational requirements makes its use beneficial.

g. In a peer-client design, the work of a client should be divided into *n* components known as peer-clients or replicas. For N peer-clients, each peer should concurrently perform approximately 1/N of the overall load. When a peer is lost, the remaining peers should assume a larger proportion of the work in accord with a deterministic algorithm (e.g., 1/(N-1) of the load). Conversely, adding a peer to the system should reduce the load on each peer (e.g., to 1/(N+1)). Component state data synchronization may be required between peers, for which appropriate design patterns for fault tolerance and state data synchronization should be evaluated.

#### 3.2.1.11 Real-time Performance

This section provides rationale and guidance for ensuring that OA components are designed so that both hard real-time and soft real-time requirements are met.

##### 3.2.1.11.1 Description

NWS functions are characterized by varying degrees of timeliness requirements. These range from human-paced decision-making to soft real-time background and throughput processing to hard real-time sensor and weapons management. Each of these requirements should co-exist smoothly in an open distributed combat system.

###### 3.2.1.11.1.1 Real-time Processing

Real-time processing is an often invoked but not well-understood term. For purposes of discussion, we consider four classes of processing requirements. As would be expected, all four classes are characterized in some regard with respect to the timeliness of their performance. Note that these classes are qualitative only for explanatory purposes and are not intended for rigorous analysis.

a. Non-Real-time: No timeliness related requirement other than user inconvenience.

b. Soft Real-time: Time-related processing requirements for which there is no sudden and distinct drop-off of computational value as with hard real-time, see below; rather, the value of the soft real-time computation becomes progressively more important as time passes. Such computations are typically scheduled at a lower priority level than hard real-time computations in order to ensure that they do not starve out hard real-time deadlines.

Downloaded from https://www.everyspec.com



Hard Real-time: Processing requirements, often but not always periodic in nature, for which failure to meet a processing deadline will result in serious, possibly fatal compromise of system operation or integrity. Such requirements are almost always grounded in the physics of the problem being solved and require operating systems with:

1. Multiple execution priority levels
2. Tightly bounded critical region execution times
3. Control for priority inversion

Extreme Real-time: Processing requirements with such low service latencies, often measured in microseconds, that normal relatively full-featured operating systems deliver insufficiently fast response and for which tightly crafted kernel executives must be used to meet latency requirements

The terms hard real-time and soft real-time are often used to characterize computing timeliness requirements. In distinguishing between these two terms, some authors suggest a time domain dividing line—often associated with periodic, (e.g., < 1 millisecond, < 100 microseconds, etc.). However, this approach is not sufficiently descriptive to be fully useful. For example, it is difficult to reach agreement on the precise value of such a boundary. Different application domains tend to produce different boundaries.

More importantly, many practitioners consider hard real-time to be associated with a requirement for deterministic performance characterized by guarantees of bounded response times and latencies for operations, service calls, and information transfer. In this context, soft real-time systems are those for which similar tightly bounded responses are not required.

In a practical sense, such applications are most often associated with physics-based computing requirements (e.g., in sensor and actuator control applications). A widely accepted body of theory is available for analysis of periodic processing with mandatory deadlines—the *rate monotonic* theory. Systems to which rate monotonic analysis may be applied are sometimes considered to be synonymous with hard real-time.

Hard real-time requirements of the type described in the preceding paragraph are generally based on the physics of the problem, and they are often used to ensure that certain physics-based error bounds are observed. By controlling the timeliness of computations, the physics of the problem are controlled. For example, the loop update rate for missile midcourse guidance uplink commands derives from error containment requirements associated with the missile guidance process. In effect, the missile guidance update periodic contains a designed-in safety margin. If these bounds are exceeded, the missile may depart from the trajectory required for target interception. While such computational requirements may not require microsecond-level scheduling granularity, they are nevertheless vital to correct system operation.

Note that even if absolute microsecond-level "hard real-time" deadlines are not needed, there still exist timeliness bounds beyond which deferral of the computation will result in drastic

Downloaded from https://www.everyspec.com



negative consequences for system operation. In the example of the previous paragraph, the missile error bounds referred to may quickly be exceeded if an update is not available for too long a period. For this reason, both component design and overall system design should carefully implement mechanisms for controlling flow of control during system operation. This is particularly true in the presence of RM. Component thread design, buffering mechanisms, and priority assignment should be done in a manner consistent with system engineering objectives and in recognition of the interaction of the flow of control characteristics of operating systems and communication mechanisms.

Proper utilization of these control mechanisms during component and system design is the key to simultaneously meeting "hard real-time" requirements while providing balanced service for remaining computational tasks.

3.2.1.11.1.2 Quality-of-Service (QoS)

Closely associated with real-time is the concept of QoS. QoS is often associated with network and information transfer characteristics. A common usage relates to the interactions between network communications paths and applications that are designed to seek a defined level of network throughput and/or latency. Sometimes network QoS is expressed in terms of ***jitter***, a variation in time of arrival of network packets at a destination. In either case, applications operating within a QoS range will either request a specified QoS from the underlying network infrastructure or they will adapt their performance expectations to the QoS available from the network. In turn, this implies that the network should be capable of managing and allocating QoS for a variety of users.

In many instances from the commercial world, QoS is a marketable commodity, and network service providers charge for higher QoS levels. In such situations, cost becomes the metric whereby user decisions are made concerning (1) what QoS to request from the provider (2) what adaptations to make in QoS utilization to remain within defined cost targets. Likewise, the provider, motivated by profit, may decide how much aggregate resource to make available for use based upon market projections and other business considerations.

Parenthetically, it should be noted that such systems generally serve a variety of customers for whom inter-application interactions and dependencies are negligible. This is significant for the following discussion concerning real-time, mission-critical systems such as Navy warfighting systems for which strong inter-component dependencies do exist.

While this guidance document incorporates the network QoS concepts discussed above, it generally employs the QoS idea in a somewhat broader manner. Specifically, the idea of QoS is expanded to incorporate processor resource use in addition to network use. This is a natural extension to the QoS concept for large distributed real-time systems. In the latter case, the components of the system should interoperate frequently in a strongly interdependent manner. Thus, the activities of each component are potentially dependent on the successful operation of many other components.

Downloaded from https://www.everyspec.com



In this situation, the concept of components negotiating QoS from an independent market-like service provider has little relevance. Instead, the system and all of its components should use the available resource base in a harmonious and cooperative manner. Furthermore, both processing and information transfer assets are clearly contributors to the overall successful operation of the system. This difference becomes even more acute when RM is introduced into the equation (see Section 3.2.2.8). In the RM case, both processor and network metrics are defined for measurement and management.

3.2.1.11.1.3 Distributed Real-time Systems

Perhaps the most interesting and challenging case of all is the situation faced by designers of large, complex, mission-critical, distributed real-time systems. For such systems, both real-time scheduling considerations and factors associated with QoS (see discussion below) are present simultaneously and sometimes in the same application. In addition, there are inevitably a substantial number of end-to-end requirements that transit several real-time programs during their accomplishment. These paths often implement low latency responses to critical external events.

Furthermore, while discussions of real-time more often focus on periodic and operating-system scheduling, distributed real-time system designers should inevitably consider as co-equals the timing characteristics of the information transfer mechanisms as well as those of the operating-system scheduler.

Another consideration is the relationship between periodic processing, event-based processing, and operator control. In large distributed real-time systems, it is difficult to enforce the structuring mechanisms (such as rate monotonic theory) that provide the theoretical foundation for guaranteed hard real-time performance. It is not so much that those structuring mechanisms are not achievable, but rather that implementation of them would inevitably engender a greater degree of rigidity to the overall system design than is consistent with modern design practices. Hence, the real challenge for distributed real-time system designers is to ensure that real-time properties are preserved while designing an architecture and a set of components that meet the supportability metrics introduced in Figure 3-6.

Downloaded from https://www.everyspec.com



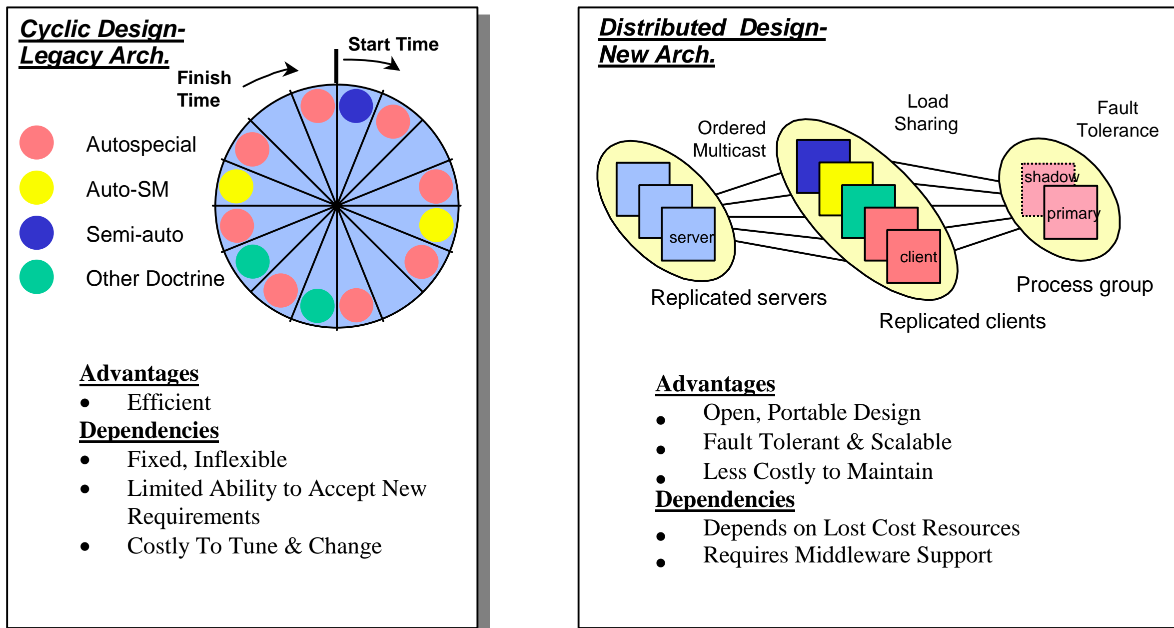


**Figure 3-6. Comparison of Design Models**

Contemporaneously with the emergence of the need to deal with large, complex systems is the development of a number of enabling technologies that were not necessarily invented with real-time requirements in mind. These technologies include networks, distribution middleware, and RM. The composability and supportability benefits of these technologies are compelling; however, their suitability for real-time use has not been broadly tested until recently. Fortunately, the technology base introduced in the preceding paragraph has evolved substantially, and in many cases it is now fully real-time capable in all but the most stringent applications.

What emerges from this confluence of design objectives, commercial practice, and technology tools is a new system architectural concept that provides a far more flexible basis for meeting supportability metrics. Rather than using shared memory and fine-grained operating-system scheduling mechanisms, this approach connects large numbers of components via networks and middleware-based inter-component communication mechanisms. The new architecture, contrasted with traditional application program design in Figure 3-6, provides increased flexibility.

In the old style cyclic design, all processing requirements are contained in a fairly large, multi-faceted application program. Each processing requirement is mapped to one or more time slices within an overall processing cycle. The total set of requirements is interleaved onto the available processing resources, either a single processor or a symmetric multi-processor. Communication between processing segments is often accomplished via explicitly shared memory. Synchronization between segments is performed via semaphores, mutexes, locks, etc.

Downloaded from https://www.everyspec.com



The distributed approach generally separates out-processing requirements into a number of smaller application programs, each of which might have been a processing segment in one large application in the older cyclic model. The distributed applications communicate with each other via an inter-process communication mechanism, often middleware. The middleware provides high-level communication semantics, much as a high-level programming language provides a higher-level abstraction than assembly language. The middleware often provides support for reliable communication, multicast, state data conservation, synchronization, replication, and fault tolerance. While the distributed approach may seem profligate in its use of processing resources, the new architecture trades expanded use of processors and communication, which has become relatively cheap in today's environment, for design flexibility and supportability benefits.

#### 3.2.1.11.2 Guidance

Support for meeting this guidance consists of managing a number of computer resources in accord with the nature of the real-time problem. In the guidance stated below, an execution path is a set of processes that should run in succession to perform a complex application function. A path is initiated by an event, and the processes that occur during path execution may result in events that initiate other paths. Initiation of a particular path may be conditioned on completion of one or more other paths. A mission-critical path contributes directly to mission success. Failure to meet timeliness requirements for a mission-critical path can result in mission failure.

a. Process and task/thread priorities should be used when necessary to ensure that mission-critical hard real-time deadlines are met, if necessary, at the sacrifice of soft real-time deadlines. However, the use of thread priorities to achieve correctness, such as using priorities to ensure that events are handled in the correct order, should be avoided.

b. Management of I/O and throughput processing should be accomplished by means of queues that buffer program units from each other and prevent busy states in one part of the system from impacting other parts of the system. Output processing may be handled through a queue mechanism or through the use of separate threads servicing different consumers. Also, most commercial middleware products that support threads (e.g. Object Request Brokers [ORBs]) will handle the output queuing in a satisfactory manner without blocking the application and without application intervention. In the event that a lower-level communications mechanism is used (e.g., group communications, TCP, UDP), then queuing of outputs and an associated thread to send the data is highly desirable.

c. The previous requirement may cause significant performance degradation in synchronous interactions such as a CORBA 2-way invocation, since the thread is blocked until the application gets around to responding. Exceptions to the previous requirement include three cases: (1) when processing requirements for input are small (e.g., updating a data item in an internal data store), (2) processing of high-urgency inputs, and (3) synchronous interactions such as Remote Procedure Call (RPC)-style request/reply interactions.

Downloaded from https://www.everyspec.com



d. Mission-critical execution paths in applications hosted by the OACE should be identified and described in terms of process sequence and timeliness requirements (e.g., hard vs. soft real-time, time in which process should complete). Timeliness requirements should be specified for each process in the path and for the path as a whole. Note: The timeliness requirements will be employed by the RM capability (see Section 3.2.2.8) to predict and manage replication, load leveling, or component migration.

#### 3.2.1.12 Process, Thread, and Memory Management

This section provides rationale and guidance for designing efficient real-time multi-threaded OA components that ensure correct operation. The section discusses techniques for ensuring that OA components are designed to avoid the inefficiencies and unpredictability of run-time memory allocation, thread creation, and garbage collection. Also, it provides guidelines to ensure that processes and threads are not hard mapped to specific CPUs in processors with more than one CPU. Generally, it is desirable to avoid mapping of threads to specific CPUs; however, this may not be feasible in all cases (e.g., due to poor cache performance or to improve I/O performance). Use of such mapping should be justified by the designer and approved as an exception when necessary. Finally, data protection and access synchronization are part of the overall requirement to preserve data integrity in mission-critical and safety-critical systems.

##### 3.2.1.12.1 Description

With modern languages and operating systems, it is possible to defer binding of many system resources until runtime. This has the advantage of flexibility of operation, but it also has the disadvantage of potential inefficiencies and lack of predictability during operations. Since efficiency and predictability are extremely important aspects of most real-time systems, pre-allocation of computing resources such as memory, processes, threads, file descriptors, etc. is strongly recommended. The preferred design approach to control these potential sources of non-determinism is through the use of pre-allocated pools of memory and threads.

Memory allocation is a good example. Programs can allocate memory from a global heap at runtime. Such programs can also release memory during operations. Over time, the global memory heap is almost certain to become fragmented or exhausted unless mechanisms are provided that enable the operating system to reclaim and reorganize unused memory. If not carefully planned as a part of the real-time flow of processing, this activity, called ***garbage collection***, can adversely impact the operation of a real-time system. In particular, for creating real-time applications, programming language facilities where unused memory disposal and garbage collection are performed automatically should be scrutinized carefully in light of their ability to compromise application determinism.

For a variety of reasons, real-time computer programs are often multi-threaded. Threads may be defined to support periodic, event handling, communication with other programs and devices, etc. In multi-processor systems, it is often possible to constrain individual threads to execute on specific processors. Such constraints usually make programs dependent upon the implementation details of the specific computer, which in turn makes them difficult to maintain and port to other processors.

Downloaded from https://www.everyspec.com



In reliable systems, access to data shared among processes and threads should be synchronized and sequenced in order to preserve data integrity, e.g., to prevent corruption of the shared data. A thread-safe object, unit of code, or component is one that can be executed by multiple threads in a virtually simultaneous manner without unintended interactions or side effects. To achieve this, certain standard mechanisms should be used.

3.2.1.12.2 Guidance

The scheme most often used for accommodation of the run-time range of processing requirements of real-time systems is pre-allocation of pools of resources. In addition, programmer-supplied services are needed to explicitly manage withdrawal from and return to the pool. Process, thread, and memory management guidance is as follows:

a. To promote portability and flexibility of design, tactical component threads should not be mapped to a specific processor. There is one exception to this: real-time programs are usually I/O intensive. Mapping I/O interrupts and processing to a processor dedicated to handling the I/O stream can lead to more efficient use of the remaining processors and is therefore not excluded.

b. Thread-safe methods should be used for synchronization of shared data and access to non-reentrant service routines by multiple threads. Such mechanisms include low-level operating system semaphores, adaptive middleware or framework thread-safe libraries, and language-based run-time support structures that provide inherent synchronization semantics, e.g., Ada and Java. Where high-level language constructs support such synchronization with sufficiently low latency, the high-level language constructs should be used.

c. When possible, object-encapsulation mechanisms and thread-safe access mechanisms should be used to encapsulate data that is not needed external to the thread (i.e., such data should be encapsulated within the visibility of the thread).

d. Thread priorities should be established in order to ensure that urgent processing for which defined latency requirements exist are able to preempt processing of lower importance, except in cases where state data consistency would be affected negatively by the use of thread priorities.

e. For mission-critical components, process priority distinctions should not be required to ensure correct system operation and satisfactory performance. Operating system scheduling management of execution flow should be implemented by means of thread priorities. Thread scheduling considerations are the primary determiner of real-time performance, even for situations where multiple components are allocated to the same processor. If used at all, process priorities should be confined to setting non-mission-critical process priorities to a low level so that such processes do not interfere with mission-critical components.

f. Mission-critical applications should only allocate heap memory at runtime as an initialization activity.

Downloaded from https://www.everyspec.com



g. Whenever possible, applications should actively manage system resources such as memory allocation and reallocation, even if this management should be performed in application code. Management of dynamically allocated assets such as garbage collection should be entrusted to system software only where system mechanisms exist by which applications can control the flow of processing associated with the system's management of those run-time allocated resources. Such mechanisms should be used by the application so as to ensure that system dynamic asset management operations do not interfere with the operation of mission-critical components.

h. Careful consideration should be given to the number of threads employed by real-time components. Developers should not construe this to imply that threads are not to be used, but rather that they should not be used indiscriminately, as they can consume significant memory resources and may affect the efficiency of some schedulers. Consideration should be given to the use of thread pools rather than hard assignment of threads to tasks, where a sizable number of predominantly idle threads might otherwise be required.

i. Components should maintain precise control of the accretion, use and disposition of system resources (memory, threads, devices, system services, etc.), so that at shutdown time they can either return such resources to the operating system or a system-wide resource control capability can ensure that the resources are released cleanly.

j. All memory and thread management should be performed in a thread-safe manner (i.e., access to data structures and non-reentrant service routines that may be shared among multiple threads should be protected by thread-safe mechanisms as discussed in Item c above.)

k. Object-oriented software components should avoid unnecessary object instantiations, as each object instance requires system resources. Furthermore, when an object is no longer needed, it should be explicitly deleted or marked for deletion by the language's garbage collection mechanism.

#### 3.2.1.13 Data Brokers

This section provides rationale and requirements for providing data broker and/or layered adaptation mechanisms to accommodate interfacing to legacy capabilities.

##### 3.2.1.13.1 Description

Data brokers are translator components that are positioned between two or more disparate systems to facilitate communications. They adapt the electrical interface, protocols, formats, and syntax of one system to that of another. Since the data broker itself is software that should be developed and maintained, this solution leads to increased development and maintenance cost, albeit likely a small increase. Furthermore, data brokers are only useful in situations when performance objectives are not adversely impacted by the overhead that the data broker adds. Thus, a data broker is a compromise solution, and it should only be used when large-scale modifications to either of the interfaced systems are prohibitive due to cost or schedule.

Downloaded from https://www.everyspec.com



Data brokers provide a standardized, consistent API for legacy applications that, when employed, “translate” the data representation of messages into one that is compliant with the OACE architecture. Conversely, information that is sent to the legacy system from the OACE architecture is translated into a data representation that is compatible with the legacy system. Data brokers permit the phased implementation of the OACE architecture by insulating legacy applications. Furthermore, when properly designed, data brokers ensure that changes will not be required in the OACE components if and when the legacy components are modified to become OACE compliant.

Adaptive layers, generally provided as linkable libraries, confer similar isolating services. The adaptive layer exports an interface that hides the details of the legacy system to which communications should be maintained. The use of data brokers and adaptive layers will contribute to reduce efforts in maintenance and upgrades by isolating the legacy applications from OACE.

3.2.1.13.2 Guidance

The guidance for legacy capture within OA is that a data broker and/or adaptive layer approach may be used to interface with legacy components.

### 3.2.2 Open Architecture Computing Environment (OACE)

The technology base, or infrastructure, for OA is referred to in this document as OACE. Since tactical applications will be implemented as a set of loosely coupled software components, the fundamental requirement for OACE is to provide a distributed real-time computing environment for NWS elements in the future. For these elements, issues of fault tolerance, timeliness, scalability, expandability, and maintainability are vital.

OACE is a high-performance computing environment for distributed processing, and it is capable of controlling and executing distributed tactical applications. OACE provides a common computing environment with services for application loading, RM, inter-application communication, fault tolerance, and adherence to real-time application requirements. The resource base controlled by OACE consists of a set of computers, internal and external network interconnection equipment, network media, operating and control software, communication software, and interface software. Distributed application software will load and execute in the OACE environment.

Listed below are a number of key technical attributes of the OACE:

a. CORBA inter-component distributed object mechanisms when performance permits

b. Distribution of continuously refreshed data via publish/subscribe protocols

c. Maintenance of state data coherency via group/ordering protocols

Downloaded from https://www.everyspec.com



d. RM, with allocation of portable processing components (i.e., “run-alike” across brands, operating systems, etc.) onto a pool of virtually homogeneous computers; see discussion in the Guidance section of 3.2.2.3

e. Instrumentation data from operating systems, network components and applications

f. Integrated failure management services across infrastructure components

g. Information security integration with resource management

h. Processors with suitable responsiveness and queuing of I/O interrupts

i. Effectively uniform switch interconnect with QoS features (e.g., the network Resource Reservation Protocol [RSVP])

j. Time synchronization via system-wide service (e.g., Network Time Protocol [NTP])

k. Open, standards-based COTS products (as near mainstream as possible)

The architecture of the OACE supports the principles, structure, and components that form the basis upon which the application components of OA are built. Thus, the attributes of the OACE support development of application components in accord with the guidance contained in Section 3.2.1. It includes technologies needed to provide guaranteed QoS to real-time applications. Figure 3-7 illustrates the functions and interrelationships of OACE.

Downloaded from https://www.everyspec.com



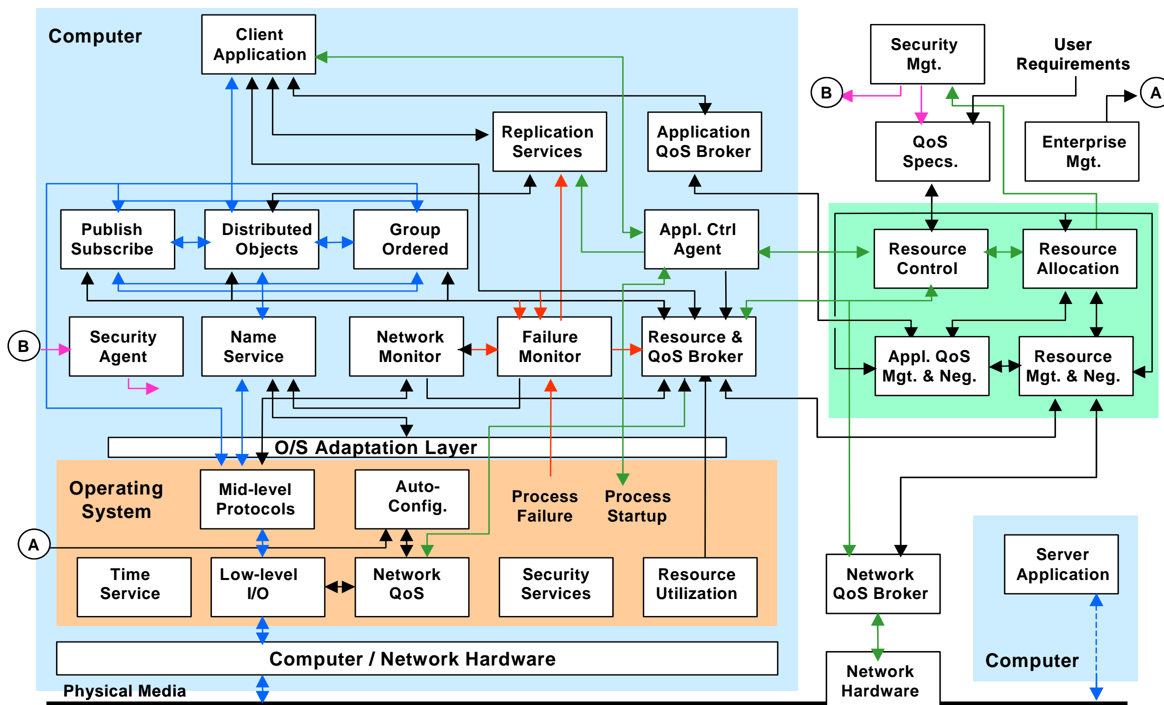


**Figure 3-7. Open Architecture Computing Environment**

Note that this representation is given in functional form and does not necessarily represent a physical design. For example, three classes of communication protocols are shown: distributed objects, publish/subscribe, and group-ordered communications. These classes exist as separate tools at present, forcing designers to select one and only one of the three for a particular interface; however, this may not be true in future products. Current middleware marketplace evolution appears to be in the direction of combined functionality in a single product (e.g., a fault-tolerant CORBA based upon group communications with publish/subscribe functionality).

This architecture provides the capabilities and infrastructure needed to construct multi-component, replicated, distributed object real-time systems. The structure is repeated throughout the computers of OA. Thus, although the diagram elaborates the technology base for a computer containing a client component, the computer hosting the server application contains a comparable structure. Also executing within the distributed system's collection of computers is a set of RM components that interact with the computers, network components, and applications of the distributed system to provide QoS management.

The OACE reference architecture, shown in Figure 3-7, focuses primarily on the infrastructure of operating system and third-party software needed to support OA warfighting applications and training systems. There is a corresponding physical network and computing equipment architecture as well. Key elements of the physical architecture are shown in notional

Downloaded from https://www.everyspec.com



form in Figure 3-8. The underlying technology components of that physical architecture are discussed in this section along with software infrastructure components.

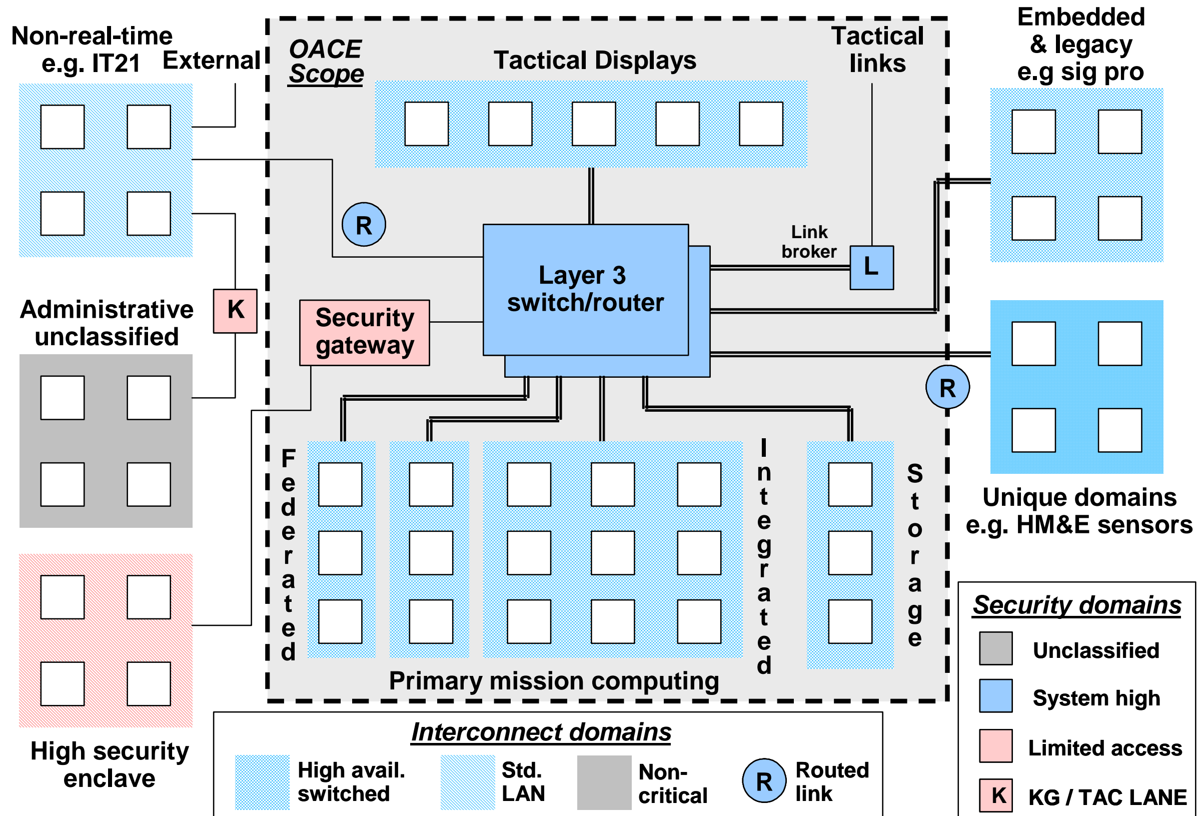


**Figure 3-8. Notional Physical Architecture**

Several amplifying comments should be noted. First, the core of the architecture is a redundant platform-wide Layer 3 switch configuration. Layer 3 switches provide both switching and routing capabilities. Second, the overall architecture subdivides into a number of computing domains. Each of these domains may potentially employ a different technology base from other domains. Within a domain, separate network and switching technology may be used if needed for performance reasons, e.g., for signal processing. Specifics are system and implementation independent. Third, the architecture partitions out information security concerns into separate domains, with security gateways providing isolation and access control. Finally, the primary concern of this guidance documentation set is the mission-critical set of resources at the center of the diagram (denoted by dotted box and gray shading). This should include all warfighting applications that are candidates for commonality of implementation as well as those mission-critical applications that are unique to a particular platform. While not all such applications have real-time requirements, many of them do. The domains identified are as follows:

Downloaded from https://www.everyspec.com



a. Mission-critical warfighting

b. General purpose support and planning

c. Non-real-time administrative

d. High-security enclaves

e. Legacy and high performance embedded

f. Special purpose (e.g., Hull, Mechanical, and Electrical [HM&E])

Within the mission-critical domain, several types of computing resources are identified: (1) federated computing assets, (2) integrated computing assets, (3) tactical displays, and (4) storage. The utility of displays and storage is perhaps self-evident. The reasoning behind representing both federated and integrated computing resources was discussed in Section 3.1.3, Federated vs. Integrated Architectures. These two differing allocations of assets represent the difference between an approach where each developer brings its own independently chosen computing resources vs. the total ship computing approach, wherein a common pool of like resources is shared between a number of systems and subsystems.

The following list constitutes the set of technologies considered under the scope of OACE:

a. Physical Media

b. Cabinets

c. Information Transfer

d. Computing Resources

e. Operating Systems

    1. General Purpose

    2. Real-time

f. Adaptive Middleware

g. Distribution Middleware

    1. Distributed Object Computing

    2. Publish/Subscribe Protocols

    3. Group-Ordered Communication Protocols

Downloaded from https://www.everyspec.com



4. Data Parallel Protocols

h. Frameworks

i. Information Management

j. RM

1. Information Recording and Assessment

2. Failure Management

k. Information Assurance Services

l. Time Synchronization Services

m. Programming Languages

Each layer constitutes a part of the underlying technology base and is discussed in the sections that follow. The OACE provides a critical set of functions required for the proper operation of applications components within OA. Among these characteristics is a capability for the OACE to monitor itself. Another is portability of the OACE technology base across a variety of COTS products and standards from which future technology refreshes are likely to be chosen. COTS hardware and support software is expected to evolve rapidly. Therefore, the OACE should be designed to support cost effective modification, addition, and replacement of hardware and software components.

#### 3.2.2.1 Cabling and Cabinets

This section provides rationale and guidance for the cabling plant and cabinet architecture of OA. The computing resources need not be accommodated within the footprint and connectivity constraints of existing cabling and cabinetry if sound technical and business case reasons exist for making changes.

##### 3.2.2.1.1 Description

New construction ships are free to use the latest cabling and cabinet technology. However, as stated in Section 3.1.3, the decision of whether to conform to existing cabling and cabinet constraints for backfit implementations is a cost-benefit tradeoff. An OA re-implementation of existing tactical programs will almost certainly fit inside current cabinet footprint on most ship classes with minimal effect on legacy I/O nodes. However, the network switching design is likely to be quite different, due to the presence of more computing nodes. Additional study will be required to determine if the installed base of fiber can handle it, despite its relatively flexible patch-panel-based design.

Downloaded from https://www.everyspec.com



3.2.2.1.2 Guidance

Guidance for use of existing cabinet footprint and cabling is as follows:

a. OA should not impact compatibility with the cable plant design on existing ships unless there is a business case reason for doing so.

b. OA should be designed to facilitate installation of its components in equipment enclosures that meet space and weight budgets for legacy ships. Physical design constraints should apply to cabinets, computer subassemblies, junction boxes, network media, and all other items comprising the physical manifestation of OA.

3.2.2.2 Information Transfer

This section provides rationale and requirements for designing the OA information transfer capability. The goal is to ensure that information transfer bandwidth and latency requirements are met and that no single location within the network represents a constraint on information flow or process to processor allocation. This latter feature is intended to support the allocation and reallocation of processing tasks at runtime.

3.2.2.2.1 Description

The network will provide the primary means of information transfer within OA. While a limited number of point-to-point interfaces will continue to exist for some time, NWS elements will continue to migrate toward a full-network implementation. For instance, programmatic and technical execution plans for system migration to OACE are ongoing for both Aegis and Ship Self Defense System (SSDS).

3.2.2.2.1.1 Basic Network Functionality

Three basic functions provided in the information transfer infrastructure are (1) connecting a processor to the network, (2) interconnecting all the components on a single network, and (3) connecting networks together. In an OA, these functions are to be provided by (1) network interface cards (NICs), (2) switches, and (3) routers, respectively.

A NIC is the means by which a processor accesses the network to send or receive data. A NIC is required for each type of networking technology used by a processor, e.g., Fast Ethernet, Gigabit Ethernet, or Asynchronous Transfer Mode (ATM).

A switch is a networking device that receives and transmits data on a number of ports. Unlike shared network architecture, a switched architecture dedicates each port on a switch to the device connected to it. In other words, a processor connected to a Fast Ethernet switch has a dedicated 100-Mbps path to transmit and receive data. The internal components of the switch temporarily connect switch ports in order to enable transfer of data to other computers or to additional switch components. The number of active ports a switch is able to realistically support depends upon the throughput of the backplane of the switch.

Downloaded from https://www.everyspec.com



Routers provide for the forwarding of data across different networks (a network is defined here as a single broadcast domain). Routing functionality may physically be provided in switch hardware, but the functionality is distinct from an architectural perspective.

3.2.2.2.1.2 Communication Mechanisms

Three basic communication mechanisms exist in the information transfer infrastructure: Uni-Cast, multicast, and broadcast. Uni-Cast messaging involves a sender transmitting a message to a single receiver.

Both multicast and broadcast involve a sender transmitting to multiple receivers. The functionality of these two mechanisms is distinguishable at two layers. At the higher IP layer, broadcast messages are transmitted to all receivers within the defined domain. For multicast, a sender transmits to a selected group of receivers defined by membership in a multicast group.

The way in which the higher-layer message is instantiated on the lower layer depends upon the type of network media being utilized (e.g., ATM, Ethernet). Broadcast and multicast messages from the higher layer are transmitted as broadcast messages at the lower layer in a shared media. Techniques such as Internet Group Management Protocol (IGMP) snooping have evolved in shared media to filter multicast messages to just those members of the multicast group. When one or more senders should transmit the same message to multiple receivers, multicasting is more efficient than point-to-point uni-casting.

3.2.2.2.1.3 Additional Network Features

The network should also support one of OA's most important design characteristics: deferred binding of program components to physical resources. Middleware supports deferred bindings by providing location-transparent communication services. RM supports deferred bindings through a capability to assign computational components to physical computers just prior to operations or even dynamically as the system runs.

Another key requirement is that the network should provide uniform service performance to real-time applications. RM enables much more flexible binding of computing tasks to processors. Potentially, dynamic RM can allocate any program capable of running on the real-time computing unit (see Section 3.2.2.3) to any computer of this class. The corollary to this allocation capability is the fact that any computer within this class may eventually originate a message destined for any other computer of like class in the system.

For this reason, all paths through the network connecting real-time units of computing resources should be effectively uniform in performance, or at least they should provide a specified minimum performance. In other words, each network path should be equally capable of supporting the overall required latency and throughput requirements for inter-computer message traffic between mission-critical components.

Switched networks are most likely to possess this virtually uniform performance characteristic. For such networks, the uniformity of performance requirement is generally only a

Downloaded from https://www.everyspec.com



concern on the individual switch-to-switch links. By definition, traffic on edge-device-to-switch links is limited to the available link bandwidth, which is, in turn, entirely devoted to the single device to which it is attached. Therefore, some configuration control over internal network characteristics is likely to be needed. This control, both in terms of instrumentation and reconfiguration, should be entrusted to the RM capability.

3.2.2.2.2 Guidance

The following constitutes guidance required of the information transfer capability by OA.

a. The information transfer infrastructure should provide an effectively uniform information transfer capability from any source in the network to any destination in the network with respect to connectivity, latency, and throughput. Effectively uniform means that all network paths will provide at least a defined minimum level of service.

b. The information transfer infrastructure should provide fault tolerance of information transfer paths.

c. The information transfer infrastructure should provide standards-based QoS control mechanisms for bandwidth reservation (e.g., RSVP).

d. The network infrastructure should provide standards-based mechanisms to dynamically modify routing paths for the purpose of ensuring the uniformity requirement following damage or disruption to the overall network. Conversely, such standards-based mechanisms may be used to ensure maximum QoS to components requiring it.

e. A common set of information transfer services should be provided to coordinate and facilitate internal communications. OACE should employ industry-standard protocols, such as the IP suite (see Section 3.2.6 for guidance on standards selection).

3.2.2.3 Computing Resources

This section provides rationale and guidance for selection of computing resources for OA from the market of high-performance, low-cost, commercial processors.

3.2.2.3.1 Description

From a maintainability standpoint, one of the benefits of distributed computing is the capability to modify system capacity without requiring changes to applications. However, due to the rapid turnover of processor technology, this necessitates portability of applications across various commercial processors. As a means for reducing costs, the Navy intends for OA to take advantage of, as much as possible, the high-performance low-cost mass-produced commercial computing market, while maintaining this portability condition. In order to do this most effectively, processor selections for OA should be made from mainstream single- and dual-processor boards. Niche market products such as many-processor symmetric multi-processors

Downloaded from https://www.everyspec.com



should be avoided unless application performance requirements cannot be met cost effectively by other means.

Real-time systems are almost universally I/O intensive systems. Handling many I/O interrupts per second can impose a large overhead penalty on a single processor computer. For this reason, some implementations utilize a second processor to field interrupts and perform initial I/O processing. This is a critical factor in equipment selection. There are multiple ways to handle this; one method is to use symmetric multi-processors (SMPs) and the operating system's inherent support for multi-processing. In this scheme, I/O interrupts may be bound to one processor exclusively. In any case, the critical design goal is to shield at least one processor from the burden of I/O handling, so it can make progress on application computations.

SMPs have frequently been used in the past to meet real-time processing requirements. Up to a point, explicit use of SMP shared memory has the potential to provide better performance than loosely coupled message passing implementations. However, this apparent advantage can prove illusionary as a system is scaled up due to memory conflicts, memory bus saturation, and operating system critical region exclusion periods. Where only an SMP will provide the required performance, clearly this technology should be used. However, the loosely coupled approach (both physically and logically) is preferred where performance demands necessitate and where SMP constraints would introduce limits to scalability. Note that for software portability, even when an SMP forms the underlying computing resource base, a loosely coupled message passing logical design is preferred because it is easier to design and debug. In essence, modern low cost, fast processors, and networks are employed to produce application designs that are cheaper to build and maintain.

VME card cages have been the typical delivery mechanism for shipboard computing. While this practice is expected to continue for some time due to market presence, the situation is changing and other alternatives are likely to become available. Therefore, this guidance document does not recommend restricting shipboard computing delivery mechanisms to Virtual Micro-bus European (VME) only.

The NWSs included in OA contain considerable breadth of processing requirements. While it is certainly possible to meet all OA processing requirements with one class of processor, it may prove to be advantageous to provide multiple classes of processing capability within the scope of OA. The guidance here is to move away from symmetric multi-processors; however, a multi-tier computing resource approach need not exclude computer/operating system combinations that provide less than the most stringent level of real-time capability and performance. For example, the processing resources devoted to display and other soft real-time activities need not meet the challenging requirements imposed by a radar control program or signal processor. In addition, it should be noted that where processing requirements dictate a specialized solution, such solutions might be adopted as exceptions to the guidance below. This partition of resources will reduce but not eliminate the flexibility inherent in an RM capability.

Downloaded from https://www.everyspec.com



3.2.2.3.2 Guidance

OACE computing resources should consist of hardware and system software with sufficient capacity and performance to host the distributed OA applications.

a. A *virtually homogeneous* environment should be provided for OACE-compliant components within the OA. The term ***virtual homogeneity*** refers to the characteristic of a system whereby each computing element performs similarly enough so that it meets the minimum performance, latency, and determinism requirements of the components that make up the system. Virtual homogeneity allows realization of the benefits of commonality without requiring absolute equipment commonality.

b. Processors with direct IO connections and, therefore, fixed computational task-to-processor mapping, are exempt from the virtual homogeneity guidance, item a above.

c. The OACE computing architecture should be based upon fundamental uniform building blocks called ***units of computing***. The unit of computing is defined as a combination of processing speed, memory, network performance, latency, and a selection of industry standard interfaces. The unit of computing concept provides a hardware modularity that permits the development of a system that is scalable, fault tolerant, allows for resource sharing, and enhances the portability of the functional design.

d. Units of computing should be combined to accommodate, via simple scaling, a variety of applications that would otherwise require unique processing solutions.

e. Each unit of computing should consist of a single or dual CPU under the control of a single instantiation of the operating system. The unit of computing should be sized to meet the requirements of the distributed tactical applications and should be characterized by the following performance parameters. Details follow below in Table 3-3.

f. The values selected for parameters in Table 3-3 should be sufficient to accommodate the needs of the tactical applications and should be representative of low-cost, high-performance, mainstream, commercial computers. The values for each of these parameters should be selected, such that the units of computing are not excessively oversized for the majority of requirements yet can be scaled to support the needs of computing intensive warfighting system elements.

g. Scalability of the OACE may be achieved by use of the following mechanisms:

1. Additional units of computing

2. Incremental upgrades of components in the unit of computing (e.g., memory)

3. Technology refresh

Downloaded from https://www.everyspec.com



**Table 3-3. Units of Computing Performance Parameters**

| PARAMETER | DESCRIPTION |
|---|---|
| SPECint95 | SPECint95 and SPECfp95 are industry standard measures of processing power. |
| SPECfp95 | |
| Memory | The quantity and type of memory for a unit of computing should be specified. |
| High Speed Network Interface | The type of high-speed network interface should be specified to facilitate combat system wide integration. |
| Transmit I/O Capacity | These parameters ensure sufficient bandwidth and determinism between distributed components. |
| Transmit I/O Latency | |
| Interrupt Response Latency | Interrupt response latency is the maximum time (in µsec) allowed from the generation of a hardware interrupt by a device to the execution of device driver or kernel code servicing the interrupt. This worst-case time includes all possible periods when interrupts are disabled in order to guard some operating system critical section. |
| Process Dispatch Latency | Process dispatch latency is the maximum time (in µsec) allowed from the generation of a hardware interrupt by a device to the execution of user code requesting service from that device. This worst case time includes the interrupt response latency, the time required to execute device driver or kernel code servicing the interrupt, overhead required in context switching to the user process, and any critical sections when context switching is disabled by the operating system. |
| Fast Clock Resolution | This is the period (in µsec) of the fast clock |
| Fast Clock Access Time | This is the maximum time (in µsec) required for a user process to read the fast clock. A local clock should maintain time values that are always monotonically increasing such that time does not run backward. In addition, each access to the clock should produce a unique value such that no two consecutive clock accesses will result in the same time value being returned. |

Downloaded from https://www.everyspec.com



### 3.2.2.4 Peripherals

This section provides rationale and requirements for selection of computer peripherals, special purpose devices, and other add-on computing capabilities. The guidance is as follows:

a. Interfaces to commonly available commercial information technology peripherals should utilize commercial mainstream industry standards.

b. Special purpose peripherals, such as NTDS interfaces, Inter-Range Instrumentation Group (IRIG) time boards, etc., should use interface protocols appropriate to the device type. Every effort should be made to provide a portable and maintainable interface mechanism.

### 3.2.2.5 Operating Systems

This section provides rationale and guidance for selection of operating systems in general and POSIX-compliant operating systems with suitable real-time features for mission-critical applications.

#### 3.2.2.5.1 Description

In the operating system domain, a number of features are required. The operating system standards chosen for OA-based applications should support a system of priorities to ensure those real-time processes and threads are dispatched when ready to run. Note that this does not mean that all operating systems used for OA must be fully real-time capable. However, it does mean that the standards family chosen does provide support for real-time. For those applications that do not need real-time features, an operating system that conforms to the standards family but does not provide real-time performance is acceptable. For real-time capable operating systems, support should be provided for kernel preemption in non-critical region processing. The operating system should also provide mechanisms to control priority inversion (e.g., priority inheritance protocols). Network drivers and other I/O operations should fall within the scope of the operating system real-time priority structure, and they should provide bounded delivery latencies under defined circumstances.

For federated architectures, multiple operating systems may be accommodated within a configuration based on individual needs. In this case, there is a fixed mapping of applications to processor/operating system combinations. However, for integrated ***total ship computing*** architectures (i.e. applications managed across a collection of pooled computers), it is simpler to employ either one operating system or a family of operating systems with binary compatibility. Note that interpreted or mobile code approaches such as the JVM also allow alternative choices of operating systems to be made. However, at present the inefficiencies of interpretive execution on JVMs do not provide sufficient performance for many real-time systems.

Downloaded from https://www.everyspec.com



#### 3.2.2.5.2 Guidance

Guidance for operating systems is as follows:

a. The operating system should provide standardized APIs. (The POSIX standards provide one mechanism for attaining this.)

b. In cases when use of POSIX-based operating system APIs is not possible, the operating system services should be provided through a portable standard operating systems adaptation layer.

c. All operating systems intended for use in mission-critical applications and support software should support the completion of hard and soft real-time deadlines required by the distributed tactical components.

d. For distributed components with real-time processing requirements, the operating system should provide standardized real-time scheduling facilities.

e. All operating systems should provide mechanisms for assigning application level priorities to processes and threads, for kernel preemption and reentrance, and for protection against priority inversions. Kernel-critical regions should be bounded with values not to exceed those established by application requirement analysis.

f. Support services include operating system services, file management services, standard time services, and other utilities required for proper system operation, including programming language facilities.

g. Software for support services should make use of industry standards and products to the maximum extent possible. Standardized file access mechanisms, particularly for shared network access such as the Network File System (NFS), should be provided as part of the support services.

h. At a minimum, operating systems should support system scope threads.

### 3.2.2.6 Adaptation Middleware

This section provides rationale and guidance for selection of adaptation middleware and products that export a POSIX-like operating system interface to application components. Adaptation middleware may also be used to export a common distribution middleware interface (see Section 3.2.2.7).

#### 3.2.2.6.1 Description

Adaptation middleware products export a standardized interface upward (i.e., from multiple-operating systems to a single API). Operating system adaptation layers export a standard set of APIs for operating system services to applications, while protecting them from differences in the underlying OS implementation, strongly promoting application portability.

Downloaded from https://www.everyspec.com



Other adaptation layers may be developed for middleware functions. These layers often provide a way of making non-standard products appear to be standards compliant. In this way, application portability is enhanced.

3.2.2.6.2 Guidance

Guidance for adaptation middleware and its use within the OA are as follows:

a. For applications that need to be portable to a variety of non-POSIX compliant operating systems, an operating system adaptation layer should be considered to achieve this portability. This adaptation layer should export an interface that is as close as possible syntactically and semantically to the standards-based APIs.

b. When commercial distribution middleware standards do not exist, appropriate adaptation layers should be developed that implement an interface representative of extant commercial products.

3.2.2.7 Distribution Middleware

This section provides rationale and guidance for use of distribution middleware in OA implementation.

3.2.2.7.1 Description

The term ***middleware*** is most often used to describe support software that facilitates interactions between major software components and masks differences in language, platform characteristics, message formats, communication protocols, data structures, and other factors. A number of communication middleware protocols have value in designing large, complex real-time systems. Furthermore, an evolution in recent years has occurred with respect to middleware semantics, driven in part by the widespread acceptance of the object-oriented paradigm among developers. Three classes of distribution middleware that provide capabilities relevant to OA interprocess communication functionality are as follows:

a. Distributed object computing

b. Publish/subscribe data distribution

c. Group-ordered communication

3.2.2.7.1.1 Distributed Object Computing

Within the domain of distributed processing, the distributed object computing model has emerged as a robust, standards-based technology area that is ready for use in real-time systems.

Downloaded from https://www.everyspec.com



At present, several major distributed object models exist. The following are examples of this type of model: Java's Remote Method Invocation (RMI) and EJB, Web Services with XML/Simple Object Access Protocol (SOAP), and the OMG's CORBA standard. Of these examples, CORBA offers the widest range of competitive vendor alternatives for a multi-language, multi-platform environment. In addition, OMG has recently released a real-time CORBA specification, and several conforming implementations are available.

CORBA provides location-transparent mechanisms to create object-oriented interfaces between distributed components. When designing a distributed-object model, one should consider the impact of the design on overall system performance. While creating network objects is rendered straightforward by CORBA, performance still depends on the network communication infrastructure.

The object-oriented approach encourages partitioning of a system into numerous objects, some of them potentially representing very small code segments. In some cases, system partitioning into numerous small objects provides a means of more precisely modeling the physical world; this point is reflected in the component partitioning section above. However, if carried to an extreme and if each object represents a separate entity from a communication perspective, a design predicated on this model may overload the network infrastructure. Consequently, achieving a balance is crucial in partitioning system requirements into code segments. If too fine-grained a partitioning is attempted, it may result in a set of components that are too small to operate efficiently. A useful concept is that of module binding strength. Data that is frequently referenced should perhaps be held locally, i.e., within a component. Data that is less frequently referenced, together with its associated processing, can be considered for placement in a separate partition (executable).

3.2.2.7.1.2 Publish/Subscribe Data Distribution

Within the domain of the publish/subscribe paradigm, messages are published to a multicast group for which membership is encapsulated by middleware. To become consumers of messages, clients register with the middleware using group names indicative of the type of data required. Servers publish data using the appropriate group names. The middleware uses the group names to ensure that data is routed to the appropriate clients, thus divorcing the server of knowledge regarding which clients have subscribed for data updates.

The publish/subscribe paradigm is well suited for location transparent distribution of continuously refreshed data such as distribution of track data to consumers as described above. In general, publish/subscribe performance tends to be better than other middleware approaches and, as such, is often the first choice for high data rate applications.

The publish/subscribe paradigm does have limitations that need to be accounted for when designing a distributed system. One limitation is the lack of sufficient functionality to fully support the replication mechanisms required to implement fault tolerance design patterns (see Section 3.2.1.9). This limitation can be overcome through the use of group-ordered communication protocols (see Section 3.2.2.7.1.3). Another limitation is the lack of a distributed object interface, such as that provided by CORBA.

Downloaded from https://www.everyspec.com



The OMG recently finalized the Data Distribution Service (DDS) for Real-time Systems standard. The DDS standard specifically addresses data-centric publish/subscribe (DCPS) functionality. Several readily available commercial publish/subscribe products provide similar services. At present no commercial products implement the DDS DCPS specification; however, several vendors have committed to this, and commercial implementations are expected within less than 1 year. In the interim, the best approach is probably development of an adaptive layer for publish/subscribe protocols. Such a layer would enable swapping out publish/subscribe products over time with a minimum of disruption to application code.

3.2.2.7.1.3 Group-Ordered Communications

Group-ordered communications middleware provides support for sophisticated reliable inter-process communications in distributed systems. Central to group communications are the process group semantics. In many respects, a process group is similar to a multicast group:

a. A process group is a set of communication endpoints.

b. Messages addressed to a process group are distributed to all members.

c. Distributed applications join a process group to exchange messages.

d. Multiple groups can be created within a distributed system.

e. Any application may simultaneously be a member of multiple process groups.

Essential group membership semantics differentiate a process group from a simple multicast group:

a. Each member of the process group has access to specific information about the other group members.

b. A group member can determine how many members are in the group, and perhaps even which specific processes executing on other nodes in the distributed system are members of the group.

c. Process group members can receive events indicating that a new member has joined the group or a current member has left the group.

These additional semantics support the development of application redundancy for dependability, fault tolerance, and load sharing. Additional features such as atomic multicast (also known as safe multicast), ordered messages, group-view changes that are ordered with respect to the message stream, and ordered state transfer provide significant benefit to the developers of reliable, distributed systems.

The work in fault-tolerant CORBA benefits greatly from a standardized group communication infrastructure. The OMG Real-time Special Interest Group (SIG) has published a Fault Tolerant CORBA specification. Development of fault-tolerant CORBA implementations,

Downloaded from https://www.everyspec.com



already available in limited numbers, will have a positive influence on product availability in this critical area. An OMG standard for Reliable Ordered Multi-Cast is currently being worked. This standard is expected to address some of the critical supporting communication characteristics found in group-ordered communication middleware.

#### 3.2.2.7.1.4 Data Marshalling

Another significant middleware consideration that contributes to successful construction of open systems is data marshalling and endian conversions. This is required due to the differences in binary data representation between computing systems. The most common representation differences arise from compiler conventions for bit ordering and hardware differences in byte addressing ("endianness".)

Using single bits for Boolean data is common in legacy systems because it was an efficient use of resources in now obsolete computers which were memory and bandwidth constrained. While constraints still exist for some systems, today's computer architectures have orders of magnitude greater supply of both memory and bandwidth. What's more, they also operate more efficiently (multi-layer caches, compiler optimizations) when data is at least a byte (addressable unit) in size. Different compilers number bits differently, even for the same computing platform (e.g. bit 1 of 32 for one compiler is bit 32 of 32 for another). This situation exists before the endian problem is applied, adding even more complexity. The most practical, interoperable solution is to make the smallest Boolean data unit the size of a computer's smallest addressable unit, a byte, and apply this rule to all data shared between computers. This removes the compiler as a source of variability in physical data representation and thus a hurdle to interoperability.

This leaves the issue of endianness. Some widely used processors use "little endian" byte ordering, and others use a "big endian" byte ordering. Little endian computers reverse or swap the order of bytes when data is received and written from memory to a processor register. Thus when in the register, the data looks the same as a "big endian" machine, but in memory the data appears as a mirror image. The swapping of bytes is also a function of the addressing mode used in accessing the data. Eight-bit addressable units (or bytes) such as characters are not swapped since single bytes have no order; multi-byte units such as integers and floating point numbers are swapped such that (0,1) becomes (1,0), and ,(0,1,2,3) become (3,2,1,0) and so on. Thus the data representation is a function of how the programmer defines the data to be shared, making this problem impossible to solve with a one-time hardware solution. It requires a priori knowledge of the format of data being shared between machines of different endianness.

If data is to be shared as messages, proper alignment of data on address boundaries is also a marshalling issue to be addressed. For example, a structure or record having a byte followed by a 32-bit value would have to be allocated by a compiler to have the byte placed on any address and the 32-bit value on address modulo 4. In order for this message to be interoperable, a programmer should supply spare fields in the structure to fill all available space and to ensure that 16-bit values fall on address modulo 2, 32-bit values on address modulo 4, and 64 bit values on address modulo 8. Failure to do so will cause compilers to add space to structures to place

Downloaded from https://www.everyspec.com



fields on addressable boundaries, and this spacing will be compiler-dependent, resulting in potential interoperability problems.

In general, the real-time computing industry has adopted a receiver-converts strategy. This allows all platforms to send in their native format, with a receiver being responsible for swapping data only if it is from an "other-endian" machine. It prevents one type of machine from being required to swap all incoming and outgoing messages, even if they were sending data to another "same-endian" machine, such as is provided by the External Data Representation (XDR) standard.

CORBA provides data marshalling through the use of its Interface Design Language (IDL) to specify interfaces and data types, and through Common Data Representation (CDR) to specify how that IDL is translated into a common format. The emerging OMG Data Distribution Service (DDS) standard mandates IDL use for the description of shared data. The mechanisms by which the data marshalling is provided are left to the individual DDS implementations. Where provided, it is recommended that the data marshalling mechanisms provided by standards-based middleware products, such as CORBA and DDS implementations, be used the extent possible to achieve data marshalling.

#### 3.2.2.7.2 Guidance

Guidance for distribution middleware and its use within the OA are as follows:

a. Distribution middleware should be provided to support the interactions among the distributed components, Commercial Item/Non-Developmental Item (CI/NDI) support services, standard services, resource management services, and legacy systems (via data brokers) that make up OA.

b. Distribution middleware software should be composed of COTS/NDI components when performance requirements permit. Some distribution middleware products may require extension to properly affect the interfaces, such as stubs and skeletons. Additionally, services may be built using the COTS/NDI components, such as adaptation layers or higher-level services. Note: Domain-specific components may be required to perform functions not included in distribution and adaptation middleware, (e.g., NTDS interfaces). These functions may be generically referred to as middleware, but since their counterparts do not exist in the commercial world, they are exempt from the COTS/NDI constraint.

c. Middleware technology for inter-component communication interfaces should be chosen in accord with the requirements of the interface. Whenever performance requirements and component design patterns permit, a distributed object interface should be used. High-volume distribution of continuously refreshed data, such as tracks or navigation information, should be accomplished via publish/subscribe methods. Middleware services for group-ordered communication should be used where state data coherency and ordering properties are required.

Downloaded from https://www.everyspec.com



d. Where available and otherwise suitable, middleware should be selected that provides explicit control over classes of service (i.e., QoS characteristics such as timeouts, latency, throughput, message delivery guarantees, etc.). An example of explicit QoS control would be a middleware product with an API allowing applications to specify a bounded latency associated with information transfer or distributed object invocation.

e. Middleware-based QoS features should not be used in ways that create system-level resource use conflicts among components. Such conflicts should be ameliorated through use of RM (see Section 3.2.2.8). Use of RM's QoS control permits resource tradeoffs to be managed explicitly at the system level rather than contained implicitly in the aggregated impact of component-level patterns of QoS invocations.

f. All middleware tools used in OA should provide an API that limits application exposure to operating system and protocol-specific functionality, product specific message formats, and data structures. Except when required to meet specific performance or functional requirements, middleware tools should be chosen that allow the encapsulation of such functionality, for example, through initialization calls or configuration files.

g. For middleware protocol types where standards are not yet available, a standard middleware adaptation layer should be developed that encompasses functionality typical of each relevant available class of products. This adaptation layer should export a uniform API to applications using such services. Such layers should be removed when suitable standards and standards-based products become available.

h. Data marshalling mechanisms provided by standards-based middleware should be leveraged to the extent practical to achieve interoperability between heterogeneous computers that have different endianness and/or different word sizes.

i. In some limited cases, the data marshalling mechanisms provided by the standards-based middleware may not be sufficient in terms of functionality or performance. In some very limited circumstances, such as for interfaces with legacy systems, standards-based middleware may not be employed. In those instances, care should be taken to properly align data to minimize the processing required to achieve data marshalling.

j. In those limited cases where data marshalling is required and standards-based middleware is not employed to implement it, a receiver-converts strategy should be adopted to minimize the overhead associated with reordering each message twice.

#### 3.2.2.8 Design Patterns, Frameworks, and Wrappers

This section provides rationale and guidance for the use of design patterns, frameworks, and wrappers in the OA product set.

Downloaded from https://www.everyspec.com



3.2.2.8.1 Description

According to Gamma, Helm, Johnson and Vlissides in their book, *Design Patterns: Elements of Reusable Object-Oriented Software*, a framework is "a set of cooperating classes that make up a reusable design for a specific class of software." The three key differences between design patterns and frameworks are:

a. Design patterns are more abstract than frameworks. Frameworks can be embodied in code, but only examples of patterns can be embodied in code.

b. Design patterns are smaller architectural elements than frameworks. A typical framework contains several design patterns, but the reverse is never true.

c. Design patterns are less specialized than frameworks. Frameworks always have a particular application domain.

A framework may provide a reusable design for all or merely a part of an application. For example, a fault-tolerance framework may provide the structure and interaction of classes that support building a fault-tolerant application. Fault tolerance is an issue that is pervasive, and a fault-tolerance framework would affect the entire application. A tasking framework may provide the set of classes and class interactions that support a particular concurrency model. Although the use of concurrency is extremely important in the component architecture, the model of concurrency can vary across different parts of the component. Care should be exercised during component design to ensure that instances of framework use do not conflict with instances where underlying services are invoked directly. The use of frameworks provides structuring mechanisms, design patterns, and code reuse opportunities that can be leveraged by application developers to reduce development time and life-cycle costs.

A ***wrapper*** is software that is used to insulate applications from the API of another set of software by exporting a different API. A wrapper may be thin or thick. A thin wrapper does a minimal translation of one API to another. A thick wrapper provides significant additional or modified functionality over that implemented by the software being wrapped.

In some cases, frameworks are used as thick wrappers around COTS products. This is done primarily to protect the software from changes in the API of the underlying implementation and/or to limit the features of the underlying product that can be used by the application developer. Under some circumstances, this can interfere with interoperability between components developed with the wrapper and those developed without the wrapper. This can happen both because of extra capability introduced by the wrapper and also because of those features excluded by the wrapper that may be used by other components.

Wrappers are not inherently bad if you are trying to shield developers against **significant** changes in APIs that are under development. However, care must be taken when deciding to use them. Otherwise, the OA goals of software reuse and interoperability may not be achieved across the Navy Enterprise. Therefore, except in a few, limited cases, it is recommended that the APIs specified in the standards be used directly rather than using wrappers.

Downloaded from https://www.everyspec.com



#### 3.2.2.8.2 Guidance

Guidance for the use of frameworks and wrappers within the OA is as follows:

a. Frameworks should be leveraged to facilitate application structure commonality and code reuse where appropriate frameworks exist and have been identified as supplying needed capabilities (e.g., fault tolerance, concurrency, database access, and instrumentation).

b. Frameworks should be well documented, extensible, and provide the ability to easily add new services.

c. Frameworks and wrappers should not be used to limit use of an OA mandated, standards-based API. For standards that are emerging or are otherwise likely to change significantly, the engineering processes established by the OA Enterprise Team (OAET) should be followed to determine the appropriateness and method of wrapping a standards-based API.

d. Where wrappers for standards-based APIs are used, they should adhere to the following: (1) the abstraction layers should not introduce any interoperability issues with applications that don't use those wrappers; and (2) the wrappers should be ultra-thin and be as close as possible syntactically and semantically to the standards-based APIs.

e. In cases where wrappers or frameworks would lead to interoperability issues with other applications but whose use would provide significant value to the Navy Enterprise, these issues should be resolved in accordance with the engineering processes established by the OA Enterprise Team (OAET).

### 3.2.2.9 Resource Management (RM)

This section provides rationale and guidance for management of computing resources and dynamic allocation of components to the pool of virtually homogeneous computers that constitute the OA equipment suite.

#### 3.2.2.9.1 Description

The purpose of RM is to provide run-time control of the computing resources in the system. In providing this capability, two levels of control are possible: static and dynamic. With static control, configurations are fixed at design time and the run-time resource manager uses these configurations to perform startup, health monitoring, reconfiguration, and shutdown.

Similar to static control, a dynamic control performs startup, health monitoring, reconfiguration, and shutdown. However, with dynamic control, the resource manager uses application and system performance requirements to derive and execute appropriate system reconfigurations at runtime, in addition to supporting pre-defined configurations. Dynamic reconfiguration of this nature enables the resource manager to ensure that the system continuously operates to provide a defined level of QoS.

Downloaded from https://www.everyspec.com



Figure 3-9 provides an RM architecture that features both static and dynamic capabilities.

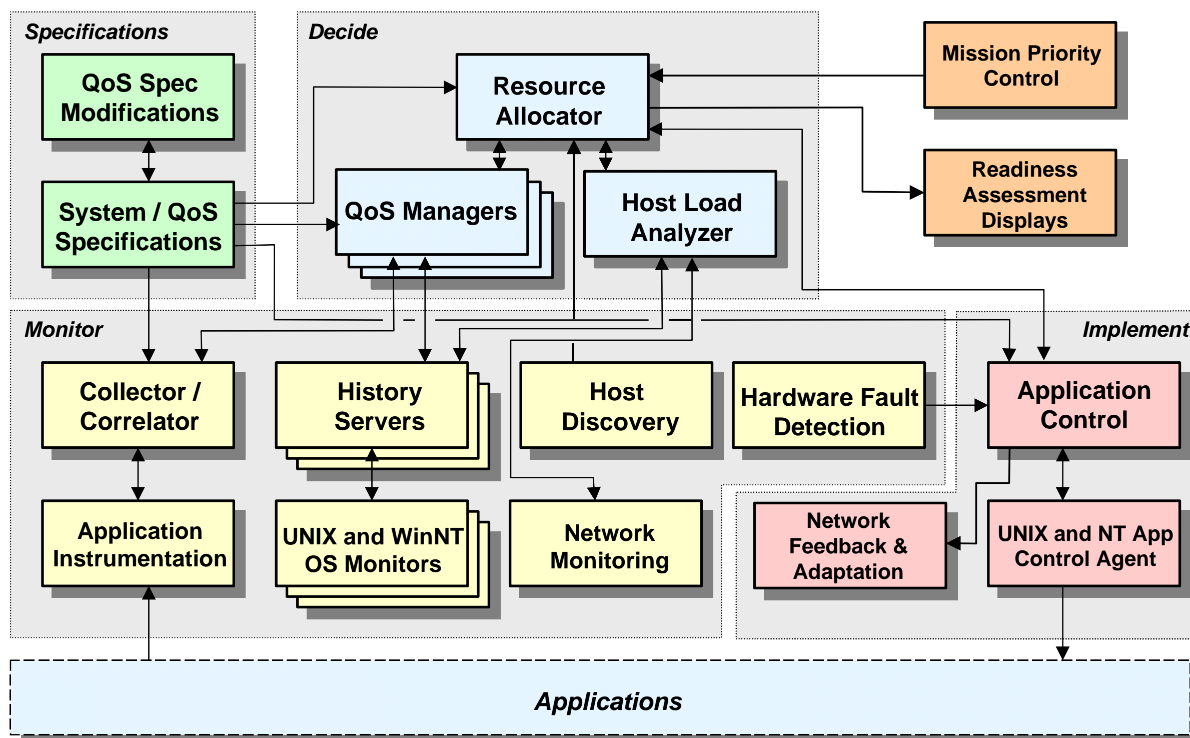


**Figure 3-9. Resource Management Architecture**

RM components provide monitoring and control of network, operating system, middleware, and application resources within the distributed computing environment. Resource and QoS management services are intended to provide the ability to dynamically configure and reconfigure hardware, operating system, and application components to ensure that desired or negotiated application, path, and mission-oriented QoS requirements are met at runtime.

To be most useful and least intrusive, the RM capability should introduce little or no impact on basic application design choices. It should not, for instance, arbitrarily force selection of single-threaded vs. multi-threaded applications, periodic processing vs. event driven, blocking vs. non-blocking I/O, or the choice of middleware approach.

The high-level architecture of the RM capability is a feedback control loop consisting of monitoring, decision-making, and control (configuration adjustment) components. System designers specify the required performance of each component. Application monitoring instrumentation measures component performance with respect to specified requirements. When a violation of required performance takes place (or is projected to take place), the RM decision-making capability utilizes status, load, performance, and fault metrics gathered by components that monitor processor and network load. The decision-making capability then determines if,

Downloaded from https://www.everyspec.com



when, and how reconfiguration actions should be carried out. The resource control components then map RM control orders to the low-level control actions that should be carried out to implement the desired reconfiguration actions.

#### 3.2.2.9.2 Guidance

RM monitors system performance to assure that the system meets the mission priorities established by ship's command. RM supports fault detection and recovery in cooperation with application components. Guidance for RM, both static and dynamic, is given below:

a. RM should have the capability to determine and control system configuration, load and initialize application components, allocate computing resources, and monitor and control the use of resources.

b. RM should have both an automatic mode of operation and an operator-controlled mode of operation.

c. RM should provide a capability for engineers and operators to statically define and activate system configurations, including processor assignment, communication routing, and other factors. A sufficient number of alternate static configurations should be established to meet availability requirements via automatic- or operator-initiated reconfiguration in response to system faults and loss of resources.

d. RM should have a capability for engineers to define QoS and performance requirements for application components. The QoS specification language should be independent of programming language and application domain. QoS specification capabilities should include the ability to define a number of processing metrics, including response times, periodic processing intervals and durations, multi-component end-to-end latencies, load-balancing metrics and failure conditions, and recovery actions.

e. RM should provide a capability to assign QoS requirements dependent on defined system modes of operation (e.g., non-combat operations vs. combat operations). A capability should be provided whereby system engineers can define required system configuration reallocations in accord with the new mode of operation.

f. RM should coordinate the recovery actions of all system components (see also Section 3.2.2.11). RM should provide services in support of fault detection and recovery. Faults should be detected in processors, peripherals, network components, middleware, and application computer programs.

g. RM should interoperate with component recovery mechanisms, i.e., time-critical applications should be designed to perform quick response failure recovery without intervention by RM. In performing the quick response recovery, if a time-critical component falls below an acceptable level of redundancy, RM should operate to restore the lost redundancy after the fact, subject to resource availability. Non-time-critical applications should be designed so that they may be restarted under RM control.

Downloaded from https://www.everyspec.com



h. RM should have the capability to continuously monitor mission and system priorities, application performance, and the status and utilization of system resources (see application and system resource instrumentation guidance in Section 3.2.2.10). When placed in the automatic mode of operation, RM should allocate resources at runtime to tasks without operator intervention as required for system optimization, tactical load balancing, and fault recovery or when faults are anticipated.

i. A dynamic RM capability should support the definition of end-to-end QoS performance requirements across multiple components and should monitor performance, latency, and load along these multi-component paths. Upon detection of a violation of the specified QoS parameter, RM should provide a capability to reallocate components for improved performance and to initiate load management via activation of multiple-replicated copies of a component.

j. A dynamic RM capability should have the capability to adjust internal network characteristics (e.g., reserved bandwidth, routing, etc.) in order to ensure tactical performance.

k. RM should provide repeatable and testable mechanisms to ensure closed-loop stability with respect to resource allocation operations.

l. To ensure component compatibility with RM, component developers should design certain features into system components. These features include the following:

1. Specification of application component configuration and performance requirements via a QoS specification language
2. Initial and statically defined system configuration
3. Application and support component startup and shutdown services
4. Operating system-level applications monitoring
5. Middleware-level application monitoring
6. Application instrumentation and performance monitoring (see Section 3.2.3.10)
7. Direct RM-to-application control capabilities

m. RM should provide operator display screens for both engineering activities and for deployment operations. Deployment displays should be limited in complexity and supported with automation to reduce required crew workload.

n. RM performance should be sufficient to support all specified OA tactical performance requirements. Detailed RM performance requirements should be defined during OA development.

Downloaded from https://www.everyspec.com



o. RM should include functionality to monitor and display data internal to OACE components, including RM itself. Human interaction with OACE via RM should include, but not be limited to, mode selection, operability tests, trouble-shooting, and maintenance. Maintenance actions should include modification of the system configuration following a change to a processor or network component and input of static component allocation data during an engineering maintenance activity.

p. RM software should make use of commercial standards and NDI products to the maximum extent possible.

#### 3.2.2.10 Instrumentation, Recording, and Assessment

This section provides rationale and guidance for instrumentation, recording and assessment services of OA components, processors, networks, etc. in support of component testing, certification and RM.

##### 3.2.2.10.1 Description

Computer systems have long been instrumented, both during development and during deployment. The benefits provided to programmers and engineers for debug and testing during development are well established. The benefits during deployment are equally valuable in that recorded data may be used to diagnose anomalous situations experienced during operation.

With the advent of RM, the benefits of instrumentation take on an entirely new dimension; i.e., permitting real-time adaptation of the system configuration resulting from system health assessments based on instrumentation data. Two types of information are required for dynamic resource allocation: (1) application performance information and (2) computing resource consumption information. Resource consumption information is further subdivided into computer resource use and network resource use.

Application performance information is required in determining whether applications are meeting their performance requirements. Examples of this include periodic processing activation times and durations, event latencies, and workload metrics (e.g., number of tracks processed). Status information is needed to determine the health of the application and whether the application is configured properly. Examples include initialization status, internal view of interface status, etc. Error condition information allows the reporting of exceptions and problems detected internally by the application. Examples include detection of corrupted data, faults on interfaces, middleware status or response problems, system call response errors, etc.

Computing resource consumption information is measured at the CPU level and at the network level. Resource use data provides the information needed to identify available resources that can be assigned when dynamically changing system configuration.

In fulfilling these instrumentation requirements, a set of instrumentation capabilities should be adopted that is consistent between the data collection needs and the RM needs.

Downloaded from https://www.everyspec.com



3.2.2.10.2 Guidance

Guidance for instrumentation for OA is as follows:

a. Instrumentation should be used to collect run-time data and provide system performance feedback to RM monitoring capabilities. Services include data extraction, collection, reduction, analysis, and presentation.

b. The instrumentation capability should provide services and an API for collection of application-defined events and performance metrics. Information collected should include but not be limited to application performance and status data, application state and state changes, internal processing loads, occurrence of key events, internally detected error condition, and internal view of interface status.

c. The instrumentation capability should provide services for measuring processor loading, memory use, and other computing system metrics from all processors. Information collected should be at both the processor level and the process level. Where possible, thread information should also be collected. Information monitored should include:

1. Operating system version and machine configuration
2. CPU configuration, status, and utilization
3. Memory configuration and usage
4. Network configuration, status, and utilization
5. File system configuration, status, and utilization
6. Process status, including CPU, memory, network, and file system utilization for each process
7. Network load as seen by a particular host
8. Information concerning any remotely mounted file systems (e.g., NFS).

d. The instrumentation capability should provide services for measuring network metrics. Metrics should be collected at both the edge device connection level and on any internal switches, routers, and their associated links. Metrics for each link should include but are not limited to bandwidth, throughput, and latency. In addition, each host's view of its network loading should be collected (see item c(7) above).

e. The instrumentation collection function should be implemented as a server that communicates with a scalable number of clients. Both the server and client processes should have minimal performance impact on the system under test.

Downloaded from https://www.everyspec.com



f. Three levels of instrumentation should be defined: developmental, operational (in support of RM, see Section 3.2.2.8), and war diary. OA and resource allocation design should be such that operational and war diary instrumentation may enabled at all times without adverse impact on tactical performance.

g. The server and the clients should not be developed with dependencies upon specific hardware items. The server should not limit the client's flexibility to manage the client's data buffer pools, control the events to monitor, specify data item formats, or group events into views of specific interest.

h. Details of services provided for monitoring and variations of collection software should be adapted to the available services where necessary.

#### 3.2.2.11 Failure Management

This section provides rationale and guidance for integrated failure management in the OACE (i.e., integration of fault detection, localization and recovery actions across all applications, and supporting technologies).

##### 3.2.2.11.1 Description

Failure management is the combined capabilities and interactions of all components of the architecture in support of fault tolerance for the overall system. It encompasses operating system services, middleware and associated system composition services, information transfer services, RM services, as well as application program design. This capability is particularly important since many component technologies present in the commercial marketplace are self-contained with respect to failure detection and fault tolerance.

##### 3.2.2.11.2 Guidance

The following constitutes the failure management guidance for OACE components:

a. The failure management capability should operate in an integrated fashion across all of OACE components. The failure management capability should be designed in such a manner to ensure that detection and recovery actions should not cause erroneous system operation. This guidance should be relaxed in the case of network segmentation.

b. For COTS products that provide adequate access to internals, externally observable behavior should be monitored to the maximum extent practicable for the purpose of detecting and diagnosing faults and initiating recovery actions.

c. Failure management should consist of multiple tiers of recovery actions. The first tier should be built-in component failure recovery, whereby components initiate fail-over to replicated copies. The second tier should be reinstantiation of lost capability under control of RM. This would normally be accomplished by means of RM reload and restart of lost

Downloaded from https://www.everyspec.com



components. In some cases where timeliness requirements permit, first-tier failure recovery via application intrinsic replication may not be required.

d. OACE external interfaces to legacy components should utilize redundant physical connections between OACE communication nodes and external systems. The interface between the fully compliant OACE implementation and the associated warfighting system elements should provide for data flow and control behavior between the respective OACE subsystem and the rest of the weapon system.

3.2.2.12 Information Assurance

This section provides rationale and guidance for incorporation of information assurance features into OA and the OACE. It also addresses the interaction between information assurance technology and the resource manager.

3.2.2.12.1 Description

The overall objective of information assurance service is to ensure that mission-critical applications have access to the necessary computing resources and data needed to complete the current mission of the ship successfully despite external attacks against the system. Non-mission-critical systems also have a need for information assurance. Traditional security services include authentication, confidentiality, and access control. Specifically, computationally expensive and complex encryption technologies that provide authentication and confidentiality services may not be needed inside the NWS enclave. These technologies may be needed on the “edges” of the enclave in order to support the encryption requirements of external systems. The technology chosen to implement OA security measures should be readily certifiable.

The shift to CI/NDI components presents security threats that were not present in previous NWS baselines. The security vulnerabilities of the commercial world, such as denial of service attacks and computer viruses, are now a concern to the NWS since the common attacks found in the commercial world apply to many of the systems that will be used to host components of the NWS.

A defense-in-depth strategy should be considered in the design of OA. Firewalls, intrusion detection devices, demilitarized zones (DMZs), and anti-virus software should be considered at points where the NWS communicates with external systems (either shipboard or land-based), as well as within the NWS enclave.

3.2.2.12.2 Guidance

The following constitutes the guidance for information security interaction with RM:

a. The information assurance technology should have the ability to prevent known information warfare attacks.

Downloaded from https://www.everyspec.com



b. The system should employ available information assurance technology to detect and counter information warfare attacks.

c. The information assurance technology should alert appropriate personnel and/or the resource manager when an information warfare attack has been detected.

d. The information assurance technology should notify and, if possible, recommend a response to appropriate personnel, security services, and/or the resource manager when an information warfare attack has been detected.

e. The information assurance technology should be capable of implementing an automated response to an information warfare attack independent of personnel or the resource manager.

3.2.2.13 Time Service

This section provides rationale and guidance for providing a standard OA time distribution and synchronization service via the widely used commercial NTP.

3.2.2.13.1 Description

The nature of real-time processing is that most, if not all, computers running real-time applications should be time synchronized to a fairly precise tolerance. This is true due to the coordination that should occur among the various weapon system components in the solution of problems involving physics and kinematics.

Two types of solutions are available to achieve time synchronization: (1) dedicated hardware (e.g., IRIG-B), and (2) software-based clock synchronization algorithms (e.g., NTP). Either approach is technically acceptable; however, the NTP approach has the advantage that it does not require additional equipment. NTP is a protocol design to synchronize the clocks of a computer over a network. The NTP approach, which is inherently distributed and decentralized, is fairly robust.

Studies have shown that, with proper engineering of the time synchronization subnet, a one-millisecond time synchronization requirement can be met with commercial computers and networks using NTP. The core of NTP is a basic feedback loop controlled by a number of parameters including minimum and maximum polling interval. This feedback loop should be engineered on a system-by-system basis to achieve the required level of performance. This performance may be characterized by a number of factors including: (1) the level of synchronization required and (2) the time required to reach that level of synchronization within a defined standard deviation after a perturbation event (e.g., startup) has occurred.

This second parameter is sometime referred to as the NTP settling time and represents the time required for the feedback loop to stabilize after reset or loss of a clock. This settling time is dependent on a number of factors, including computer and operating system types, network characteristics, and internal NTP parameter settings. Any decision concerning NTP use in the

Downloaded from https://www.everyspec.com



OA implementation should be based on engineering analysis of likely NTP settling time and synchronization accuracy compared to NWS operational requirements.

3.2.2.13.2 Guidance

Guidance for the OACE time service is as follows:

a. Support for a standardized combat system-wide single time reference should be implemented in OA.

b. When performance characteristics permit, NTP should be considered for use as the standard time reference.

c. An NTP synchronization subnet should be engineered for optimal performance and fault tolerance including the use of multiple time sources and adequate clock implementations to support fine tuning of local clocks.

d. Existing approaches to distributing time over the Internet are vulnerable to external attack and tampering, as these do not take advantage of any authentication or cryptographic methods.

e. If a synthetic time service is used for embedded simulation and training operation, it should not interfere with or result in erroneous warfighting system operation. The artificial nature of this capability should be obvious to warfighting system operators and should be capable of rapid deactivation.

3.2.2.14 Programming Language Facilities

This section provides rationale and guidance for incorporation of programming language support facilities into the tool set for the OA development effort. The following programming languages are expected to be supported within OA applications:

a. C/C++

b. Java

c. Ada

The guidance for programming language facilities is as follows:

a. Programming language facilities should support Procedural and Object-Oriented Distributed programming.

b. All OACE infrastructure services should allow for bindings to all high-level languages used within the OA set of components.

c. Standard reuse libraries should be used where appropriate and effective.

Downloaded from https://www.everyspec.com



d. Common coding and naming conventions should be applied throughout all OA software per recognized industry and development organization standards.

e. Programming language facilities should support Open Standards in order to interconnect distributed applications and disparate platforms.

f. Programming language facilities should be able to support the implementation of high-volume, scalable, distributed environments.

g. Since adequate documentation is crucial for both product maintainability and component reusability, some programming language facilities (e.g., Java) provide utilities that facilitate creation of documentation in the course of implementing the source. To the extent that such utilities are available, their use should be considered.

### 3.2.3 System Composition

System composition is an aggregate term that describes synthesis of application components from available technologies, tools, models, methods, and mechanisms. It consists predominantly of utilizing support software, or ***middleware***, to construct application computer programs in accord with design patterns such as elaborated in Section 3.2.1.

Where middleware standards exist, products conforming to these standards should be used to construct application components. When standards do not yet exist, effort should be made during product selection to pick products that have widespread utilization and a commercial support structure.

For each application component, early design activities should focus on determining the functional capabilities and boundaries of each component. These activities include deciding what design pattern best fits the requirements, determining appropriate mechanisms for fault tolerance and scalability, identifying the interfaces with other components, and deciding on the appropriate technologies to utilize in effecting interactions between the component and other components.

### 3.2.4 Displays

This section provides rationale and guidance for designing the OA display capability. It is recommended that the display architecture be designed so that any operator may use any console to accomplish any assigned mission.

#### 3.2.4.1 Description

In legacy system implementations, display consoles were often unique to the operator they supported. In some instances, standardized consoles were customized through use of special purpose add-on devices and interfaces. This design approach was justifiable given the point-to-point interfaces and limitations in the graphics hardware and graphical support tools.

Downloaded from https://www.everyspec.com



However, modern network and graphics technology provide an opportunity for a much more standardized and, therefore, logistically appealing solution in which any console may be used to perform the entire range of actions needed to support any operator mode or task. This approach not only provides logistics benefits, but it also promotes software reuse. Furthermore, the resultant equipment configuration is inherently more resilient in battle.

#### 3.2.4.2 Guidance

The following provides the guidance for display console organization with respect to generality of use.

a. Display and operator support software should conform to the guidance contained in this document.

b. Except for unique and/or special purpose requirements, the OA system should be designed such that each console supports the entire range of actions that need to be executed by each operator.

### 3.2.5 System Test and Certification

This section provides rationale and guidelines for certifying systems based on the principles of dynamic resource allocation.

#### 3.2.5.1 Description

Certification of traditionally designed systems is well understood. It consists primarily of functional and performance testing against specified system requirements, usually organized according to test cases and scenarios. However, systems employing dynamic management of resources, particularly those involving a degree of heterogeneity in the equipment configuration, require a more complex approach. Not only should the allocation mechanisms be tested, but also differences in equipment and configurations should be reconciled. This gives rise to additional requirements for certification of such systems.

In describing the guidance for certification of dynamically allocated systems, the term virtual homogeneity is used to describe a property of the computer and operating system that is required to support the certification process. This property states that each computer within a defined class of computers should perform substantially like every other computer in the class. Executables compiled for each computer in the class should yield operationally and statistically equivalent answers for an appropriate abstraction hierarchy (i.e., the granularity of system performance that is of interest to a particular user—in this case, the warfighting system's operators and sensor and weapons capabilities). Thus, virtual homogeneity is preserved when the variation in component output caused by any difference in computing resources is operationally negligible, a criterion that can be defined and tested.

Downloaded from https://www.everyspec.com



Multiple classes of computers may be required for the foreseeable future to handle efficiently differing warfighting system requirements. For example, the type of computers and operating systems most effective for real-time weapons control may not be fully suitable for providing display capabilities and vice versa. However, several computers and operating systems may be fully capable of performing weapons control functions. The various computers falling in this latter category form an equivalence *class*, and all products in this class are candidates for selection as OA computer resources.

When combined with the pool of computers concept discussed in Section 3.2.2.8 on RM, this concept leads to multiple pools encompassing the equivalence classes needed to fully implement NWS functionality. This constraint implies that several conditions should be met. First, a small number of computing resource types is preferable. Identical resources meet the virtual homogeneity criterion by definition, whereas a high degree of variation among processing resources will likely increase testing time and cost. In addition, subsetting the total resource pool into classes of resources may result in a significant constraint on the RM.

If such constraints are needed, they could take the form of a tree of subsystems or aggregations of components. Each aggregation exists in only a manageable number of configurations. A case can be made that there are no (or negligible) interactions between the choices of configurations of sibling subsystems. Finally, all computers should provide a minimum level of performance (***unit of computing***) sufficient to ensure that system and component performance requirements are met.

Note that logistics considerations weigh in favor of limited numbers of classes and real homogeneity within classes.

Finally, any configuration that RM creates in the resource allocation process should be schedulable. Thus, part of the certification process should schedule issues. Also, requiring consideration during certification are stability considerations. In effect, RM's allocation should not "trash" or produce a series of configuration changes sensitive to relatively small changes in the external environment. These considerations will likely require that support for the schedule and stability is incorporated directly into RM itself.

3.2.5.2 Guidance

The guidance for system testing and certification of OA is as follows:

a. Traditional functional and performance testing should be done on tactical and support components.

b. Functional and performance testing should be done on the RM capability.

c. Testing should be done to verify that system resources constitute "virtually homogeneous" pools.

Downloaded from https://www.everyspec.com



d. If virtual homogeneity is not ensured across computing resource variants, additional testing is required to verify that resource allocation configuration constraints reflective of this are applied correctly.

e. Verification of scheduled analysis methods, stability analysis models, and capabilities should be built into the resource manager's allocation methods.

f. All OACE components should have a regression test suite.

### 3.2.6 Selection of Standards

A major advantage of the open-system approach is that obsolete or failed components can be replaced quickly and easily. So that this is accomplished effectively, it is highly advantageous to pick products in the commercial mainstream. Such products are far more likely to be readily available and interchangeable than niche market products.

The OACE should use industry standards to take maximum advantage of commercial mainstream technologies. Types of standards are listed in Table 3-4 below. Note that wherever possible, OACE-compliant system components should adhere to appropriate industry standards. There should be the minimum possible dependence of system components on vendor proprietary features. When features for which standards exist are employed, those standards should be employed. OACE components should be built and/or selected in a manner to foster portability across a variety of lower level technology base components. Metrics for measurement should be identified, and goal thresholds should be established for the metrics.

Downloaded from https://www.everyspec.com



**Table 3-4. Types of Standards**

| STANDARD TYPE | DEFINITION |
|---|---|
| Formal Standard | Standards that are developed, approved, and maintained by a formally recognized standards committee by a consensus process. Examples: POSIX, CORBA. |
| Industry Standard | An industry standard is a formal or *de facto* standard that has been widely accepted and broadly implemented. A sufficient number of sources for products or services adhering to the standard are available to provide reasonable vendor independence. Examples: American National Standards Institute (ANSI) C, Small Computer System Interface (SCSI), Parallel Computer Interface (PCI), and Java. |
| *Defacto* Standard | Standards that may have begun as proprietary but have achieved widespread acceptance by users. These standards are typically licensed to other vendors to produce compatible products. The vendor that produced the standard maintains the standard. Example: PCI. |
| Proprietary Standard | Standards that have been published and are publicly available but the number of vendor implementations is limited (usually one). Example: Visual Basic. OACE should avoid the use of proprietary standards unless no other products will provide required functionality or performance. |

#### 3.2.6.1 Applicable Standards

A number of standards are sufficiently mature to merit their use in OA. These are invoked in detail in the companion volume to this document, *Open Architecture Computing Environment Technologies and Standards*. Traceability of the OA standards to the Joint Technical Architecture (JTA) is maintained and the relationship of OA standards to JTA is discussed in the just-mentioned standards document. Generally, the families of standards invoked for OA are as follows:

a. TIA physical media standards

b. IETF network standards

c. POSIX operating system standards

d. OMG middleware standards

Downloaded from https://www.everyspec.com



### 3.2.6.2 Open-Source Products

The open-source model provides an additional source of technology components for construction of the OACE. Open-source products have the advantage of permitting the developer to review source code for quality control purposes and, if needed, add critical features vital to system operation. Some open-source products do lack a commercial support base; however, many have a commercial support structure. Open-source products should be considered when there are economic reasons for doing so and/or when such products provide a needed capability that is not currently obtainable by any other means. The guidance for open-source products is as follows:

a. Wherever possible, open-source products should have a commercial product support base.

b. Wherever possible, open-source products should have a broad user base.

c. Licensing obligations for use of open-source products should not in any case obligate the Navy or its contractors to return code to the open-source community, nor should use of open-source product compromise the integrity of the Navy's warfighting system acquisition process.

Downloaded from https://www.everyspec.com



# APPENDIX A

# ACRONYMS

Downloaded from https://www.everyspec.com



| Acronym | Definition |
| --- | --- |
| ANSI | American National Standards Institute |
| API | Applications Programmer Interface |
| ASN (RDA) | Assistant Secretary of the Navy (Research, Development, and Acquisition) |
| ATM | Asynchronous Transfer Mode |
| BF | Battle Force |
| C3I | Command, Control, Communications, and Intelligence |
| CCM | CORBA Component Model |
| CDR | Common Data Representation |
| CI | Commercial Item |
| CIO | Chief Information Officer |
| CNO | Chief of Naval Operations |
| COE | Common Operating Environment |
| COEA | Cost and Operational Effectiveness Analysis |
| CORBA | Common Object Request Broker Architecture |
| COTS | Commercial Off-the-Shelf Technology Systems |
| CPU | Central Processing Unit |
| DARPA | Defense Advanced Research Projects Agency |
| DCPS | Data-Centric Publish/Subscribe |
| DCOM | Distributed Component Object Model |
| DDS | Data Distribution Service |
| DISA | Defense Information Systems Agency |
| DMZ | Demilitarized Zone |
| DOC | Distributed Object Computing |
| DoD | Department of Defense |
| DoDD | Department of Defense Directive |
| DoN | Department of the Navy |

Downloaded from https://www.everyspec.com



| Acronym | Definition |
|---|---|
| EDM | Engineering Development Model |
| EJB | Enterprise Java Bean |
| FIFO | First-In/First-Out |
| FTP | File Transfer Protocol |
| HiPer-D | High-Performance Distributed Computing |
| HM&E | Hull, Mechanical, and Electrical |
| HTTP | Hyper Text Transfer Protocol |
| IDE | Integrated Development Environment |
| IDL | Interface Design Language |
| IEC | International Electrotechnical Commission |
| IEEE | Institute of Electrical and Electronics Engineers |
| IETF | Internet Engineering Task Force |
| IGMP | Internet Group Management Protocol |
| I/O | Input/Output |
| IOP | I/O Processor |
| IP | Internet Protocol |
| IPT | Integrated Product Team |
| IRIG | Inter-Range Instrumentation Group |
| ISO | International Standardization Organization |
| ITIA | Information Technology Infrastructure Assurance |
| ITSG | Information Technology Systems Guidance |
| J2EE | Java 2 Enterprise Edition |
| JTA | Joint Technical Architecture |
| JVM | Java Virtual Machine |
| LAN | Local Area Network |
| LTR | Letter |

Downloaded from https://www.everyspec.com



| Acronym | Definition |
|---|---|
| MAIS | Major Automated Information Program |
| MDAP | Major Defense Acquisition Program |
| NDI | Non-Developmental Item |
| NFS | Network File System |
| NIC | Network Interface Card |
| NSWCDD | Naval Surface Warfare Center, Dahlgren Division |
| NTDS | Naval Tactical Data System |
| NTP | Network Time Protocol |
| NWS | Naval Warfare System |
| OA | Open Architecture |
| OACE | Open Architecture Computing Environment |
| OAET | Open Architecture Enterprise Team |
| OATA | Open Architecture Technical Architecture |
| OATF | Open Architecture Test Facility |
| OMG | Object Management Group |
| OO | Object-Oriented |
| OOP | Object-Oriented Programming |
| ORB | Object Request Broker |
| ORD | Operational Requirements Document |
| OSJTF | Open Systems Joint Task Force |
| PC | Personal Computer |
| PCI | Parallel Computer Interface |
| PEO | Program Executive Office |
| PEO IWS | Program Executive Office for Integrated Warfare Systems |
| PIDS | Prime Item Development Specifications |
| POR | Program of Record |

Downloaded from https://www.everyspec.com



| **Acronym** | **Definition** |
|---|---|
| POSIX | Portable Operating System Interface Standard |
| QoS | Quality of Service |
| R&D | Research and Development |
| RACI | Responsible Accountable Consulted Informed |
| RD&A | Research, Development and Acquisition |
| RDBMS | Relational Data Base Management System |
| RFP | Request For Proposal |
| RM | Resource Management |
| RMI | Remote Method Invocation |
| RPC | Remote Procedure Call |
| RSVP | Resource Reservation Protocol |
| SBC | Single-Board Computer |
| SCIB | Ships Characteristics Improvement Board |
| SCSI | Small Computer System Interface |
| SECNAVINST | Secretary of the Navy Instruction |
| SIG | Special Interest Group |
| SMP | Symmetric Multi-Processor |
| SOAP | Simple Object Access Protocol |
| SSDS | Ship Self Defense System |
| TCP | Transport Control Protocol |
| TCP/IP | Transport Control Protocol/Internet Protocol |
| TIA | Telecommunications Industry Association |
| TSC | Theater Surface Combatants |
| UDP | User Datagram Protocol |
| UDP/IP | User Datagram Protocol/Internet Protocol |
| UI | User Interface |

Downloaded from https://www.everyspec.com



| Acronym | Definition |
| --- | --- |
| UML | Unified Modeling Language |
| VME | Virtual Micro-bus European |
| W3C | World Wide Web Consortium |
| WSDL | Web Services Description Language |
| XDR | External Data Representation |
| XML | [E]Xtensible Markup Language |