MP 00W0000088

MITRE PRODUCT

A Guide to Understanding Emerging Interoperability Technologies

July 2000

Terry Bollinger

Sponsor:	ONR	**Contract No.:**	DAAB07-99-C-C201
Dept. No.:	W09T	**Project No.:**	0700V13A-AA

The views, opinions and/or findings contained in this report are those of The MITRE Corporation and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

Approved for public release; distribution unlimited.

MITRE

Washington C3 Center
McLean, Virginia

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **JUL 2000**	2. REPORT TYPE	3. DATES COVERED **00-07-2000 to 00-07-2000**

4. TITLE AND SUBTITLE **A Guide to Understanding Emerging Interoperability Technologies**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **MITRE Corporation,7515 Colshire Drive,McLean,VA,22102-7539**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES **86**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

MITRE Department Approval: ________________________________

Gary R. Brisbois

MITRE Project Approval: ________________________________

Daniel J. Lowen

Abstract

Interoperability is the ability to use resources from diverse origins as if they had been designed as parts of a single system. Over time, individual interoperability problems tend to disappear as the resources involved literally become part of one system through integration and standardization, but the overall problem of interoperability itself never disappears. Instead, it simply moves up to a new level of complexity that accepts earlier integrations as a given. Interoperability is especially critical for military systems, where international politics can lead to abrupt realignments where yesterday's foe becomes today's coalition partner. This report on interoperability has five sections. The first section is an introduction to the interoperability problem, and the second section describes fundamental interoperability concepts and develops a terminology for describing interoperability issues and needs. The second section also addresses the vital issue of interoperability security. The third section is about the processes by which interoperability technologies are standardized, including comparisons of the interoperability benefits of different processes. The fourth section is an overview of a number of emerging information technologies relevant to interoperability, and the fifth section suggests opportunities for further action.

KEYWORDS: interoperability, security, standardization, processes

Acknowledgments

I would like to thank the following MITRE people for their excellent review comments and overall assistance in creating this document: Dan Lowen, Larry Stine, Chuck Heazel, Chuck Howell, and Thomas Buckman. I would also like to thank Ralph Wachter of ONR for supporting this work, and for providing a number of very helpful suggestions on the content of the document.

Table of Contents

Section **Page**

List of Figures

List of Tables

Section 1
Introduction

1.1 Purpose

An *interoperability technology* is an integrated, automated set of capabilities that makes it easier to share resources. The shared resources are usually data, but interoperability technologies may also promote the sharing of software, physical components, or even people. A *fully deployed interoperability technology* is one whose use has already saturated most of its potential application areas. An example of a fully deployed interoperability technology is ASCII (American Standard Code for Information Interchange), a old standard for exchanging character data that has almost totally replaced alternative character coding technologies such as EBCDIC (Extended Binary Coded Decimal Interchange Code) for the global exchange of character data. Fully deployed interoperability technologies may be extended or replaced by new technologies, but once in place they tend to remain stable due to the high cost and lack of benefits of replacing them with a comparable alternative.

An *emerging interoperability technology* is an interoperability technology that exists and is at least partially in use, but which has not yet – and may never – reach saturation of its potential usage areas. Because they are not yet fully deployed, emerging interoperability technologies are both riskier (they may never be fully deployed) and higher in potential (full deployment may greatly increase benefits to all users) than fully deployed interoperability technologies. As of mid 2000, XML (eXtensible Markup Language) is a good example of an emerging interoperability technology. XML was designed as an easier-to-extend replacement for HTML (Hypertext Markup Language), the language upon which the Internet was originally built. XML is gaining rapidly in popularity, but it is currently far short of its potential saturation levels as a replacement for HTML.

The overall purpose of this report is to provide general guidance on how to select and promote the use of information-related interoperability technologies, and to briefly review a number of promising (mid 2000) emerging interoperability technologies. To accomplish this, the report first defines concepts and terms for understanding the interoperability problem, and also addresses the process by which technologies become standardized to help address such problems. The report then briefly reviews a number of emerging information technologies, focusing on major concepts and comparative merits with references provided for readers interested in technical details. No prior knowledge of the technologies is assumed. The information-related emerging interoperability technologies covered in this report include:

- High-portability programming languages
- Java
- Jini

- JavaBeans and InfoBus
- EJB (Enterprise Java Beans)
- Microsoft COM (Common Object Model)
- DCOM (Distributed Component Object Model) and COM+
- Microsoft SOAP (Simple Object Access Protocol)
- WAP (Wireless Application Protocol)
- CORBA (Common Object Request Broker Architecture)
- Java RMI (Remote Method Invocation)
- XML (eXtensible Markup Language)
- COE (Common Operating Environment)
- Linux

Finally, the report suggests some strategies for applying its information to naval issues.

1.2 Background

Interoperability is a vital issue whenever groups and equipment from diverse origins must work together to meet a shared objective. Modern naval operations must deal with a broad range of resource types, time scales, and scopes of operations. A naval operation may be anywhere from local to global in scale, and may vary greatly in terms of the diversity of the personnel and equipment involved. A simple operation may only require resources from one defense service, but more commonly it will require the interactive use of multi-service, allied, or even *ad hoc* coalition resources and personnel.

Effective selection and deployment of interoperability technologies can directly benefit naval operations by making the composite command structures more cohesive and responsive, and by providing groups at the tactical level with richer and more up-to-date information resources. The improvements in the coordination from using interoperability technologies in a composite force can literally make the difference between the success and failure of that operation, and are one of the main reasons for pursuing greater use of such technologies.

Another major benefit of good interoperability shows up in training and operations. High levels of interoperability across systems can greatly increase the value of trained skills by making them more “portable” across a wide range of equipment. Interoperability can also increase the reliability with which a skill is used, since its user will no longer have to learn a large number of confusing variations of it for different systems. Finally, interoperability makes it easier to move personnel into critical situations by increasing the overall pool of personnel who can operate potentially critical systems.

On the negative side, a poor choice of interoperability technologies can actually increase the isolation of a system by making it incompatible with later systems. Another important risk in using interoperability technologies is that better interoperability also unavoidably means broader, easier system access, which in turn can make security an increased concern. For these reasons the selection of a new interoperability technology should never be taken too casually.

1.3 Interoperability Technology Impacts

Interoperability is not just a technology issue. For example, no technology can directly provide an organization with the ability to ensure uniform use of interoperability standards. On the other hand, the explosive growth of the global Internet in the 1990s aptly demonstrated how a well-designed interoperability technology can encourage the growth of interoperability. In the case of the Internet, a collection of relatively simple network protocols and associated software (HTTP, TCP/IP, and web browsers) played a profound role in making possible the rapid transition to global data and resource sharing over the Internet. A major motive for examining emerging interoperability technologies is that one or more of these technologies could well turn out to have similarly powerful impacts within their own particular ranges of application.

1.4 Interoperability Technologies and Policies

Although the focus of this paper is interoperability technologies, it should be noted that effective use of such technologies cannot be fully separated from policy for using and administering sharable resources. After all, a technology that enables sharing of data, software, parts, or people will be of little benefit if the policies (or lack of policies) that control those resources makes such sharing difficult or impossible. Supporting interoperability technologies at the policy level can be difficult, and even well intentioned policies can have unexpected results. For example, simply issuing a service-wide policy requiring use of a specific interoperability standard or technology can actually end up hurting overall interoperability instead of helping it due of the risk of the technology being "orphaned." That is, the technology may lose its support in industry and other DoD services because of rapid changes in competing technologies or their supporting markets. Orphaned interoperability technologies can be especially troublesome, since they can end up isolating its users and making the overall interoperability problem harder instead of easier.

There is no easy answer to the issue of how to combine interoperability technologies with sound supporting policies, and such an answer would in any case be outside the scope of this paper. On the other hand, the increasing use of information technology in warfare means that there will always be a need for interoperability technologies as more and more of the data exchanges in the battlefield begin to take place on time scales that are too fast for people to follow. This need for real-time interoperability without necessarily involving humans and policy directly will help ensure that interoperability technologies have an important place in future systems, even when the precise policies for supporting and using such systems have not yet been worked out.

1.5 Overview of Report

This report is divided into five sections, each of which addresses a particular aspect of understanding emerging interoperability technologies.

Section 1 is this introduction, which explains the problem and provides an overview.

Section 2 defines fundamental interoperability concepts and develops a terminology for describing interoperability issues and requirements. The purpose of this section is to provide a way to describe the requirements for emerging interoperability technologies more precisely. This section also discusses the vitally important issue of interoperability security.

Section 3 part describes the processes by which interoperability technologies are standardized, and how such standardization processes affect the use and selection of emerging interoperability technologies. This section also compares the interoperability benefits of the various standardization processes that it defines.

Section 4 provides a quick, high-level overview of a number of high-potential emerging interoperability technologies, and provides references for further investigation of these technologies.

Section 5 concludes the report with a summary of potential opportunities for applying emerging interoperability technologies to naval and defense interoperability problems.

Section 2

Fundamental Interoperability Concepts

2.1 A Definition of Interoperability

Interoperability is the ability to use resources from diverse origins as if they had been designed as part of system. Individual interoperability problems may disappear as such resources literally become part of a single system through integration and standardization, but the problem of interoperability itself never disappears – it simply moves up to a new level of complexity that accepts earlier integration as a given.

This same example emphasizes the critical role that security plays in interoperability. Improved interoperability opens new highways that an adversary may be able to use just as effectively as a coalition partner.

Interoperability is a difficult problem for military forces, where international politics can result in abrupt realignments where yesterday's foe is today's coalition partner. The time available to make their systems and resources work together is often very short, and security becomes a major complication as forces attempt to share resources in the short term without necessarily providing any additional mutual access in the long term. Another security issue that must be addressed in military interoperability is a higher risk of major security breaches, since the "highways" of sharing that interoperability builds between forces can often be used as effectively by intruders as they can by the forces for which they were constructed.

Interoperability can also be difficult to design into military systems because the strongly hierarchical operational command structures of military services can inadvertently lead to the development of new systems in isolation from each other, even when the developers are aware of the need for such systems to interoperate. The result can be a "stovepiping" effect that results in systems that can interoperate only with great cost and difficulty.

Section 2.2 discusses the unique interoperability needs of information technologies and why they differ in important ways from physical interoperability requirements. This is followed in Section 2.3 by the development of a set of concepts and terms for describing all types of interoperability requirements for both hardware and software.

2.2 Unique Features of Information Technologies

As will be seen in the definition sections that follow this one, there is relatively little difference in how fundamental interoperability concepts apply to hardware (physical) and software (information) resources. On the other hand, it would be grossly incorrect to leave the impression that there are no significant differences in how interoperability works at the hardware and software levels. This section looks at some of those differences and their implications to emerging information interoperability technologies.

2.2.1 Unique Aspects of Software

Software interoperability differs from hardware interoperability largely because software itself differs from hardware in a number of significant ways. Some of these differences include:

- **Software is more complex than hardware.** While it is true that some large, costly physical machines such as the Space Shuttle can be extremely complex, software is unique in that even low-cost, relatively small software "machines" have complexities comparable to or higher than those of the most expensive physical machines. This "casual complexity" of software can make software especially hard to understand and standardize.
- **Software changes faster than hardware.** Because it is so easy to change binary information, there is a powerful incentive in system design to keep as much design complexity and uncertainty as possible in the form of software. While this is definitely a positive trend in terms of the overall flexibility and reliability of systems, it also means that software is the most likely part of a system to change. Rapid changes can make it extraordinarily difficult simply to obtain interoperability in the first place, and can make it equally difficult to keep it current.
- **Software is easier to replicate than hardware.** Because software and data are both forms of pure information, they are enormously easier to copy and disseminate than hardware. This can be both a good and a bad thing, since on one hand it means that critical information resources (e.g., tactical situation data) can be dispatched to wherever they may be needed. On the other hand, easy dissemination of software and data can also greatly complicate security by making theft of data much easier.
- **Software does not wear out.** Although software can (and does) quickly become obsolete or unusable for any of a number of reasons, "wearing out" is not one of those reasons. As information machines, the patterns that constitute software and data remain the same for as long as the software is recorded on some form of digital media. When software does stop working, it is nearly always due to modifications made to it over time, or because of changes to its environment.
- **Software is extremely compact.** Especially for modern computers, storing software, requires very little physical space, and most of the space that it does occupy is taken up by packaging and documentation. Software that is transmitted over the Internet with its documentation in electronic form is so compact (essentially a few square centimeters of a disk surface) that issues of physical size are essentially meaningless.

Given these differences, the bottom line is that software interoperability is more complex than hardware interoperability. Software interoperability thus requires strategies that are more complex than those required in hardware interoperability. These differences are discussed below.

2.2.2 Types of Hardware Interoperability

Hardware interoperability technologies focus more on the physical aspects of the interoperability problem. Four main categories of hardware interoperability methods and strategies can be defined:

- **Mechanical Interoperability** looks at whether components are physically compatible with each other.
- **Chemical Interoperability** looks at whether systems can make use of the same chemical assets (e.g., low-octane gasoline).
- **Electrical Compatibility** similarly looks at whether diverse systems can make use of (or convert) the same electrical power assets.
- **Data Link Interoperability** addresses use of the same logical interpretation by hardware components to permit sending and receiving of data. Data link interfaces can be electronic (wires), optical (fiber), or radio frequency in nature. Data link interoperability tends to be the most complicated form of physical interoperability, both because of the different media involved and because the complexity of some connectors is driven in part by how they will be used by software.

The use of standards is especially important for hardware interoperability, since adapters for physical interfaces can be costly, clumsy, and at times highly restrictive. The details of the engineering methods used to design and fabricate hardware-level interoperability interfaces are beyond the scope of this document. However, the obvious overlap of the *Electrical* and *Data Link* categories with the basic ability to exchange data shows how important hardware interoperability technologies are to achieving the initial levels of information interoperability upon higher levels are based.

2.2.3 Software Interoperability Strategies

Software interoperability strategies use the power and flexibility of computers and related information technologies to convert information from one form to another. As processors have become cheaper and more powerful, the importance of this approach to interoperability has increased greatly, since processors are now capable of converting many types of information (including audio, visual, and radio frequency) quickly enough to make conversion and adaptation approaches viable. The negative side of software-based interoperability is that it requires enough knowledge about each type of resource to allow "translation" between them. Collecting and integrating this information into a system can be daunting when many types of resources are involved (e.g., in a coalition deployment environment), or when the details of some resources are not well known (e.g., when older legacy systems are involved). To help control the complexity of the problem, software-based interoperability strategies rely on a combination of standards (e.g., of the protocols used to build the Internet) to permit basic communications, and more complex conversion and bridging strategies that are built on top of those standards.

2.2.4 Software Levels of Interoperability

Another notable feature of software interoperability is that it is usually accomplished by creating layered systems, in which each new layer makes possible the exchange of increasingly complex forms of data or software. From an engineering perspective, layered approaches reduce design complexity and make it easier to add new capabilities. From a field operations perspective, layered approaches increase flexibility and adaptability by providing "fall back" levels of interoperability that can be used when higher levels are unavailable. For example, *ad hoc* coalition ships would not typically be able to exchange entire applications and their associated databases, but they can usually fall back to some lower level of interoperability. This lower level might consist of an exchange of structured data such as selected database records, or at a still lower level, an exchange of unstructured data such as freeform text.

It should be noted that using layers to build interoperability is a design decision, and not a fundamental characteristic of interoperable systems. Layering is most commonly used in digital information systems in which it is easy to define more than one level of complexity in the way a string of bits is interpreted. In contrast, traditional analog voice radios usually lack layering, since the analog voice data is very difficult to break down into simpler, more fundamental units.

One example of a layering approach for software interoperability is shown below, but many other variations are possible.

2.2.4.1 Physical Link Layer

The *physical link layer* is the physical infrastructure that enables basic exchange of raw binary or analog data between systems. It includes components such as cable, radios, antennas, CPUs, signal encoding, bit error recovery logic, NICs, hubs and repeaters. For allied and coalition environments, reaching even the physical link level of interoperability can be difficult due to incompatibilities of equipment and standards.

2.2.4.2 Networked Data Layer

The *networked data layer* provides a layer of abstraction between the hardware oriented physical layer and the software-oriented layers. It includes the operating system, protocol stack, network management, router control logic, address assignment and naming (location) services. This second level of interoperability enables directed, point-to-point exchange of raw binary or analog data (e.g., via packet or switched circuits).

2.2.4.3 Basic Syntax Layer

The *basic syntax layer* enables exchange of syntactically structured data between components, and requires the use of standardized conventions and software between recipients. IEEE standard formats for text (ASCII) and numeric data are examples of standards that operate at this level. This level of interoperability enables exchange of meaningful data and values, but does not ensure consistent use or interpretation of the received values.

2.2.4.4 Complex Syntax Layer

The *complex syntax layer* includes any kind of software that helps ensure the consistent interpretation of data when it is ported between different types of computer systems and platforms, and can be viewed as a more advanced extension of the data consistency standards of the Basic Syntax level. By providing more structure, this level also begins to support the active exchange of control data to direct remote processing operations. Examples of generic support level technologies include CORBA (Common Object Request Broker Architecture) standards and software, and XML (eXtensible Markup Language) standards and tools.

2.2.4.5 Domain Semantics Layer

The *domain semantics layer* of interoperability provides a sufficiently high level of automation between recipients to allow common interpretation of complex, applications-specific data. Examples include the ability to consistently and correctly interpret visual battlefield data at multiple sites and across multiple groups and cultures. Ideally, this level would include generic support for each group and culture represented, so that operations such as metrics conversion can always be done using the same infrastructure tools. In practice, however, domain semantics support is more often accomplished at current technology levels by incorporating the conversion capabilities in each of the tools that require it.

2.2.4.6 Full Semantics Layer

The *full semantics layer* provides sufficient machine-based "understanding" of data and requests to allow transfers of information even between different domain-specific areas of expertise. Thus, for example, requests for the status of a tactical situation could result in collection of meaningful data from all coalition partners and systems, and for all aspects of the battle situation. This level of interoperability does not exist in fully automated form in any current systems, although can be said to exist in limited form when networks incorporate people to interpret and translate semantics for the rest of the network. Although it may never be fully feasible, complete automation of the full semantics level represents an important overall automation goal towards which interoperability technologies can be developed and applied incrementally.

2.3 Concepts for Describing Interoperability Requirements

The remainder of Section 2 presents a set of concepts and terms that were defined as part of the development of this report. They apply to all forms of interoperability, both information-based and hardware-based. Examples are given throughout this section to show how the concepts apply to both classes of interoperability

2.3.1 Missions, Assets, and Interoperability Domains

The most fundamental concept of this section is that interoperability exists as a problem only relative to some well-defined type of *resource* or *asset* that is being applied to a common purpose. A simple example of this concept, based on the idea of a bullet being such a shared asset, is shown in Figure 2-1.

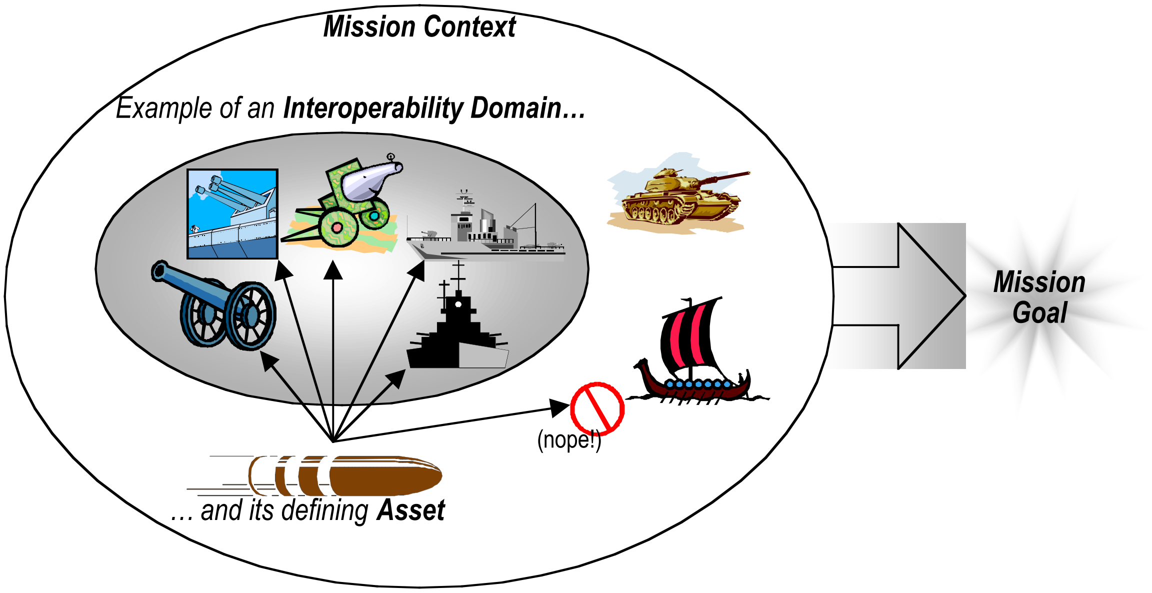

Mission Context
Example of an Interoperability Domain...
(nope!)
... and its defining Asset
Mission Goal

Figure 2-1. Missions, Assets, and Interoperability Domains

A *mission* (Table 2-1) is an undertaking that defines such a problem context. Missions may range in size from very small in size and short through international efforts that last for decades, but they all require contribution of resources or assets from the participants.

An asset can be anything that helps contribute towards meeting the objective of the mission. There are three broad categories of interoperability assets:

- *Physical assets* are material objects such as vehicles, fuel, and personnel.
- *Information assets* are items such as software and data are composed entirely of digital information.
- *Skill asset*s human abilities to accomplish tasks, such as interpreting acoustic signals or operating computers. Skills are similar to software in terms of what they can accomplish, and they can in some cases be replaced by software. However, a skill is not digital information and can only exist in a person.

For interoperability purposes, a *system* is a constructed entity that uses assets to accomplish a goal. Systems are also assets, but are distinguished by their ability to use or process simpler assets. Systems can be physical (e.g., an engine), information based (e.g., a software application), or skill based (e.g., a standardized procedure for interpreting reconnaissance data).

Table 2-1. Missions, Assets, and Interoperability Domains

Mission	An undertaking that is defined by a single overall goal to which all the participants in the mission have agreed. The **mission context** is the region of space and time for which the mission exists. The **mission assets** are the assets available within the mission context. The **mission systems** are the entities that use or process assets within the mission context.
Asset	Anything that can be used to help accomplish a well-defined mission or goal. There are three broad categories of assets: physical, information, and skill. A **physical asset** such as a vehicle, fuel, or a person is a material object. An **information asset** such as software or data is composed entirely of digital information. A **skill asset** is a non-digital human ability to accomplish tasks such as interpreting acoustic signals or operating computers.
System	A constructed entity that uses assets to accomplish a goal. A system is also an asset, but is distinguished by its ability to process other simpler assets. A system may be physical (e.g., an engine), information based (e.g., a software application), or skill based (e.g., a standardized procedure for interpreting reconnaissance data).
Interoperability Domain	The full set of mission systems in which a specific type of asset can be used. A **simple interoperability domain** supports the sharing of a single type of asset. The collection of all systems capable of receiving and understanding a standardized radio broadcast of a time synchronization signal is an example of a simple interoperability domain. A **compound interoperability domain** supports sharing of more than one type of asset. The set of all coalition locations that can use U.S. equipment is a compound interoperability domain.
Target Interoperability Domain	An interoperability domain that has been defined as a desired objective, but which does not yet exist within the mission context.
Interoperability Goal	The full set of target interoperability domains defined for a mission.

With the terminology given above, it is now possible to define the concept of interoperability with greater precision. An *interoperability domain* is the full set of mission systems in which a specific type of asset can be used. A *simple interoperability domain* supports the sharing of only one type of asset. The set of all vehicles that use diesel fuel is an example of a simple interoperability domain. A *compound interoperability domain* supports sharing of more than one type of asset. The set of all coalition locations that can use U.S. equipment is a compound interoperability domain.

It should be noted that this approach defines interoperability in terms of assets, as opposed to "compatibility" of systems. The concept of asset-based interoperability domains provides greater precision in describing interoperability status and goals for a mission. A *target interoperability domain* is just such a goal, stated in the form of an interoperability domain that has been defined as a desired objective, but which does not yet exist within the mission context.

Similarly, the *interoperability goal* of a mission is simply the set of all target interoperability domains for that mission. The interoperability goal can be thought of as a statement of the interoperability requirements of a particular mission, stated in terms of specific assets and the systems on which they need to be usable.

2.3.2 Asset Characteristics

Different types of assets require different strategies for making them interoperable. Figure 2-2 shows three resource examples, each of which demonstrates at least two of these characteristics. The full set of characteristics is described in Table 2-2.

Simple / Single-Use ***Compound / Reusable*** ***Replicable / Perishable***

Figure 2-2. Asset Characteristics

Table 2-2. Asset Characteristics

Simple Asset	An asset that cannot be divided up into smaller parts without losing all significant value within the context of a mission. An intact rifle barrel is a simple asset in a tactical mission, but a broken rifle (which cannot normally be repaired within a tactical context) is not.
Compound Asset	An asset that is comprised of two or more simple assets. A rifle is compound asset, since even in a tactical environment it can be taken apart into simpler parts and reassembled later.
Single-Use Asset	An asset such as fuel or artillery shells that is either destroyed or irreversibly transformed by the act of using it. Most supplies (e.g., ammunition and food) are single-use assets. A single-use asset may also be called a **consumable asset**.
Reusable Asset	An asset such as a ship, software, or a human skill that (barring its unplanned damage or destruction) can be used and reused many times over the duration of a mission. A reusable physical asset is also called a **durable asset**.
Replicable Asset	An asset that can be quickly and cheaply replicated or reproduced many times within the timeframe of a mission. Many information assets are replicable, whereas most physical and skill assets are not.
Perishable Asset	An asset whose value declines rapidly in comparison to the overall duration of a mission. The decline may be preventable through the application of other assets (e.g., refrigeration of perishable food), or it may be unavoidable due to a direct dependency on time or other external factors (e.g., reconnaissance data in a tactical environment). Tactical data is an example of an information asset that is both replicable (it can be replicated quickly and easily) and perishable (it becomes useless after a certain length of time no matter how many times it has been replicated).

Simple assets cannot be divided up into smaller parts without losing all significant value within the context of a mission, such as bullet or (for information technology) a data packet. A rifle is compound asset, since even in a tactical environment it can be taken apart into simpler parts and reassembled later. A modular software application with options for adding and removing features as needed is an example of a composite information technology asset.

Compound assets often imply significant interoperability issues. In the cases of both rifle and modular software applications, it will not in general be possible to share or interchange the simpler components from which they are constructed unless those components have previously been designed to meet a common standard. This can in turn "fragment" the associated interoperability domains, resulting in complex support logistics and higher risks of unexpected incompatible combinations.

Single-use assets such as fuel, bullets, or data packets are either destroyed or irreversibly transformed by the act of using them, and for that reason may also be referred to as *consumable assets*. Most supplies (e.g., ammunition and food) are single-use assets, and so are most of the information constructions such as packets that are used to transfer data around in a network. *Reusable assets* are items such as ships, software, or human skills that normally can be applied many times over the duration of a mission. When a reusable asset is also a physical asset, it may be referred to as a *durable asset*.

Replicable assets are ones that can be quickly and cheaply created or reproduced within the timeframe of a mission. Most information assets, such as data packets, files, and software applications, are replicable, whereas most physical and skill assets are not. Replicability thus is one of the more significant differences between information and physical assets. Because there is so little cost in duplicating the assets themselves, it is often worth the trouble of building a complex transport system (a network) to carry copies of them quickly throughout the entire mission space. Because they can be broadcast so easily, replicable assets provide a major opportunity for sharing useful resources if they can be made interoperable across all of the systems in a mission.

Perishable assets are ones whose value declines quickly in comparison to the overall duration of a mission. The decline may be preventable through the application of other assets (e.g., refrigeration of perishable food), or it may be unavoidable due to a direct dependency on time or other external factors (e.g., reconnaissance data in a tactical environment). Tactical data is an example of an information asset that is both replicable (it can be replicated quickly and easily) and perishable (it becomes useless after a certain length of time no matter how many times it has been replicated). For perishable information assets, speed of transport and interpretation (that is, of interoperation on mission systems) can be especially critical.

2.3.3 System and Asset Adaptability

In general, there are two strategies for making an asset usable within an interoperability domain. The first and easily most preferable method is to use agreed-to standards during the original design of the systems that use an asset. However, standards-based interoperability domains are not always feasible for missions with diverse participants and systems. The development and use of standards is discussed in another section of this document.

The second strategy is to use *adapters* (Figure 2-3 and Table 2-3) that that convert one type of asset into another (usually similar) form that works on a different system. Compared to using standards to create interoperability domains, adapters require more design, are more expensive, and can lead to slower systems. On the other hand, adapter-based systems are often more flexible and robust than purely standards-based systems, since the ability to handle a wide range of unexpected cases is an inherent part of adapter design.

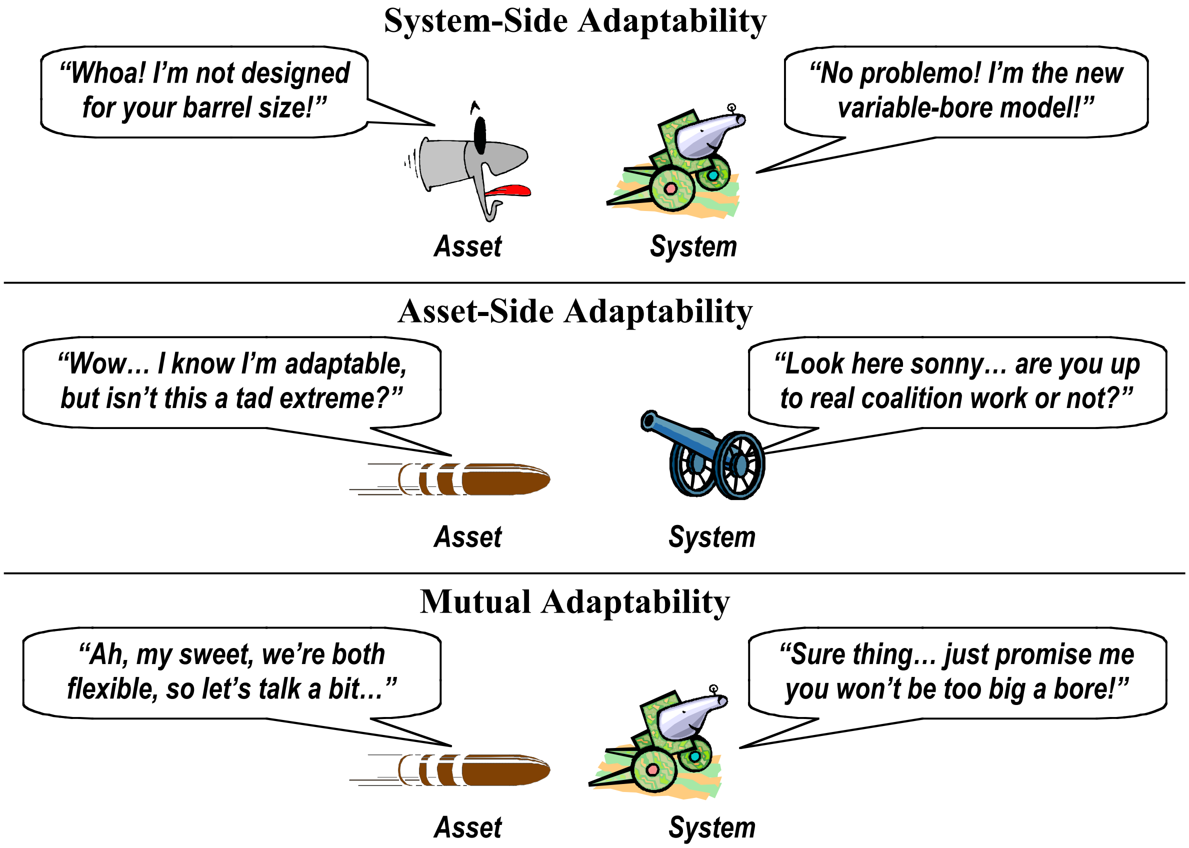

System-Side Adaptability
"Whoa! I'm not designed for your barrel size!"
"No problemo! I'm the new variable-bore model!"
Asset
System
Asset-Side Adaptability
"Wow... I know I'm adaptable, but isn't this a tad extreme?"
"Look here sonny... are you up to real coalition work or not?"
Asset
System
Mutual Adaptability
"Ah, my sweet, we're both flexible, so let's talk a bit..."
"Sure thing... just promise me you won't be too big a bore!"
Asset
System

Figure 2-3. System and Asset Adaptability

Adaptability can be designed into any kind of a system, but it is an especially important strategy for information technology systems because of their ability to transform data rapidly and in complex ways. For physical systems, simple examples of adaptability include adjustable wrenches (mechanical adaptability), engines designed to accept a wide range of fuels (chemical adaptability), portable computers that can accept a wide range of input power voltages and frequencies (electrical adaptability), and radio receivers that accept multiple frequencies and formats. For information systems, examples of adaptability include compilers that translate source code for use on more than one type of computer, applications that can display more than one type of graphical file, software for translating email messages, and network portals that transform data packets for transmission across incompatible networks. Many of the technologies described in Section 4 deal with adaptability issues.

Table 2-3. Asset Adaptability

Adapter	An mechanism that allows an asset to be used in a system for which it was not specifically designed. Adapters may reside in the asset (**asset-side adaptability**), in the system that uses the asset (**system-side adaptability**), or on both sides (**mutual adaptability**).
System-Side Adaptability	The ability of a system to adapt itself to use a variety of assets. An example would be an internet browser that supports display of many different types of graphics files.
Asset-Side Adaptability	The ability of an asset to adapt itself for use in a variety of target systems. An simple example is a screw that accommodates both regular and Phillips head screwdrivers.
Mutual Adaptability	When both an asset and the system that uses it have the ability to adapt to each other. Systems with mutual adaptability may require a brief period of negotiation to determine the most effective combination of asset and system adapters to use. Modem systems that operate at more than one data rate on both sides are examples of mutual adaptability.

Figure 2-3 and Table 2-3 define three major strategies for building adaptability into a mission. Adapters may reside in the system that uses the asset (*system-side adaptability*), in the asset itself (*asset-side adaptability*), or on both sides (*mutual adaptability*). Examples of system-side adaptability include Internet browsers that can display many different types of graphics files, and software-based radios that can adapt themselves to many different radio formats. A simple example of asset-side adaptability is a screw designed to accommodate both regular and Phillips screwdrivers. The combination of a multi-bit screwdriver with a multi-head screw provides an example of mutual adaptability, since more than one choice is possible on both sides. Systems with mutual adaptability may require a short period of negotiation to determine which combination of asset and system adapters to use. A simple example of mutual adaptability is the way home computers connect to Internet service providers. To establish such connections, a home computer modem and a service provider modem must both look for and lock onto the highest possible mutually compatible data rate.

2.3.4 Asset Logistics

An important dynamic aspect of an interoperability domain is how it transports assets among the systems in the domain. This concept of *asset logistics*, or the timely distribution of assets within an interoperability domain, is described in Figure 2-4 and Table 2-4.

The *initial asset distribution* of an interoperability domain describes where assets of a particular type are located immediately after they are first created or as they are added into a mission. In contrast, the *optimum asset distribution* describes where the assets need to be to provide the maximum benefit for accomplishing the mission objectives. These two endpoints are often vastly different, and that difference describes the overall requirements for dynamic transport of the asset within its domain. The *asset transport system* is simply the set of mechanisms that make such dynamic transport of the assets possible. A fleet of aircraft carriers is an example of an asset transport system for aircraft, and a data network is an asset transport system for command, control, and operational data. As shown by these examples, assets usually share such transport systems, although dedicated transport systems may be constructed for especially critical types of assets (e.g., for command and control data).

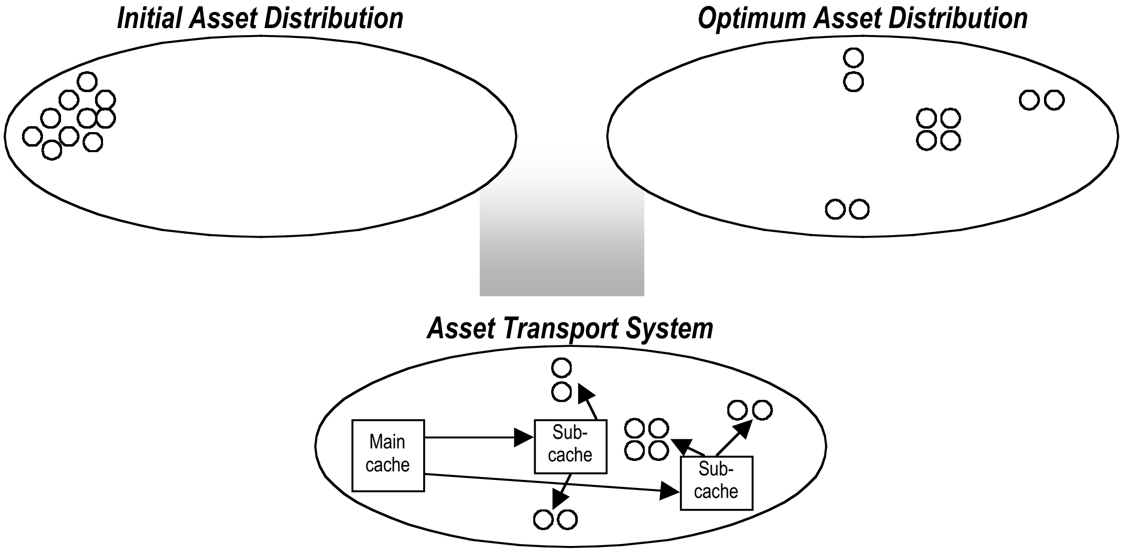


Figure 2-4. Asset Logistics

Table 2-4. Asset Logistics

Initial Asset Distribution	The set of locations where assets of a particular type are located as they are first created or as they are added into a mission.
Optimum Asset Distribution	The set of location where assets of a particular type provide the maximum benefit for accomplishing mission objectives.
Asset Transport	The act of relocating an asset.
Asset Transport System	Within an interoperability domain, the set of mechanisms that make transport an asset possible. A fleet of aircraft carriers is a transport system for aircraft assets, and a data network is a transport system for command, control, and operational data.
Asset Cache	A location at which assets are stored until needed. For a particular asset a mission may use more than one size and location of caches, depending on levels of need and how perishable the asset is. This may also be called an **asset depot.**

Finally, an *asset cache* is simply a location at which asset can be stored until needed. Like transport systems, different types of assets usually share these caches. For a particular type of asset the optimum storage arrangement will generally include more than one cache size and location, and determining the best arrangement of caches for can be a complex process.

2.3.5 Asset Security

One of the more important (and possibly more overlooked) aspects of building interoperability into mission systems is the following relationship between security and interoperability. We suggest the following rule-of-thumb as a way of expressing this issue:

> *The higher the level of interoperability, the greater the risk of system-wide failure due to malicious intrusion*

This unsettling rule is simply a consequence of the observation that you cannot commit a robbery until you have a way to get to the bank. As the idea of interoperability domains helps point out, the real goal in interoperability is to create "highways" that lead directly into all the systems involved in a mission. With full interoperability, malicious intrusions that previously might have affected only one system can travel more easily to other systems by way of the interoperability domains that have been defined for them. This problem is especially true when the defined interoperability domains include assets that control system resources and security.

The pervasiveness of this rule can be seen by its existence in natural biology. When populations of animals or plants become too "standardized" genetically (e.g., due to inbreeding), the chances of a disastrous plague increase dramatically as the pathogens "discover" exactly the same genetic breeding ground exists in all members of the species. This is one reason why cloning is relatively rare in nature – it creates populations that are too easily destroyed by a single new pathogen. (It is also an interesting argument against achieving interoperability *only* by using standards, which could create extremely homogenous systems that could all fail at the same time.)

Biology also provides evidence that high levels of interoperability are nonetheless possible, but only if adequate security mechanisms are in place. Complex organisms such as humans have cells that freely share a very broad range of resources, yet the system as a whole remains capable of resisting most pathogen attacks. The difference is that these cellular sharing mechanisms include the use of complex security techniques such as self-identification, self-evaluation, and constant policing of the "interoperability medium" for suspected pathogens.

Figure 2-5 and Table 2-5 introduces a few basic concepts for understanding the security issues of interoperability domains. An *asset container* is that portion of an asset that does not directly contribute to its use, but which is necessary either to prevent damage or loss of integrity of an asset while it is being transported. An asset container may also be responsible for providing an environment that is required for correct operation of the asset. Fuel tanks and shipping containers are physical asset containers, while network data packets are information containers. Humans are containers for skills, since they provide the "environments" for exercising them. Containers are themselves a type of asset and may be single-use (food wrappers) or durable (shipping containers). Containers are important from a security viewpoint because they can include security features.

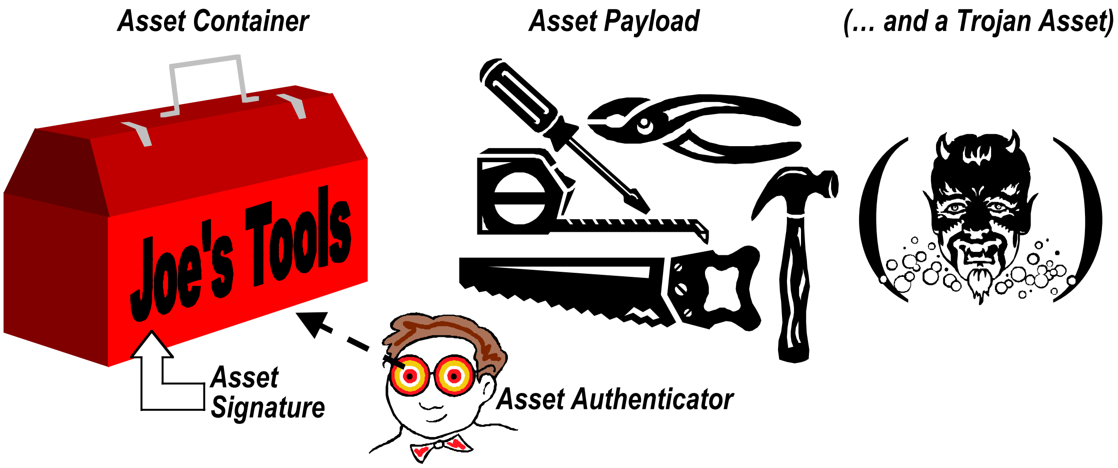


Figure 2-5. Asset Security

Table 2-5. Asset Security

Asset Container	The portion of an asset that does not directly contribute to its use, but which is necessary either to prevent damage or loss of integrity of an asset while it is being transported, or to provide an environment that is required for the correct operation of the asset. Fuel tanks and shipping containers are physical asset containers, while network data packets are information containers. Humans are containers for skills, since they provide the "environments" needed to use them. Containers are themselves a type of asset and may be single-use (food wrappers) or durable (shipping containers).
Asset Payload	The usable content of an asset, as distinguished from the asset container. The asset payload of a fuel tanker is the fuel, and the asset payload of a data packet is its usable data content.
Asset Signature	The external characteristics of an asset that make it of recognizable and usable within a mission. For a physical asset such as an artillery shell, the signature may include issues such as size, shape, weight, chemical composition, and engineered performance characteristics. For an information asset such as a software application, the asset signature consists of the features of the application that allow it to operate correctly and safely on a specific computer system. For a skill asset, the signature consists of a human-understandable description of the skill and how it applies to mission objectives.
Asset Authenticator	A mission component that is responsible for analyzing the signature of an asset and deciding whether the asset is what it claims to be, and that it is safe to use in a particular context.
Trojan Asset	An entity that presents the signature of a valid asset, but whose payload is malicious – that is, in opposition to the overall objectives of the mission. Nonfunctioning ordinance, software viruses, and human agents are all examples of possible Trojan assets.

An *asset payload* is the usable content of an asset, as distinguished from its container. The asset payload of a fuel tanker is the fuel, and the asset payload of a data packet is its usable data content. An *asset signature* is the collection of external characteristics of an asset that make it recognizable (and usable) within a mission. For a physical asset such as an artillery shell, the signature may include issues such as size, shape, weight, chemical composition, and engineered performance characteristics. For an information asset such as a software application, the asset signature consists of the features of the application that allow it to operate correctly and safely on a specific computer system. For a skill asset, the signature consists of a human-understandable description of the skill and how it applies to mission objectives.

An *asset authenticator* is a system mechanism that is responsible for analyzing the signature of an asset and deciding whether it is what it claims to be. The authentication process may be either continuous or periodic, depending on the design of the system, but it should always ensure that authentication will be completed before the asset is opened or used in any way.

Finally, a *Trojan asset* is an entity that presents the signature of a valid asset, but whose payload is malicious – that is, in opposition to the overall objectives of the mission. Nonfunctioning ordinance, software viruses, and human agents are all examples of possible Trojan assets.

2.4 Interoperability Domains and Standardization

In summary, interoperability issues can be defined and quantified more specifically by defining *interoperability domains* over which the sharing of a specific type of resource can take place easily and safely. Identifying where and how these domains should overlap to make logistical support easier helps guide overall design, and defining sharp boundaries for the domains helps define and quantify the security problem. There are two main approaches to making such sharing of assets possible: standardization, and adaptability. By far the most important of these two approaches is standardization, since even adaptable parts and systems must usually share some set of underlying standards to work effectively.

Once a system has been understood in terms of interoperability domains and sharing of assets, the next problem is often the creation of standards to support the required domains. Because of the importance of standardization in the creation of effective interoperability domains, the next section (Section 3) is devoted entirely to developing and understanding how standardization works, how it can be promoted, and how it can fail. Unlike locally controlled design, standardization is a complex process that is as much economic and political as technical, and understanding how it works is vital to applying new technologies.

Section 3

Standardization of Interoperability Technologies

3.1 Standardization as a Competitive Process

Standardization of interoperability technologies is a far more Darwinian process than the placid pace of many standards committees might lead one to think. Firstly, the promise of a new technology is rarely obvious enough to persuade conservative standards groups to invest the substantial time and personnel resources needed for a full committee-style standardization effort. Secondly, the actual success of a standard often depends upon competitive issues that have little to do with the standards process itself. Standards that fail to gain sufficient recognition or approval within their target communities may fail despite great technical promise. The market success of the VHS videotape standard over the technologically more impressive BetaMax standard is one example of how such market forces affect the viability of standards.

This section of the reports looks at the standardization process in depth in order to help understand the factors that help lead to the success or failure of emerging interoperability technologies, and to help plan standardization strategies when specific technologies are selected for interoperability applications. Section 3.2 discusses the phases in the full lifespan of a standard, including technology inception, technology exploration, recognition of technology potential, architecting the standard, building a consensus, mainstream use of the standard, and phase-out of the standard. Section 3.3 defines and explains basic standardization processes based on how they deal with issues of owenship and distribution of information. Section 3.4 introduces composite standardization processes, in which the basic standardization processes are combined to create more complex (and often more effective) standardization processes. Section 3.5 addresses technology impacts of standardization processes, including technology exploration and standards prototyping, risks of using closed de facto standards, standardization time scales, and self-destruction of promising technology markets. Section 3.6 looks at effects of transitions in composite processes, including closed-to-open transitions (good), de facto-to-explicit transitions (good), open-to-closed transitions (bad), and explicit-to-de facto transitions (bad). Section 3.7 describes three recommended standardization processes: an ideal open standardization process, an ideal closed-to-open standardization process, and an ideal closed standardization process. Finally, Section 3.8 discusses evaluating technologies in terms of standardization processes.

3.2 Phases in the Lifespan of a Standard

Committee-style standardization thus is better understood as simply the most conspicuous phase of a much more complex overall process of technology acceptance and standardization. The phases of this process are described below.

3.2.1 Technology Inception

Contrary to what one might expect, successful standards usually begin their careers as highly innovative ideas that push or even break the boundaries of acceptable behavior (often as defined by earlier standards). New technologies “sneak in” through niche application areas and tools in which their benefits to a small community of users outweighs the disadvantages of ignoring the standards of some larger community. The premiere example of this effect is the Internet, whose TCP/IP protocols were first developed and proven out by a small and (to most people) little-known community of DoD researchers who used the early Arpanet as a mechanism for sharing research results. In sharp contrast, the laboriously constructed International Standards Organization (ISO) seven-layer protocol stack that was intended to address many of the same issues as TCP/IP is effectively dead in the current market. Another interesting example is the MPEG-4 data compression standard, which creates an innovative framework for diverse types of hardware and software technologies that is very different from its much more rigidly defined, hardware-oriented MPEG-2 predecessor.

One important implication of this is that the best place to look for long-term interoperability standards is probably not in existing standards, but rather in new and innovative “niche market” technologies that may not follow current standards all that closely.

3.2.2 Technology Exploration

For information technology, an innovative new technology generally enters its niche area not as an abstract concept, but rather as part of a specific tool or product that directly demonstrates its value to potential users. The Mosaic browser is an example, since its success in presenting the Internet to users as a set of hyperlinks (the Web) greatly helped promote and solidify the HTTP (HyperText Transfer Protocol) standard that underlies all current Web browsers.

The core set of users of such a new technology tool are typically willing to take more risks with an unproven new technology, and are often found in research or applied research environments. Their adoption of the tool provides an important phase of hands-on use and maturing of the technology, which stabilizes it and helps prove (or disprove) its overall viability.

3.2.3 Recognition of Technology Potential

Next, a group within or at the periphery of the core group recognizes the broader potential of the technology, and decides to push for formal standardization as a way of increasing its value. The transition from niche technology into formal standardization is an especially critical one in terms of its impact on the long-term success of the technology, since it is at this point that major decisions must be made on how to create a generalized architecture for the technology.

3.2.4 Architecting the Standard

Based on the successes of the original core community in using the technology, a small group of architects or designers must propose a new framework for applying the technology to a broader community. Poor design decisions at this point can lead to a framework that is poorly integrated, inconsistent, inflexible, or of little interest to the broader community. The architectural generalization effort may be done by a small architecting group, or as part of the early work of a full-fledged standardization committee.

The initial extraction of the XML (eXtensible Markup Language) standard from the earlier and much larger SGML standard provides a good example architectural generalization, and it also shows the critical role that design decisions play in this phase. In the case of XML, the architecting group chose to make XML a strict subset of SGML. This reduced risk by avoiding inherently risky technology extrapolations, and speeded the dissemination of the standard by making it possible for XML users to take advantage of a mature set of tools developed earlier by the SGML community.

At the other end of the spectrum of architectural generalizations is Ada 83, a programming language that was inspired by the earlier languages Pascal, Algol, and PL/I. While many of the simpler features of Ada 83 were based on well-proven programming principles, the standard also included a number of innovative features (e.g., the *rendezvous* mechanism for inter-process communications) whose implications were not nearly as well understood. This lack of understanding of the implications of the standard significantly slowed the creation of Ada compilers and support tools, and made it harder for programmers to apply Ada effectively. Compared to XML, the adoption of Ada by its target community consequently was much slower and far less pervasive.

3.2.5 Building a Consensus

After generalization of its architecture, a technology is then ready for the familiar and often far more conspicuous committee-style standardization process. By this time the outline of how the technology will be used is usually well understood, so the committee standardization process focuses more on building a community consensus on the details of how to use the technology. Major debates on the technology may still erupt during this phase, but if they do it is often an indication that the earlier architectural generalization phase was either incomplete or poorly done. The goal in this stage is to get a strong consensus from as diverse a range of competing parties as possible, to increase both the completeness and the stability of the draft standard. (Narrowly focused and non-competitive committees can produce unstable standards because they may not adequately represent the full range of issues and priorities needed.)

3.2.6 Mainstream Use of the Standard

The mainstream life of a standard usually begins with the release its first complete draft. The first draft is usually updated soon after its release, in response to the first reports of actual field use of the standard. Standards in mainstream usage are mostly likely to succeed if they are open enough to allow incorporation of new, often unforeseeable innovations and technologies. Rigid standards that cannot handle unanticipated change are more likely to fall by the wayside and be replaced by standards better able to handle technological innovation.

3.2.7 Phase-Out of the Standard

The final phase of a standard is obsolescence and phase-out. This can happen rapidly for poorly designed or out-of-favor standards, since market reluctance to use standards that are perceived as risky can make them irrelevant with surprising speed. Open, adaptable standards may not really become obsolescent at all. Instead, they may be come deeply incorporated into the lower levels of other technologies and standards and survive indefinitely as components of new standards.

3.3 Basic Standardization Processes

Not all technologies follow the above sequence. In particular, standardization processes vary in important ways based on how they deal with issues of technology ownership and distribution. Based on these issues, four basic variations of the standardization process are listed in Table 3-1.

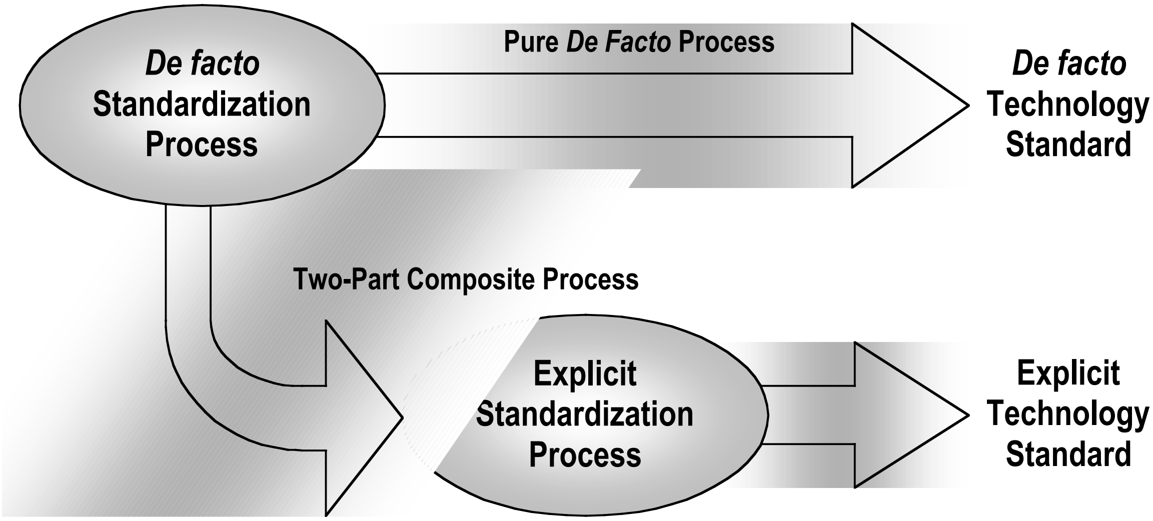
Table 3-1. The Four Basic Technology Standardization Processes

	De Facto	**Explicit**
Closed	Closed De Facto (Example: Windows NT)	Closed Explicit (Example: I2O - Intelligent I/O)
Open	Open De Facto (Example: Linux)	Open Explicit (Example: XML)

Windows NT is a closed de facto standard, since it is proprietary and has never undergone explicit standardization. The competing Unix-like operating system Linux is an example of an open de facto standard, since it has never undergone explicit standardization, but was developed in a very open fashion. The Intelligent I/O (I2O) standards process is explicit but closed, since its results are available only to members of the I2O organization. Finally, the eXtensible Markup Language (XML) is open and explicit, with a committee-based standardization process and results that are readily accessible to any interested group.

3.4 Composite Standardization Processes

For a given technology, more than one of the processes in Table 3-1 may apply over time. Figure 3-1 shows two particularly important applications of the fundamental standardization processes: A pure *de* facto process, and a composite process that begins with *de* facto standardization and subsequently moves into an explicit process. The *de facto* process is important because it produces many of the informal standards that go into commercial information technology products such as Microsoft Office, while the composite process in Figure 3-1 is important because it defines the most common sequence for creating a formal standard. In fact, the typical overall process described earlier is essentially this two-step process, with technology inception and exploration providing the *de facto* phase of the composite process, and the committee-based process that grows out of it providing the explicit phase.



De facto Standardization Process
Pure De Facto Process
De facto Technology Standard
Two-Part Composite Process
Explicit Standardization Process
Explicit Technology Standard

Figure 3-1. Pure and Composite Standardization Processes

3.5 Technology Impacts of Standardization Processes

This section discusses some of the important impacts of standardization processes on the long-term growth and use of an emerging technology.

3.5.1 Technology Exploration and Standards Prototyping

To understand the interplay of standardization processes, the composite process can be viewed in terms of prototyping. The first de facto phase is equivalent to developing and thoroughly testing a prototype of the final standard, while the subsequent explicit phase corresponds to rigorous development of a final standard. The prototyping phase is needed because it is perhaps the most effective way to identify the potential risks and inconsistencies hidden in any type of complex technology specification, and because it allows core developers to be creative in exploring alternatives without undue consequences to a larger community of users. Conversely, being overly creative during the second (explicit) standardization phase can be quite risky. If major technology extrapolations are made at that point, they will lack the proof-of-concept validation that had been provided by earlier prototyping work. Consequently, these late-addition features will tend to be much riskier when played against other more proven components of the standard.

The standardization effort for the Ada computer language is a good example of the risks of attempting creative technology exploration in the explicit phase of standardization. While much of the Ada standard relied on well-proven programming language concepts, it also specified a number of new concepts (the "rendezvous" is a particularly notable example) whose implications to both tool builders and users were largely unknown at that time. The result was lengthy delays and significant confusion in the release and use of efficient Ada compilers. In contrast, the standard for the C programming language was based very heavily on earlier C compilers that had become de facto standards. As a direct consequence, the C standard was much more quickly accepted and adopted by its target community than had been the case with Ada.

3.5.2 Risks of Using Closed De Facto Standards

Closed (that is, single-vendor) de facto standards are a normal part of technology innovation, and the strong profits associated with them provide an important incentive for encouraging vendors to explore innovative approaches. From a consumer viewpoint, however, long-term reliance on closed de facto standards entails significant risks. There are three main problems: closed de facto standards tend to be cryptic, unstable, and costly.

Closed de facto standards are *cryptic* because it may be difficult or impossible to obtain detailed information about them from their owners. They are *unstable* because they are not based on any shared public standard to which a large number of competing parties have agreed. Without such an agreement, there is nothing to keep technology owners from changing a technology standard independently of the broader needs of a user community. Finally, they are *costly* because of their monopolistic nature. With only one owner, users have few ways to influence the price or release of new versions of the technology.

The problem of instability in de facto standards is exacerbated by financial factors that encourage the companies that own them to alter them. A company that owns a popular de facto standard can make large profits at low risk simply by periodically altering the standard, which in turn forces their user community to buy upgrades for products that either use the technology or interface with it. The impending (year 2000) move of much of the PC server industry from Microsoft Windows NT 4.0 to Microsoft Windows 2000 is an example of how changes to a closed de facto standards can destabilize an entire industry. Windows 2000 incorporates a significantly different network infrastructure. These changes effectively obsolete earlier NT 4.0 systems, and thus may require massive upgrades to entire networks.

3.5.3 Standardization Time Scales

Historically, standardization of interoperability technologies often occurs a long time (months or more years) after the introduction of the first tools and products that demonstrate them. An example is the POSIX standard, which helped standardize Unix-like operating system functions that in some cases had been first introduced years or even decades earlier. This delay between introduction and standardization of a technology is not overly surprising, since the promise of a new technology is rarely sufficiently convincing to lead to an immediate effort to standardize it.

3.5.4 Self-Destruction of Promising Technology Markets

Another factor that can delay or even destroy standardization is competition between companies that are attempting to create their own closed de facto standards. Many companies prize closed de facto standards because they can provide monopolies on promising new technologies, and because they give the company a huge advantage in the subsequent support and upgrades market. (It should be noted that open de facto standards usually do not suffer from these problems, since no one company can fully control an open standard.) The positive side of competition among companies for creating closed de facto standards is that it encourages thorough exploration of the technology and its potential uses. The negative side is that the resulting standard is usually closed, and thus subject to potentially arbitrary marketing decisions by the company that owns it.

More importantly in many cases, severe competition to create closed de facto standards can literally destroy the market for a promising new technology, especially if other less promising but serviceable technologies exist alongside the new one. Technology self-immolation is most likely to happen when the following three conditions exist:

- **The technology promises substantial financial rewards.** The first condition is that the technology has high to very high financial promise. Curiously, this has a "gold rush" effect that makes it much more difficult to elicit cooperative behavior from competing groups, and so diminishes the chances for the type of cooperation needed to create a successful standard.
- **The manufacturing process for the technology requires substantial set-up time.** The second condition is that it takes a long time to prepare to manufacture the product once a standard has been set. This condition places companies in a sort of Catch-22 situation in which losing the *de facto* standard is tantamount to giving the entire market over to the competitor closest to the final standard.
- **An older but serviceable technology exists for the same market.** This final condition means that there is a "release valve" for consumers who become cynical with the lack of standardization of the new technology. Even if it is substantially inferior, an older technology that provides serviceable support can leach support away from a much more impressive new technology that lacks full-market standardization

The first two conditions of high profit potential and difficult setup can tempt trailing companies to encourage market self-destruction. The incentives are very basic and can be summed up in an old schoolyard bully slogan: "If I can't have it, you can't either." The strategy is more than a simple emotional response, however. In particular, a company that already has a large market share based on older technologies (and which is trailing in a new one) may surmise that self-destruction of the new technology is the safest way to maintain an existing advantage. Unfortunately, consistent promotion of technology self-destruction by entrenched companies quickly degenerates into monopolistic behaviors that can seriously stifle innovation throughout an entire industry.

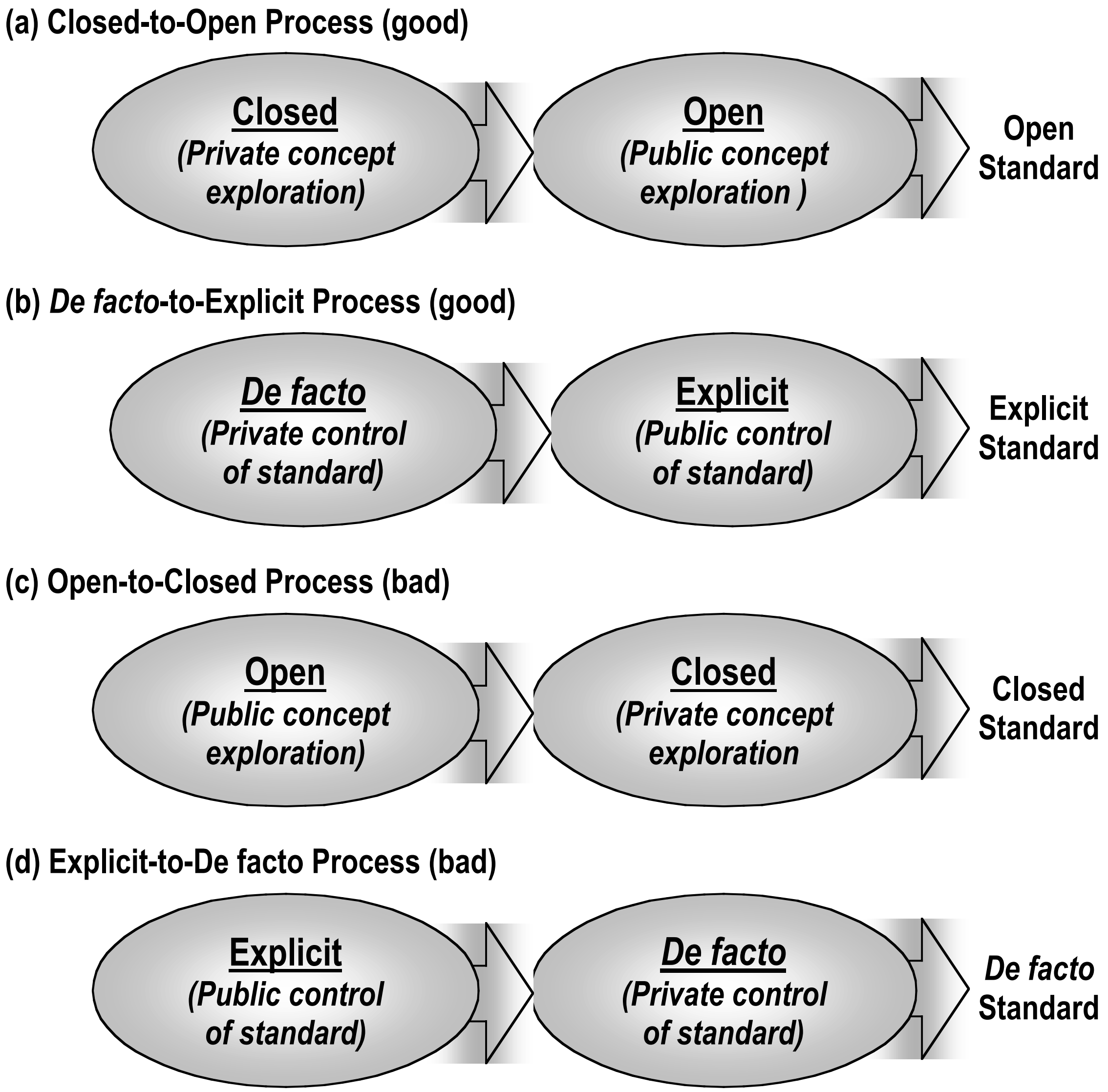
Encouraging market self-destruction works best in combination with the third condition of having technology alternatives. Having a serviceable older technology available makes it easier for companies to devise alternatives, and heavy investment in older technologies can make it difficult for new technologies to break into a market. The CRT (Cathode Ray Tube) display is an example of an older technology with major disadvantages in terms of weight, volume, security, implosion risks, complex manufacturing, and use of high voltages. If the CRT were introduced now as a display technology, its disadvantages would be so overwhelming that it would probably fail. In actuality, the CRT has benefited from so many decades of technical investment and cost-lowering mass production that it is difficult for newer and technologically superior approaches such as flat-screen displays to compete against CRTs in the general market.

Self-destruction of a new technology leads to one of two long-term outcomes. The first outcome is a delay of years or even decades in the deployment of the technology. Examples of such delays include laser videodisks and cellular telephony, both of which were delayed for over a decade by self-destruction of early standardization efforts. The second possible long-term outcome is permanent failure of the technology through its replacement with an even newer technology. An ongoing example of this outcome is the increasing popularity of recordable and rewritable compact disks (CD-R and CD-RW) for digital audio recording. While the easier alternative of digital audiotapes has been technically feasible for decades, broad use of that technology has been hampered by repeated self-destruction of standards. Given the low cost and high reliability of CD-R and CD-RW technologies, it is now possible that widespread use of digital audio recording tapes simply will never occur at all.

There are many interesting examples of technologies being delayed or failing altogether due to self-destruction of their associated standards efforts. A notable example of self-destruction and delay of a promising technology market was the first (1980s) attempt to market videodisks. Fierce competition between companies with proprietary formats resulted in an implosion of the market, and potential customers turned instead to magnetic tape based VHS (Video Home System) and BetaMax technologies. These tape technologies were less impressive technologically than videodisks, but they delivering good service to home users. Consequently, the advent of a commercially significant laser videodisk industry was delayed for well over a decade until the late 1990s release of the DVD (Digital VideoDisk) standard.

3.6 Effects of Transitions in Composite Processes

Basic standardization processes can be combined in many different ways to create composite processes. Figure 3-2 shows four important transitions that can take place within composite standardization processes. The first two transitions have generally positive affects, while the second two usually result in a deterioration of standards. When constructing a specific strategy for standardizing an emerging interoperability technology, it is advisable to avoid the latter two types of transitions if at all possible.


(a) Closed-to-Open Process (good)

(b) *De facto*-to-Explicit Process (good)

(c) Open-to-Closed Process (bad)

(d) Explicit-to-De facto Process (bad)

Closed
(Private concept exploration)
Open
(Public concept exploration)
Open Standard
De facto
(Private control of standard)
Explicit
(Public control of standard)
Explicit Standard
Open
(Public concept exploration)
Closed
(Private concept exploration
Closed Standard
Explicit
(Public control of standard)
De facto
(Private control of standard)
De facto Standard

Figure 3-2. Four Types of Transitions in Standardization Process

3.6.1 Closed-to-Open Transitions (Good)

Figure 3-2(a) shows a *closed-to-open transition* in which a company or individual decides to fully disclose the details of a previously private technology standard. This is most likely to occur when the owner of the technology sees more marketing potential from opening up a selected technology than from keeping it closed (e.g., because they feel it will result in wider use of a standard with which they are already intimately familiar). These transitions generally lead to more stable standards upon which new markets and technologies can be built more easily and more quickly.

3.6.2 De Facto-to-Explicit Transitions (Good)

Figure 3-2(b) shows a *de facto-to-explicit transition* in which a company or individual chooses to relinquish control of the standardization process to a group of competitors. This can be a difficult decision for a company to make, since there are financial benefits to retaining control of de facto standards. Such transitions are usually beneficial to the community as a whole, however, since they help stabilize and generalize the standard.

3.6.3 Open-to-Closed Transitions (Bad)

Figure 3-2(c) shows an *open-to-closed transition* in which a standard that is initially open and public is "captured" and converted into a proprietary standard. Open-to-closed transitions are usually accomplished through a gradual process of "standards pollution," in which a powerful company incrementally releases new or updated software components that deviate enough from the public standard to make it unusable. Open-to-closed transitions are costly and disruptive to users, and should be discouraged when they are done solely to capture a market. On the other hand, there are also cases where old or badly designed standards genuinely cannot meet market needs. In those cases, the prospect of an open-to-closed transition can provide a valuable incentive for businesses to provide real innovations.

3.6.4 Explicit-to-De Facto Transitions (Bad)

Figure 3-2(d) shows an *explicit-to-de facto transition*, which tends to occur in combination with an open-to-closed transition. Like open-to-closed, explicit-to-de facto is most likely to occur as a result of intentional "standards pollution" by a company that wishes to obtain full de facto control of an important public technology standard.

3.7 Recommended Standardization Processes

The next three sections provide recommendations for building standardization processes that avoid negative transitions and enhance positive effects.

3.7.1 Ideal Open Standardization Process

Figure 3-3 shows the ideal open process for standardizing a new technology. It is a composite process consisting of a fully open early phase of de facto technology exploration and trial use, followed by and open and explicit standardization process that tracks very closely to the proof-of-concept results of the de facto prototype. The Internet, which is the direct descendent of the DoD-sponsored Arpanet “prototype,” is the most impressive example of this style of standardization through evolution and extension.

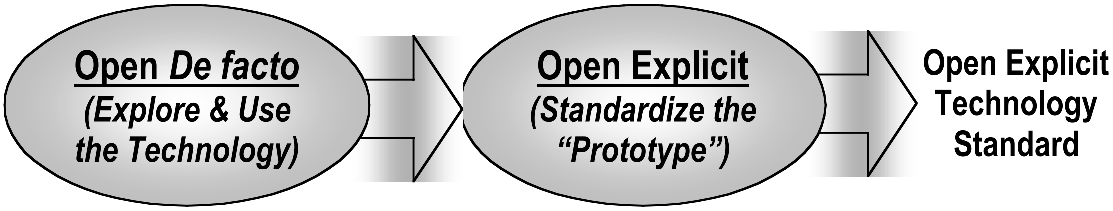

Open *De facto*
(Explore & Use the Technology)
Open Explicit
(Standardize the “Prototype”)
Open Explicit Technology Standard

Figure 3-3. Ideal Open Standardization Process

The ideal standardization process uses de facto exploration of the technology to encourage the greatest possible exploration of implications of the technology before full standardization. This phase also encourages creative inputs at the right point in the process – that is, before the final committee-style formal standardization process begins.

Making initial phase *open* improves the stability of the resulting standard by ensuring input from as large groups of potential technology users as possible. It also increases the opportunities for creativity and innovation during the “prototyping” part of the process, which can substantially improve the overall value of the resulting standard. The Internet is an especially powerful mechanism for implementing such open de facto standardization activities, since it can be used to reach a very wide audience of potential users. *Open source* methods can further increase the effectiveness of this phase. In open source technology exploration the source code for a new information technology is always be available to all interested parties, and no one group or individual can own the source code. Open source methods prevent “capture” of the technology by any one group during the exploration phase, and later helps ensure that companies focus on applying the technology instead of attempting to capture it in the form of a proprietary standard.

In this ideal process, the second (open and explicit) phase of committee-style standardization maintain a very low-risk approach. In particular, it should not attempt to introduce innovations that have not already been explored in the earlier prototyping phase, but should instead focus on selecting, generalizing, and fully documenting the most useful results produced by the earlier phase. The XML standardization effort is a good example of this approach, with its strong focus on selecting a subset of the most valuable features of the earlier SGML language while keeping compatibility with SGML tools and capabilities. The XML case also demonstrates that while the standardization activity may be conservative, a successful standard may then lead to a new cycle of prototyping and innovation in associated standards. For XML this second phase of , as has happened in the case of XML with a number of supporting technologies standards that have since developed about the stable XML core standard.

While the temptation to innovate can be strong during this phase, the lesson of efforts such as Ada 83 (a negative lesson) and XML (a positive lesson) seems to be that innovation should be done before and after the explicit standardization process, not within it. Technologies such as XML that have been explicitly designed to support later extensions and generalizations work particularly well in such an approach, since they provide good outlets both before and after the actual standardization effort for further innovation.

3.7.2 Ideal Closed-to-Open Standardization Process

While Figure 3-3 provides a strategy for fully open standardization, many standardized technologies begin as closed, proprietary concepts or technologies that only later become candidates for open standardization. Figure 3-4 shows a three-step modification of the ideal open process that provides a way to transition proprietary technologies into open standards.

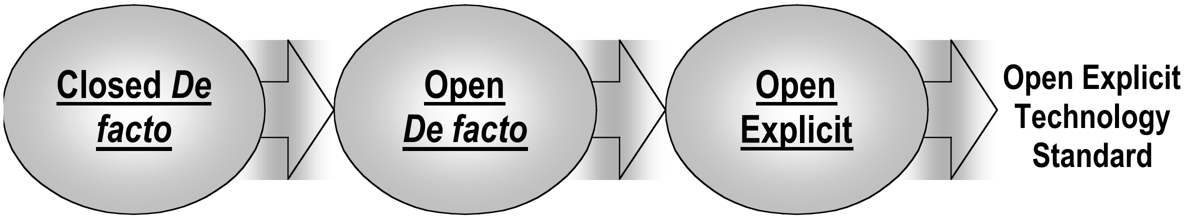


Figure 3-4. The "Mozilla" Process for Standardizing Proprietary Technologies

Standardization of closed, proprietary technologies can occur only if the company that originally owns the technology is willing to release full control of it. A company may release a technology if they feel that it will result in a highly profitable new market in which their profits will be higher than is possible if the technology is not universally adopted. Both audio and the new (DVD) generation of video laser disks provide examples of this incentive. Companies may also release technologies as a way of preventing competitors from dominating a market, or because the technologies are no longer directly profitable to them.

Whatever the incentive behind opening a proprietary technology to standardization, the three-step Figure 3-4 allows the implications of the proprietary technology to be explored in a more open forum before sending it on for committee-style explicit standardization. This approach can work especially well with software-based information technology that can be placed on the Internet for faster dissemination and exploration. The AOL/Netscape "Mozilla" effort to develop an open source web browser is an excellent example. Mozilla began with proprietary technology (the Netscape browser), transitioned the browser into a fully open development and testing process (the Mozilla effort), and as of early 2000 is encouraging the development of XML-based XUL (XML-based User Interface Language).

The Mozilla effort also demonstrates some of the disadvantages of this approach, since the original proprietary code base had to be largely re-written during the transition process.

3.7.3 Ideal Closed Standardization Process

Another option for standardizing technologies with high profit potential is the closed process shown in Figure 3-5. This composite process is exemplified by the Intelligent I/O (I2O) process, which extended proprietary technologies into a larger (but still closed) group of potential competitors without fully releasing it. This approach is in some ways a restricted version of the open-to-open model that uses a smaller group. The main disadvantage is the smaller size of the exploration effort and the unavailability of the resulting standard to all interested users. On the other hand, for technologies with high profit potential the closed model can provide a powerful commercial incentive for developing a strong standard.

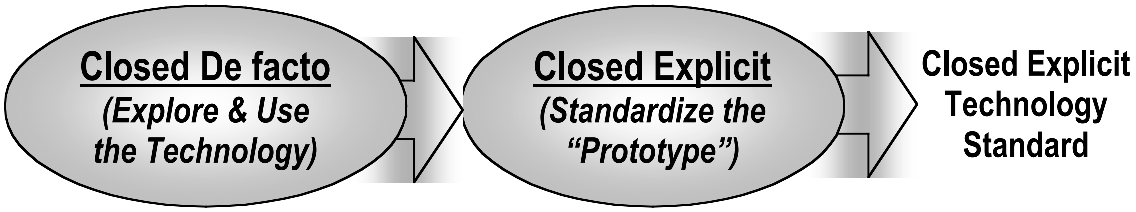

Closed De facto
(Explore & Use the Technology)
Closed Explicit
(Standardize the "Prototype")
Closed Explicit Technology Standard

Figure 3-5. Ideal Closed Standardization Process

3.8 Evaluating Technologies in Terms of Standardization Processes

It is easy to view technology standardization as being largely a matter of getting an appropriate standards body to accept responsibility for a technology. However, when evaluating early and rapidly changing technologies, factors need to be taken into account that extend far beyond the issue of finding a standards group.

As described in the section on the ideal standards process, a technology that stays "aggressively open" throughout its lifespan has the best chance for enduring and expanding, and so is more likely to be a good choice for interoperability support. Conversely, a technology that forfeits the benefits of open standardization in exchange for the nearer-term advantages of proprietary ownership and control is much riskier and likely to be much costlier in the end.

Many of the emerging interoperability technologies described in the rest of this document are in the early exploratory phases where innovation and hands-on use of preliminary tools need to be encouraged. Research organizations can help encourage this type of exploration by pushing the boundaries of such technologies can be used through innovative applications, and in some cases by developing tools that make it easier for others to access and try out the technology.

Section 4

Emerging Interoperability Technologies

4.1 Introduction

The purpose of this section of the report is to briefly summarize a number of emerging information technologies. The summaries focus on major concepts and comparative merits from the perspective of how they could support higher levels of interoperability. No prior knowledge of the technologies is assumed, but references are provided for readers interested in investigating specific technologies in more detail. The information-related emerging interoperability technologies covered in this section include:

- High-portability programming languages
- Java
- Jini
- JavaBeans and InfoBus
- EJB (Enterprise Java Beans)
- Microsoft COM (Common Object Model)
- DCOM (Distributed Component Object Model) and COM+
- Microsoft SOAP (Simple Object Access Protocol)
- WAP (Wireless Application Protocol)
- CORBA (Common Object Request Broker Architecture)
- Java RMI (Remote Method Invocation)
- XML (eXtensible Markup Language)
- COE (Common Operating Environment)
- Linux

4.2 Application Portability

4.2.1 High-Portability Programming Languages

What are high-portability programming languages?

High-portability programming languages are simply programming languages that work "the same" on a wide range of operating systems and hardware platforms. What makes portability difficult, however, is determining exactly what the phrase "the same" means for platforms that may have very different capacities, performance levels, and attached peripherals. For example, if the main purpose of a unit of software is to provide an easy-to-use graphical user interface, it may be very difficult to interpret such a program on older systems that lack sufficiently powerful displays, processors, or operating system software to handle the needs of the program. Similarly, programs that use detailed features of the operating system such as "threads" (a way of having multiple independent tasks running simultaneously within one program) may have difficulty executing fast enough or even correctly on a different or simpler operating system.

Historically, many of our main programming languages started out as efforts to provide high-portability programming. Ada is a particular example, but languages such as C, C++, Cobol, and even Fortran are examples of languages intended to be highly portable, also. However, it would probably be safe to say that all of these languages (including Ada) have fallen far short of their original portability goals.

A new twist on portable languages has arisen with the advent of the Internet during the 1990s. With its ability to pass large programs around very easily and to large to very large audiences, the Internet has provided both the incentives and the test bed needed to make programming languages more portable. Economically, highly portable programs have far more impact on the Internet, and so highly portable languages are attractive to developers of Internet software. From a testing perspective, distribution of programs over the Internet provides a ferociously effective way to validate both the program and the language in which it was written over a very diverse set of platforms and user needs.

These incentives have combined to encourage the creation of new languages that "live" on the Internet and are standardized not by committee, but rather by their level of success or failure at distributing software functionality over the Internet. Two notable examples of such Internet-first languages are Perl and Python, both of which are widely used on Internet servers. Older languages such as C, C++, and Ada have also benefited from *de facto* standardization through Internet distribution of powerful open source (that is, free with original source code provided) compilers that help make proprietary differences between compilers largely irrelevant by taking over much of the market for those languages.

One of the most notable and widely publicized Internet-oriented new languages, however, is the Java language from Sun Microsystems, which looks like a carefully selected subset of C++. Java is an *interpreted* language, which means that each platform that runs it must have its own virtual "Java computer" implemented to run Java programs. Although this is by no means a new concept for making languages portable, Java has benefited from the much more powerful machines now available, as well as from advances in compiler technology that help make interpreted programs more efficient. Unlike open source languages such as Perl, however, Java remains a proprietary language of Sun Microsystems. Java thus has probably has not benefited as much from Internet-based development and testing as other Internet languages, but instead relies on a Sun-sponsored certification program to ensure portability across platforms.

How are high-portability programming languages relevant to interoperability?

High-portability programming languages are relevant to any effort that requires the same software to exist on a wide range of platforms and operating systems. They are far less important in cases where all of the relevant computers use the same operating system and same general class of computers. In general, and especially for Internet-based software systems, the use of a high-portability language provides highly significant cost, time, and availability benefits by making it trivially easy to use the same software on different systems.

It should be noted that an important alternative to using highly portable languages is instead to standardize the data and processing requests that are passed between different types of software and systems. Such *middleware* approaches have their own difficulties, but they can be a useful alternative when the use of high-portability programming languages is not practical.

What are the best opportunities for using high-portability programming languages?

High-portability programming languages are necessarily one of the foundation blocks of building highly interoperable systems, especially as Internet-based, multi-computer applications become more common. Java in particular is important both because of its similarity to the popular C++ language, which makes it easier to find or train Java programmers, and because of the extensive effort that has gone into it in the past few years to develop closely associated interoperability technologies. These associated technologies include Jini (which is intended to support the dynamic composition of software services, including those provided by networked devices), JavaBeans (to make graphical programming and interfaces easier), and Enterprise JavaBeans (to extend use of Java components to the level of entire enterprises.)

Other high-portability languages, in particular Internet-based languages such as Perl and Python, should also be monitored and probably utilized where appropriate, since these languages have become a surprisingly large part of the overall Internet infrastructure. Similarly, Internet-standardized versions of languages such as C (e.g., the GNU compiler) and C++ may also provide valuable assets to efforts that require extensive interoperability, since they often execute on the same platforms as proprietary compilers, but generally lack the extensions and exceptions that make such proprietary compilers non-portable.

References

http://www.sun.com/java/platform.html, “What Is the Java Platform?”

http://language.perl.com/index.html, “Perl Language Page.”

http://www.python.org/, “Python.”

http://www.gnu.org/software/gcc/gcc.html, “GNU C Compiler.”

4.2.2 Java

What is Java?

Java is a compact programming language that is mostly a subset of the popular (but much larger) programming language known as C++. Java is an "interpretive" language, meaning that its programs are compiled down to machine-independent "byte code" instead of the native machine language of a computer. The fact that Java uses byte code rather than machine-dependent assembly code makes it possible to run the same compiled Java computer on nearly any type of computer. However, it often means that Java applications are somewhat slower, since the byte codes must be interpreted while the program is running. Performance has been an area of substantial progress as the Java virtual machines and compilers mature, and there is strong evidence that this maturation will continue. However, performance is clearly an issue that must be carefully managed from the start and monitored as a risk management priority for any performance sensitive application using Java.

Java is easier language to use than C++. Firstly, it is a much smaller language than C++, with fewer parts to learn. Secondly, Java automatically takes care of many important but error-prone "housekeeping" chores such as managing memory resources, which allows programmers to worry less about managing the internal resources of a computer and focus more on their primary programming tasks. Thirdly, it is so similar to C++ that programmers can learn to read Java with little or no additional training. Finally, Java is an example of a *consistently* "object-oriented" programming language, which means that the only way a programmer can develop software in Java is in the form of software "objects."

(A software *object* is simply a structured collection of data, such as "employee address data," for which there is a clearly defined set of "operations" that can be performed on it, such as creating, printing, or updating the data. This is in contrast to more traditional "functional" programming, in which programmers first create functions to process data and only then worry about how data will be structured and passed around. If used consistently, object-oriented software design can result in software that is easier to understand, more compact, and less costly to maintain than the equivalent functional code. When object-oriented methods are used inconsistently or incorrectly, however, they can produce rather spectacular programming disasters. The purely object-oriented style of Java has a distinct advantage over C++ on this point, since it makes it much harder to mix object and functional styles. In contrast, C++ allows (and even encourages) risky mixing of functional and object-oriented style of programming.)

Java is not just a language. Associated with it are standards such as *Java Beans* and *Enterprise Java Beans* that help prescribe how to write "composable" sets of Java components that can be combined together even when they were originally created by different groups. Another associated software technology is called *JavaScript*, which is actually not a version of Java at all, but a similar, complementary language is easier to embed in web pages. Finally, Java was defined specifically designed for use as a platform-independent programming language for the Internet, so that it is most often seen in association with programs such as Internet web browsers and other Internet technologies. For example, small Java programs called “applets” are often used to add custom data processing capabilities temporarily to remote Internet web browsers.

How is Java relevant to interoperability?

Java is probably the most promising computer language for increasing the ability of computers to share software easily across a heterogeneous network of many different types of computers. Its use of computer-independent byte codes means that the same compiled Java program can usually execute on many different computers and under many different operating systems. Furthermore, its popularity and similarity to C++ make it a much stronger candidate for general adoption than are other similar byte-code languages (e.g., the venerable and ruthlessly object-oriented language known as Smalltalk).

Use of byte-codes also means that Java is more *mobile* than most other forms of software. That is, a Java program can often be “moved” over a network to the location (such as an Internet user’s web browser) where it can work fastest and with the least overhead. This ability to move Java programs to where they are most needed is the software equivalent of a rapid deployment force, and may in time allow naval operations to shift data processing resources around quickly enough to handle rapid changes in the battlefield configurations and status.

The portability of Java programs is further enhanced when they use standards such as Enterprise Java Beans to ensure their overall ability to interoperate with other Java components.

What are the best opportunities for using Java?

Java clearly needs to be researched and developed as an important (possibly the most important) software language for implementing mobile and interoperable software components. In particular, tracking and exploration of how to use Java and its associated Internet technologies in web-based applications looks like promising research and development path for making new systems both more interoperable and more adaptable to specific battlefield situations.

Java also appears to have good potential as a “wrapper” technology for making legacy systems appear to be coded in a more Internet-friendly language – that is, to make them look like Java applications. For example, Java applets can act as front ends for accessing the capabilities of a valuable but cumbersome remote legacy system. Remote Internet users can then access those legacy systems without having to have any knowledge of the special access mechanisms that are hidden away in the Java applet.

References

http://www.sun.com/java/platform.jhtml, “What Is the Java Platform?”

http://www.sun.com/java/javameansbusiness/, “Java™ Technology Means Business”

4.2.3 Jini

What is Jini?

Jini (which is a name, not an acronym) is an application of the Java language to the problem of how to interconnect diverse types of electronic devices and more generally any provider of computing services (from printers to GPS receivers to any kind of algorithm that is provided as a service) with little or no manual intervention. It is essentially a set of standards produced by the Sun Corporation for creating electronic devices that are self-installing, self-configuring, and self-diagnosing. The focus of Jini on device self-management makes it potentially powerful tool for interoperability, since lack of information about how a device works is one of the most fundamental road blocks to making it work together with other devices. Bill Joy, the key architect of Jini, defines it as "the BIOS for the age of networked devices".

The first major Jini concept is that of *service self-description*. Every service in a Jini system is required to have in effect a "resume" (written in Java) that defines what that service can and cannot do, and how to use the service. The Jini concept relies heavily on the object-oriented programming concepts to do this, since the description of the device looks like a Java software object. Services are often associated with devices (e.g., a printer provides printing services).

The second major Jini concept is the idea of a *lookup service*, which is essentially an electronic matchmaker that stores definitions of all the known Jini devices in a system or network. The lookup service helps devices recognize each other's presence, and then provides the devices with all the information needed for them to communicate directly. After the initial connection has been made, the devices thus can communicate without any further use of the lookup service.

Jini assumes that most of the devices using it will be "smart" in the sense of having their own built-in processing and storage capabilities. However, the presence of the lookup service means that Jini can also accommodate older "dumb" devices by storing their resumes in the registry. This allows other "smart" devices to retrieve the information for how to access an older device.

Sun's Jini program is still relatively new as of early 1999, and the level of adoption that it will see in the commercial world is still uncertain. However, even if Jini does not prove to be widely adopted, the principles that it represents are already widely acknowledged by the commercial world as being crucial to sales of the next generation of smart appliances and devices. Thus even if Jini does not become that standard, some other similar set of standards will almost certainly take its place. Similarly, Java has an excellent chance of becoming the underlying programming language for representing Jini "resumes," since it is both popular and was originally designed for just such smart appliance applications.

It should be noted that while the basic concepts of Jini are straightforward, there are many, many unsolved problems implied by the Jini scheme. In particular, simply describing an interface at the level of a Java interface does not in general provide enough information for another device to understand fully what the device is capable of doing. Jini will presumably take care of many cases by defining a variety of standard interface types (such as "disk drive" or "mass storage device"), but detailed descriptions of the operating characteristics of some devices will also be needed. Even more generally, some device types will be completely unknown to older Jini devices, and so may require even more abstract descriptions of their functions. It is not clear yet how far Jini will be able to move in such directions. Even for current devices, more complex standards will be needed, and the interoperability implications of different sets of such standards are not currently well know.

How is Jini relevant to interoperability?

Jini is immediately relevant to hardware interoperability due to its focus on helping electronic devices to integrate together and share data more easily. Jini is particularly relevant to the widely anticipated "next revolution" in smart appliances (versus the current PC revolution), and in that role it may help spawn a wide variety of new off-the-shelf hardware components that will be far easier to integrate than is typical of the current generation.

Jini also has potential as a structured approach to "retrofitting" aging hardware components with newer, smarter Java-based interfaces that will be easier to access and use.

What are the best opportunities for using Jini?

If for no other reason than that it is probably the current leading candidate for how to implement a strongly anticipated next generation of smart commercial appliances and devices, Jini clearly merits attention as a Navy interoperability technology. As a research and development topic, Jini is rich in both pragmatic application issues and advanced research problems. On the pragmatic, development-oriented end are how to standardize and apply interface definitions for widely used classes of commercial and military hardware, while on the research side there are many issues regarding how to add arbitrarily complex "new" devices to a Jini network. Additional concepts such as explicit hierarchies of metadata to describe new devices may be needed, as well as work on how to resolve conflicts and ambiguities. Another potential research topic would be how to apply Jini concepts effectively to "wrapping" legacy Navy systems for easier future access, and whether such methods could be at least partially automated to speed conversions.

References

http://www.sun.com/jini/overview/,
"Jini™ Connection Technology Executive Overview"

http://www.sun.com/jini/faqs/index.html,
"Jini™ Technology Frequently Asked Questions"

4.3 Component-Based Software

What is component-based software?

Programming in conventional English-like computer languages is slow, and the constructs and statements that are used to create a function usually bear little or no resemblance to the resulting operations. Component-based software is simply the idea that if the programming process could somehow be made to "look like" the actual functional results, the entire programming process could be made faster, simpler, and much more reliable.

The problem is that for many types of software (e.g., mathematically intensive software), there are no few obvious ways of "attaching" final results back to the software in a way that would be meaningful for changing the software. On the other hand, there are also large and important classes of software for which there is a very strong link between the visual results of software and the way that software is used and configured. In particular, development of *graphical user interfaces,* or *GUIs,* is very amenable to this type of higher-level, non-linguistic programming.

Consequently, the phrase *component-based software* most often refers to a style of "visual programming" in which components with well-defined graphical outputs can be manipulated and combined on a display screen, often without having to use conventional coding at all. It can also refer to a more conventional style of programming in which large, predefined software components are used as objects or procedures that can be combined quickly and easily to create new programs. In both cases, the objective of component-based programming is to start the programming effort at a much higher, more results-oriented level of capability. Stated another way, the common thread throughout component-based software development is that one should always begin the programming process using powerful, predefined modules, and should use conventional source code only as a last resort.

How is component-based software relevant to interoperability?

While component-based software has obvious advantages when building new systems, its connection to interoperability is a bit subtler. One way that it contributes is by making it easier and less painful to phase out older, less interoperable systems by allowing rapid recreation and verification of their capabilities on a new system. Another way in which component-based software contributes is that it can be used to spread standardized over an entire network much more quickly than would have been possible with non-component software. Finally, component-based software can be designed to "accommodate" significant differences in the underlying hardware and operating system architectures of different systems. An example of this is the JavaBeans component architecture, which uses the portability of Java to hide differences between systems and make new software more immediately portable and interoperable.

What are the best opportunities for using component-based software?

Perhaps the most important reason for using component-based software in the Navy is that it provides a powerful mechanism for building up and maintaining higher levels of interoperability across systems.

References

http://www.sei.cmu.edu/str/descriptions/cbsd.html,
"Component-Based Software Development / COTS Integration"

4.3.1 JavaBeans and InfoBus

What are JavaBeans and InfoBus?

JavaBeans are small software units that are written in Java and are capable of being modified and combined graphically to create new graphical user interfaces (GUIs) and applications. Software units can be called JavaBeans only if they meet a set of standards for reusability, modifiability, and graphical support. *InfoBus* is an extension of the JavaBean standards that provides a fast, uniform mechanism by which JavaBeans can broadcast data items or arrays of data items to each other, provided that the components are on the same computer system.

Historically, JavaBeans are Sun's answer to the earlier success of Microsoft visual components, such as the components provided in Microsoft's Visual Basic language. In both cases, a major objective was to dramatically reduce the amount of time required to create a graphical user interfaces when compared to older hard coded approaches to creating such interfaces. With graphical components, building a user interface becomes little more than a matter of selecting components, modifying them graphically, positioning them on the screen, and linking them to other components. Graphical component programming also benefits from the ease with which developers can experiment with the interfaces that they create. This makes possible a better fit to actual user needs than is possible with hard-to-change traditional interfaces.

While JavaBeans are not restricted to being only graphical "widgets" such as response boxes in a graphical user interface, modifying and combining JavaBeans (or any other type of visual component) becomes substantially more difficult and complex as the processing behavior of the JavaBean becomes more complex. This is because the easy visual analogies that make JavaBeans simple to use for creating graphical interfaces become more strained as the connection between the screen and the type of processing needed becomes more remote. For this reason, the easily observed productivity gains that come from using visual components to construct graphical user interfaces should not automatically be assumed to apply to the construction of more complex, algorithmically complex software (e.g., server applications). JavaBeans and their Microsoft equivalents thus have substantial value as programming tools, but it is a value that is largely constrained to the domain of graphical user interface design.

Compared to JavaBeans, Microsoft visual components have the advantage of being integrated with and closely tied popular Microsoft products such as the Windows operating systems and the Microsoft Office suite. However, Microsoft components are based on the Component Object Model (COM), which as a unique software communication scheme that is fully supported only by the Microsoft Windows family of operating systems. JavaBeans have the advantage of being more broadly supported across a range of platforms, since they are implemented in Java instead of with proprietary formats.

The *InfoBus* extension of JavaBeans standards increases the power of JavaBeans substantially by giving them a simple, fast, and standardized way of broadcasting data items and arrays of data to each other. It is important to note that InfoBus only applies to JavaBeans that reside on the same computer; it is not at present a distributed or multi-computer standard. This restriction has the advantage of allowing very rapid, memory-based exchange of data between JavaBeans, but it also has the obvious disadvantage of strictly limiting the range over which JavaBeans can communicate using the InfoBus.

The InfoBus is conceptually similar to an electronic bus such as those used in a computer. *Producer* JavaBeans place information on the InfoBus, from which any (or all) of the *consumer* JavaBeans on that InfoBus can then pick up the information. This approach assumes that all the JavaBeans on an InfoBus have the same level of trust, since any JavaBean on that InfoBus can in principle read any information placed on that bus. However, it is also possible to create more than one InfoBus in the same application, so that groups of JavaBeans can be created on the basis of what type of information they need to exchange over their shared InfoBus.

How are JavaBeans and InfoBus relevant to interoperability?

For situations that require interoperability across operating systems and platforms, JavaBeans and its InfoBus extension have the clear advantage of being based on the comparatively platform independent Java language. However, it should also be noted that for situations where all platforms use Microsoft Windows, the alternative of using Microsoft components (e.g., via Visual Basic) is probably stronger. Within a Windows-only environment, Microsoft components are more tightly integrated with the operating systems, and at least somewhat faster because they use compiled code instead of interpreted Java. In both cases, however, component-based methods for constructing graphical components have strong cost and quality advantages over older and more laborious approaches of hard coding such interfaces.

What are the best opportunities for using JavaBeans and InfoBus?

For the Navy, use of JavaBeans and InfoBus is necessarily closely tied to use of Java. Use of JavaBeans should also be assumed for any program that is looking into Java for interoperability. For programs that are interested exclusively in interoperability across Microsoft Windows platforms, the focus should probably be on COM-based Microsoft components, although the uses of Java there should not be overlooked. In either case, the importance of using component based approaches to create graphical user interfaces should be taken as a given because of the much higher productivity and reliability of component based approaches for such systems.

A potentially high-value (but difficult) research topic in this area would be to look into ways to extend the JavaBeans graphical programming model to encompass more complex forms of programming. The relationship of such research to interoperability is that it could help configure or adapt systems quickly to work in new environments or military situations. The most powerful approach could be to develop JavaBean sets that specifically address interoperability issues, such as JavaBeans that could be used to quickly set up graphical interfaces to incompatible or legacy computer systems. Research challenges in such an approach would include the need to develop a sufficiently powerful "language" (set of complementary components) to describe the full range of Navy system compatibility issues, and (even more difficult) how to connect the resulting new interface descriptions into the legacy software with a minimum of new programming.

Developments in the InfoBus standard should also be tracked over time, since the InfoBus may eventually be extended to cross computer platforms. If InfoBus is extended in this fashion, it could provide a powerful cross-platform interoperability capability by making exchange and remote display of data significantly easier. If such extensions do occur, they are likely to be based on existing standards for distributing calls, such as Common Object Request Broker Architecture (CORBA) or Java's own Remote Method Invocation (RMI), and so would represent a convention way of using one or more of those standards.

References

http://java.sun.com/beans/FAQ.html,
"JavaBeans™ Frequently Asked Questions."

http://java.sun.com/beans/infobus/,
"JavaBeans™ FAQ: InfoBus."

http://java.sun.com/products/jdk/rmi/index.html,
"Java™ Remote Method Invocation (RMI) Interface"

4.3.2 Enterprise Java Beans (EJB)

What is Enterprise JavaBeans?

Enterprise JavaBeans, or EJB, is a set of standards for how to write server applications in Java so that peripheral, non-enterprise-related programming issues are kept to a minimum. The goal it to make server applications highly portable and scalable by eliminating many of the issues that tend to "lock down" an application to a single server or operating system architecture.

Despite similar names, Enterprise JavaBeans and JavaBeans have little in common. EJB is largely non-graphical, application-oriented, and is intended to reside on the server side, while JavaBeans are interface-oriented and reside on the client side. Their only real connection is that they are intended to complement each other by allowing both the server and client sides of applications to be written in Java.

How is Enterprise JavaBeans relevant to interoperability?

EJB classifies as a component technology in the sense that it allows application programming to be done at a higher and more platform-independent fashion than would otherwise be possible. Its main advantage to interoperability is that it should allow applications written to EJB standards to be more easily ported to a wide range of platforms, so that a wider range of system can use those applications.

What are the best opportunities for using Enterprise JavaBeans?

EJB is an interesting and relatively young effort that is well worth watching as a possible approach to creating platform-independent server applications that promote interoperability.

References

http://java.sun.com/products/ejb/, “Enterprise JavaBeans™ Technology.”

http://java.sun.com/products/ejb/faq.html, “Enterprise JavaBeans™

4.3.3 Microsoft COM (Common Object Model)

What is Microsoft COM?

The Microsoft Component Object Model (COM) is the basis for what has arguably been the most successful component-based programming system ever created. COM is a set of binary-level standards for how compiled components can call each other and exchange data with each other within Windows 95, 98, and NT operating systems. It is used heavily throughout both these three operating systems and in many (probably most) of the applications that run on them.

COM makes possible the visual style of programming seen in Microsoft tools such as Visual Basic. Major Microsoft Windows applications such as Word, Excel, and PowerPoint also have COM interfaces that flexibly exchange data and processing requests with other applications. It is COM, for example, that make it possible to define an array of data in an Excel spreadsheet and then display it within a complex 3D bar chart in PowerPoint.

Despite these strong positives, COM also has a number of conspicuous limitations. For example, the fact that it is a binary standard makes it necessary to use it indirectly through other tools and macros. This is inconvenient, since users cannot simply specify in a standard language how they want an interface to appear. However, the single most important limitation of COM is that it is very closely tied to the Microsoft Windows series of operating systems. As a binary-level interface, it is also difficult to extend meaningfully to other operating systems without ending up simply replicating large chunks of the Windows operating system.

Finally, when discussing COM it is important to recognize that it is distinct from Microsoft's Distributed Component Object Model, or DCOM. The objective of DCOM is to extend COM across multiple systems on a network, and it uses a distinct set of communication technologies not seen in COM itself to accomplish that. DCOM is used mostly in small LAN environments, and is seldom (as of the late 1990s) used in large, highly distributed applications. The weak record of DCOM is particularly conspicuous given the strong success of COM.

How is Microsoft COM relevant to interoperability?

Within a Windows 95, 98, or NT based system, COM is a valuable technology for building new applications rapidly. An indirect interoperability benefit of COM is that many remote invocation technologies (e.g., DCOM and CORBA) target COM to allow remote software applications to use components on Windows systems. This can be highly beneficial to interoperability, since it can allow non-Windows systems to access and use certain aspects of Windows applications.

What are the best opportunities for using Microsoft COM?

COM should be viewed primarily as an important component technology that is already firmly entrenched within Windows systems. Tools that make it easy to access COM components should be tracked and possibly actively encouraged as a good approach to enhancing interoperability between Windows and non-Windows systems.

References

http://www.microsoft.com/com/, Microsoft home page for all COM technologies.

http://www.microsoft.com/com/tech/com.asp, COM – Component Object Model.

4.4 Middleware

What is middleware?

Middleware is software that makes it easier for different types of software at different locations in a network to call each other and exchange data. In general, it does this by providing both a standardized calling format and a special communication protocol for conveying calls and data across a network. Middleware calling formats often closely resemble the formats used to invoke objects or procedures in ordinary programming languages, and the communication protocols are often based on widely used protocols such as the Internet's TCP/IP.

How is middleware relevant to interoperability?

Middleware is a fundamental interoperability concept. For example, when it is not practical to redevelop legacy applications, middleware often provides the best overall approach to getting incompatible systems to interoperate with each other.

What are the best opportunities for using middleware?

Middleware is particularly appropriate for "wrapping" older, hard-to-change legacy systems in a new set of interfaces that then can be used to interact with other systems. It can also be used to create new "distributed" applications – that is, applications that are designed to reside on more than one computer in a network.

References

http://dii-sw.ncr.disa.mil/coe/topics/atd/recommend/all.doc,
"Recommendations for Using DCE, DCOM, and CORBA Middleware"

4.4.1 DCOM (Distributed Component Object Model) and COM+

What are DCOM and COM+?

DCOM is middleware from Microsoft for making COM component available across a network. It is essentially an application of an older middleware technology called DCE (Distributed Computing Environment) to the specific problem of supporting COM interfaces. Microsoft generally does not emphasize the DCE roots of DCOM, preferring instead to focus on it as an extension of COM. (DCE itself is essentially obsolete, with few prospects for being implemented on any new systems.)

COM+ is one of several marketing terms that has been promoted by Microsoft to describe its next generation of both its COM component technology and its DCOM middleware technology. It is difficult to obtain non-marketing definitions of what COM+ currently represents technically.

How are DCOM and COM+ relevant to interoperability?

As a middleware technology, DCOM is relevant for connecting systems with Windows 95, 98, and NT over a network. It is not particularly practical or useful as a way of connecting into non-Windows systems, since it expects to "find" COM components on the remote systems. Such COM components are generally impractical to implement on non-Windows systems. A better approach is usually to use tools that bridge from more universal middleware technologies (e.g., CORBA) and go directly to COM. Of particular interest over the first few years of the new millennium will be the Microsoft and IETF supported and XML-based Simple Object Access Protocol, or SOAP (see below). In sharp contrast to DCOM and COM+, SOAP is readily portable to non-Windows architectures and has good support from a wider range of communities.

What are the best opportunities for using DCOM, and COM+?

DCOM and COM+ clearly should be tracked, but the recent (late 1999) release and strong promotion by Microsoft of a new XML-based Simple Object Access Protocol (SOAP) protocol (see below) for the same general market area may be a strong indication that the days of DCOM and COM+ as anything other than marketing labels may be numbered.

References

http://www.microsoft.com/com/,
Microsoft home page for all COM technologies.

http://www.microsoft.com/com/tech/DCOM.asp,
DCOM – Distributed Component Object Model

http://www.microsoft.com/com/tech/COMPlus.asp,
COM+ – Component Object Model Plus.

4.4.2 CORBA – Common Object Request Broker Architecture

What is CORBA?

CORBA is a middleware technology with broad support in the software industry. It uses an "object-oriented" approach to transferring data and calls between software modules, which means that functions (procedures) are specifically associated with definite data types. CORBA uses calling formats that are similar to that of the object-oriented language C++, which makes it easier to use with that language. CORBA is more difficult to use with non-object languages such as C, since in effect it requires that the interface objects be constructed manually.

CORBA currently (late 1999) does well at interoperability across vendors, and, in contrast to Microsoft DCOM, CORBA is available on a wide range of platforms.

How is CORBA relevant to interoperability?

Overall, CORBA is probably the best current middleware candidate for getting a diverse range of systems to interoperate with each other, and it is also the strongest candidate for "wrapping" older legacy systems to make them accessible to newer systems and networks.

What are the best opportunities for using CORBA?

CORBA is an important interoperability technology that merits both direct application to existing Navy interoperability problems, and research into how it might be used to solve more complex classes of interoperability problems. The Internet-based CORBA protocol for transmitting calls and data over the Internet (IIOP) could be particularly useful for getting systems to interoperate over the web.

Any use of CORBA, however, should first be checked against the potential of the new Simple Object Access Protocol (SOAP) that is being promoted by both the IETF and Microsoft. Unlike CORBA, SOAP is XML based and thus portable to any system that can use a text-processing language such as Perl to implement SOAP parsing. On the other hand, SOAP is a communications protocol that competes directly only against the communications (IIOP) component of CORBA, rather than against the interface definition language (IDL) that CORBA uses to specify access into objects. CORBA thus could potentially use SOAP as an alternative to its own IIOP. As of early 2000, however, no such option has been defined for CORBA.

References

http://www.whatis.com/corba.htm, What is CORBA? (a one-page definition)

http://www.omg.org/, "OMG – Object Management Group"

http://www.aurora-tech.com/corba-faq/toc.html, "CORBA FAQ Overview"

4.4.3 Java RMI – Java Remote Method Invocation

What is Java RMI?

RMI is simply a way of extending Java calls across a network, in much the same way that DCOM is a way of extending COM calls across a network. RMI works only within Java, but since Java is available on a wide range of platforms, this means that RMI can be use to link software on many types of systems .

How is Java RMI relevant to interoperability?

RMI can help link remote systems using Java together, and so classifies as a technology for increasing the interoperability of such systems. It is limited to Java only, however.

What are the best opportunities for using Java RMI?

For a Java programmer, RMI is significantly easier to use than CORBA, since it works within the normal framework of the Java language and does not require re-definition of any interfaces. Unlike CORBA, however, the Java-only RMI technology does not provide for any interaction with other non-Java software modules. As an interoperability technology, RMI may be the easiest way to achieve interoperability between Java applications on different types of machines.

References

http://java.sun.com/products/jdk/rmi/index.html,
"Java™ Remote Method Invocation (RMI) Interface"

4.4.4 XML (eXtensible Markup Language)

What is XML?

XML (eXtensible Markup Language) is closely related to HTML, the language in which Internet web pages are generally implemented. In HTML, readable “tags” are used to specify how a particular piece of text should be displayed on the screen – e.g., the tag “<p>” can be used to specify where a new paragraph begins. However, HTML tags can only be used to define how to display text; they do not say anything about the actual content of the marked text.

XML simply broadens the concept of a tag to include arbitrary, user-defined tags that can then be used to specify what kind of text is being marked, rather than just how to display it. Thus, it would be possible to define a new tag called “<integer>” to mark text that should be interpreted as an integer number. There is no upper limit to how complex the meaning of a tag could be, so XML allows the specification of complex content information.

It should be noted that XML does not in any way solve the problem of how to define, standardize, use, or display data. All it immediately does is provide a standardized way to represent content tags.

How is XML relevant to interoperability?

One major impediment to interoperability between many types of systems is a lack of a standard way to describe the format and content of data. One system may use binary data, another may use structured text, and another may use a different style of structured text. What XML provides is standard way of expressing data formats for transmission over a network, although it does not by itself tell how those formats should be interpreted.

What are the best opportunities for using XML?

The current market situation for XML is excellent. Many companies (including Microsoft) are making significant commitments to using it in their products and infrastructures, and important affiliated standards for using XML in specific application areas are appearing at a surprisingly rapid pace. XML is benefiting both from its close ties to the earlier Internet success of HTML, and from the ease with which it can be read and processed.

XML should be viewed both as an important immediately available tool (e.g., via supporting COTS products such as XML-enable web browsers) for increasing data interoperability between systems, and as a longer-term research issue for how to make effective use of its capabilities.

References

http://www.ucc.ie/xml/#FAQ-GENERAL, “XML FAQ – A. General Questions.”

http://www.w3.org/XML/, “Extensible Markup Language (XML^{TM})”

http://msdn.microsoft.com/xml/default.asp, Microsoft XML Developer Center

4.4.5 XML Metadata Technologies

What are XML metadata technologies?

As described in the previous section, XML (eXtensible Markup Language) is closely related to HTML and provides a highly generalized way to attach "tags" to textual data. One particularly important application of XML is as a mechanism for adding *metadata*, which simply means "data about data," to data or software. Metadata is important to interoperability because it can be used to provide a context or overall understanding of how data should be used or interpreted, and thus of how it can be made to interoperate across diverse systems.

For example, metadata that tells which version of Java is being used in a module is immediately helpful in making that module interoperable by allowing anyone receiving it to know whether and how to interpret the associated Java code. A more complex example of metadata is the registry system of Windows operating systems. The Windows registry is a structured repository of shared metadata that allows applications to know what services and capabilities are available on that system. The applications can then use that metadata to work within the constraints of that particular system.

It should emphasized that metadata is itself a form of data, and so consists of bits, bytes, and characters just like regular (or *referential*) data. The difference is in the topic addressed by those bits and bytes. Regular or referential data measures aspects of physical (e.g., sensor) or logical (e.g., financial) systems that exist independently of the computer system in which the data resides. In contrast, metadata measures aspects of data sets that exist within a computer system, such as referential data files collected from a sensor or a financial system. It is also possible to have metadata that describes other metadata, since metadata is itself a form of data. Multiple layers of metadata can be useful for providing increasingly broad layers of context in which to interpret data or software applications.

Metadata can be either automated or manual. *Automated metadata* is metadata that can be created and used entirely by software, without any direct human intervention. Database index files are an example of fully automated metadata, since they are generated and used without requiring an interpretation by people. *Manual metadata* is usually associated with complex tasks that require more sophistication than is possible with current levels of automation, and for that reason it is often descriptive or text-based in nature. Design comments in a software source file are an example of manual metadata that requires interpretation by humans before it can be used effectively.

Another important distinction is how metadata will be used. *Optimization metadata* is used to make processing of the original data more efficient. Database index files and data caches are both examples of optimization metadata. Because optimization metadata often can be derived directly from the original data set, it tends to be easier to automate and use. *Semantic metadata* is a form of metadata that is more difficult to automate, and more relevant to interoperability. The objective of semantic metadata is to make the original data “meaningful” or usable in a broader range of contexts than would normally be possible. Version tags, Windows registries, and source code comments are all examples of semantic metadata, since they all help make applications usable in a broader range of contexts. While metadata can sometimes be derived directly from original data such as source code, in most cases it must be captured from external sources at the same time that the original data is being collected. Once omitted, lost, or damaged, semantic metadata can be extremely hard to rebuild or recreate.

XML is relevant to metadata because of its ability to add tags and structured text to data items. Both the tags and their associated data can be used to associate arbitrary new data with existing data, and that new data can represent (for example) semantic metadata about the interpretation and use of the original data. Another advantage using XML for metadata is that it provides a uniform way to represent both fully automated metadata (e.g., numeric data with a precise interpretation) and manual data (e.g., textual comments). Consequently, interest in XML has also given rise to a variety of related technologies that address the issue of how to express and associate metadata with referential data.

As of mid 2000, several XML technologies deal with various aspects of metadata. The XML technology that is most specifically and generically associated with metadata is RDF (Resource Description Framework). RDF was explicitly designed to support the addition of metadata to the Internet, which is difficult problem because the Internet is both decentralized and global. RDF is particularly oriented towards adding optimization metadata that can be used (for example) to speed the process of querying the Internet. However, the RDF framework may also prove to be suitable for adding other types of metadata (semantic metadata in particular) to networks that use Internet technologies.

A second fundamental XML technology related to metadata is XML Schema. As of mid 2000, XML Schema is an extensive emerging standard for defining valid XML constructs and specifying acceptable data for those constructs. XML Schema is a more powerful replacement for the older DTD (Document Type Definition). In contrast to the older DTD technology, XML Schema can be used to create new XML document types dynamically, verify data validity, and uses standard XML syntax. These capabilities make XML Schema an important tool for enabling the addition and use of metadata, since it makes it easier to create new schemas that provide or add metadata to existing XML document types. XML Schema is a more generic tool than RDF (see paragraph above), since it is intended for full definition of XML documents and not just metadata.

Another specialized XML metadata technology is XMI (XML Metadata Interchange). Despite its name, XMI is intended more for the exchange of object-oriented design and business process information than for the exchange of generic metadata. XMI was developed the Object Management Group (OMG), which is the same group that is responsible for the Common Object Request Broker Architecture (CORBA) middleware architecture, and it is used to exchange objects based on the OMG Object Analysis and Design Facility. Such objects are more commonly called UML (Unified Modeling Language) models and MOF (Meta Objects Facility) objects, and the type of metadata that they represent is abstract descriptions of object-oriented designs. XMI is probably best viewed as more as an XML-based support tool for UML and object design than as a generic tool for expressing metadata.

How are XML metadata technologies relevant to interoperability?

As XML metadata technologies and standards develop over time, they should provide a range of useful off-the-shelf solutions for expressing metadata specifically oriented towards interoperability. New technologies addressing interoperability can also be developed using XML metadata technologies as a starting base.

What are the best opportunities for using XML metadata technologies?

RDF (Resource Description Framework) presents a good opportunity for adding new metadata to XML based systems. The most likely near term uses of RDF and other XML metadata technologies will probably be for Internet search and retrieval. However, competitive pressure from search engines such as Google (http://www.google.com) that rely on fully automated metadata extraction to improve searches could slow the use of RDF, since RDF requires explicit manual entry of the metadata by document creators. This is less of an issue for interoperability, however, since there are relatively few options for automated extraction of interoperability metadata from existing data and source codes. RDF thus may provide a good tool for research-oriented efforts in capturing and applying interoperability metadata. Other XML tools that support metadata, such as XML Schema and (possibly) object-oriented XMI, may also prove useful in developing new ways to capture and use interoperability metadata.

References

http://www.xml.com/xml/pub/98/06/rdf.html,
“RDF and Metadata…What is RDF? Why would you want to use it?”

http://www.w3.org/TR/REC-rdf-syntax/,
“Resource Description (RDF) Model and Syntax Specification,” 22 Feb 1999

http://www.w3.org/TR/2000/WD-xmlschema-0-20000407/primer.html,
XML Schema Part 0: Primer

http://www.w3.org/TR/xmlschema-1/,
XML Schema Part 1: Structures

http://www.w3.org/TR/xmlschema-2/,
XML Schema Part 2: Datatypes

http://www.ucc.ie/xml/#SCHEMATA,
XML FAQ answer to question on differences between DTDs and XML Schema

http://www.omg.org/news/pr99/xmi_overview.html,
“An Overview to the XMI”

4.4.6 SOAP (Simple Object Access Protocol) and DNA 2000

What are SOAP and DNA 2000?

SOAP is a very recent (late 1999) XML-based middleware from Microsoft. DNA 2000 is the latest incarnation of Microsoft's "Distributed Network Architecture" marketing concept, this time based on SOAP rather than on COM+ (which was never well defined). While DNA 2000 is largely just a marketing label, the SOAP protocol is well defined and supported by the Internet Engineering Task Force (IETF), as well as by Microsoft. Perhaps the single most important distinction between SOAP and earlier Microsoft middleware efforts such as DCOM is that SOAP is truly portable to non-Windows platforms, and is already extensively supported by such non-Microsoft-focused communities such as Perl programmers. This portability and the use of XML in SOAP make it a more interesting and potentially powerful technology than the earlier and technically much weaker offerings of DOM and COM+ that Microsoft previously promoted for this same general market area. Indications are strong that Microsoft will pursue XML and SOAP vigorously over the next few years, and will likely de-emphasize or even begin to phase out its earlier DCOM based efforts at middleware support.

How is SOAP relevant to interoperability?

SOAP is likely to be readily available on all Windows systems, and should be readily available on nearly any other type of actively supported computing platform. One of the reasons for this portability is that SOAP is based on XML, which uses text strings that are readily parsed by highly portable programming languages such as Perl. As a direct result, SOAP can be ported with relative ease even when a platform does not currently support it directly. SOAP thus is likely to be a significant component of future interoperability in distributed networks.

What are the best opportunities for using SOAP?

SOAP has unusually good prospects for becoming a globally important method for supporting communicating between distributed software components, and should be tracked closely. SOAP is also well worth using in early prototyping and interoperability exploration work, since its XML base makes it an interesting tool with good potential for building up new interoperability between systems fairly quickly.

References

http://www.develop.com/soap/,
Simple Object Access Protocol (SOAP)

http://www.develop.com/soap/soapfaq.htm,
SOAP Frequently Asked Questions

http://msdn.microsoft.com/workshop/xml/general/soapspec.asp,
SOAP 1.1 Specification

http://msdn.microsoft.com/xml/general/soap_faq.asp,
Microsoft SOAP FAQ

http://msdn.microsoft.com/xml/general/soap_white_paper.asp,
SOAP and Firewalls

http://www.devx.com/upload/free/features/entdev/1999/11nov99/cv1199/cv1199.asp,
"DNA 2000: Opening New Windows"

4.4.7 WAP (Wireless Application Protocol)

What is WAP?

WAP (Wireless Application Protocol) is best understood as a customization of Internet protocols to the needs of mobile devices. WAP takes Internet protocols that were designed for fixed, high-reliability, land-based networks with high data rates, and adapts them for use in mobile networked environments. Compared to the land-based networks, such mobile environments have limited bandwidth, uncertain and unreliable access, higher latency (delay), modest client-side processing capacities, and limited client-side battery life. These are all issues that are addressed in one way or another by WAP standards.

WAP standards development began in earnest in early 1998, and reached an important threshold of commercial maturity in late 1999 with release of the WAP 1.2 specification. As of mid 2000, roughly 90% of mobile device manufacturers had committed to using WAP 1.2 in their next generation of mobile devices. This unusually high level of support for a new standard indicates good prospects for WAP to become the basis for the mobile Internet. Additionally, the WAP standards group, WAP Forum, has chosen a well-formulated, highly cooperative approach that should help stabilize and promote WAP. Rather than defining new standards that compete with existing standards from other groups, the WAP Forum focuses on how to support use of such standards (e.g., XML) in mobile systems. WAP specifications thus focus on how to adapt such standards to the unique needs of mobile devices, with standards from other groups referenced freely wherever appropriate.

How is WAP relevant to interoperability?

WAP is especially relevant to naval needs because it focuses on essentially the same problems that traditionally set naval and DoD communication environments apart from commercial ones: limited bandwidth, uncertain and unpredictable access, higher latency (delay), limited client-side processing capacities, and limited client-side battery life. These problems are typical of deployed military forces and tactical environments at sea and on land, so a technology that makes the proven interoperability of the Internet more accessible in such environments also has substantial potential for benefiting naval interoperability.

What are the best opportunities for using WAP?

The impending release over the next year or two (2001-2002) of the first generation of commercial WAP-capable mobile devices translates into a unique opportunity for adapting such devices to naval needs. WAP has already addressed many of the same problems found in tactical military environments, so WAP devices presents a significant opportunity to leap ahead in resolving such issues. The two areas that will require the most attention for effective naval use of WAP are: applying WAP devices to specific DoD needs, and ensuring that high levels of security can be supported.

References

http://www.wapforum.org/what/index.htm, What are WAP and WAP Forum?

http://www.wapforum.org/what/WAP_white_pages.pdf, WAP White Paper

http://www.wapforum.org/what/technical.htm, WAP Forum Specifications

4.5 Portable Operating Systems

What are portable operating systems?

A portable operating system is one that provides the same basic services across a range of hardware platforms. It should be noted that having a portable operating system does not immediately guarantee that applications will be similarly portable, since applications that have been compiled to work in the binary language of a computer will not run on a different platform.

As used here, portable operating systems also includes cases where "adapter layers" have been added to other operating systems to make them appear more compatible.

How are portable operating systems relevant to interoperability?

As with portable languages, portable operating systems make interoperability easier by making it easier to run the same applications on different systems. From an operational perspective, they can also make the use of a system more consistent and thus easier.

What are the best opportunities for using portable operating systems?

Portable operating systems probably provide the greatest interoperability benefits in the area of training and support, and (like portable languages) they can reduce the overall cost and difficulty of creating new systems and upgrading older ones.

References

– None –

4.5.1 COE – Common Operating Environment

What is COE?

The Defense Information Infrastructure Common Operating Environment (DII COE, or simply COE) is an example of an "adapter layer" applied to two Unix and one Windows (NT) operating systems to make them appear more uniform. It also includes custom software that manages startup, access, and installation of new applications, with the objective of providing the same kind of easy installation and de-installation of applications found on Microsoft Windows systems. The COE also includes a large suite of approved commercial software for use with COE systems.

How is COE relevant to interoperability?

The goal of the COE is interoperability between diverse types of operating systems. At present that goal is limited by the fact that the COE only supports three operating systems: two Unix systems, plus Windows NT. Additionally, the COE only applies to relatively recent computers and operating systems; it cannot be used or older and smaller systems, since it requires significant graphics processing.

What are the best opportunities for using COE?

The COE effort is ongoing and should be carefully tracked by any naval interoperability effort. To see if it applies to a specific situation and computer environment, the referenced checklists can be used.

References

http://spider.dii.osfl.disa.mil:80/dii/aog_twg/twg/kern4_5.ppt,
"Overview of RT COE issues"

http://spider.dii.osfl.disa.mil:80/dii/aog_twg/twg/DECISION.DOC,
"Decision tree for using the RT COE"

http://spider.dii.osfl.disa.mil:80/dii/kpc/KPCP_doc/KPCP_doc.htm,
"DII COE Kernel Platform Certification (KPC) Program"

4.5.2 Linux

What is Linux?

Linux is a free, full-featured, open-source, Unix-like operating system. Although it was originally designed for PCs, it currently runs on at least as wide a range of platforms as any other operating system currently available. The fact that Linux is open-source means both that all of the source code for it is available, and that it can be obtained without charge.

Although commercial vendors generally avoided Linux in the past because of its free, open-source status, that situation has change dramatically over the past year. Because Linux has increasingly come to be viewed as a non-proprietary alternative to Windows NT, many large vendors such as IBM and Oracle, as well as many smaller vendors, have announced that they will port products for Linux.

How is Linux relevant to interoperability?

Linux is a potential asset for interoperability both because of its availability on a wide range of platforms, and because the availability of its source code means it can be adapted to specialized uses or even custom machines.

A recent graphical interface to Linux, called GNOME, also represents an intriguing possibility for greater interoperability at the graphical user interface level. GNOME is unusually flexible, and may be capable of emulating a wide range of other interfaces without undue effort.

What are the best opportunities for using Linux?

Linux and Linux applications should be tracked carefully over the next few years. Where its use does not contradict applicable standards, Linux could be used to provide a more uniform environment that can more readily include older, smaller systems, since in general Linux runs more efficiently on such systems than other (e.g., NT) operating systems.

References

http://www.linux.org/, "Linux Online."

Section 5

Summary: Naval Interoperability Opportunities

5.1 The Role of Technology in Interoperability

While military interoperability is often viewed as being more a matter of policy than technology, the vital role of emerging technologies in supporting those policies must not be overlooked. Emerging information technologies not only make communication of data between allies easier, but also allow such communications to take place at much faster speeds than is possible using only people-based policy decisions. For example, fast-paced tactical environments require data exchanges and re-alignments of resources at speeds that will be feasible only if higher levels of automated interoperability become widely available.

5.2 Specific Opportunities

While this document is only intended to summarize and clarify some of the issues surrounding emerging interoperability technologies, the topics covered in this report also suggest some possible specific strategies for promoting interoperability technologies. The following paragraphs outline one such approach.

5.2.1 Analyze Needs In Terms of Assets and Interoperability Domains

In this step the objective is to understand better what is being shared and how it needs to be shared to be effective in supporting multi-force and allied missions. Activities include:

- Defining current levels of intentional and “accidental” sharing of resources.
- Determine logistically reasonable interoperability domains in which assets that need to be used together are part of the same composite interoperability domains. The goal here is to prevent useless distribution of parts or data to locations that do not have a sufficient overall set of assets to make use of them. Such “frayed” distributions are wasteful, operationally confusing, and can be significant security risks (e.g., sending out RF messages to allied forces that are incapable of receiving them).
- Define clear, sharp boundaries for each resulting composite interoperability domain. These boundaries should be defined for both space (where assets may go) and time (how long assets should be available to, say, an allied force), and their structure will delineate the overall security requirements for building interoperability firewalls.
- For each type of asset, determine internal security needs for detecting, slowing, halting, and removing threats that penetrate the outer (“firewall”) protections.

5.2.2 Search for Applicable Technologies

In this step, the requirements defined by analysis of assets and interoperability domains are mapped into known and emerging technologies that could support those needs. Fallback positions may need to be defined when there are no good candidates for supporting a specific interoperability requirement. For information technologies, general technology categories to investigate include:

- *Standards.* Good international standards should be the first starting point for any kind of interoperability exploration, since they offer the most efficient and economical way of getting diverse systems to interoperate.
- *Networking.* Good networking technologies usually have to deal with various aspects of interoperability because of their need to interconnect resources with diverse origins, and so are some of the first technologies that should be investigated to look for improved ways to interoperate. The impressive commercial success of the Internet and resulting commercial investment makes Internet technologies and systems particularly worth looking at as possible off-the-shelf interoperability solutions.
- *Security.* As pointed out by the unavoidable "firewall" requirements of well-defined interoperability domains, security technologies are a vital part of workable solutions to interoperability.
- *Adaptability.* Due largely to the Internet, there has been a significant growth in the use of technologies that deal with adaptability and translation, as well as greater recognition of the need for such products. These technologies range from some of the high-portability languages discussed in this report to low-cost, reasonably effective translators for human languages. Interesting advances are likely in this area.

5.2.3 Prototype and Evolve the Use of Applicable Technologies

In order to help promote new technologies that deal specifically with special naval and military needs, prototyping and helping to evolve new standards and technologies is an important part of the overall approach to building better interoperability. In particular, naval groups can investigate promising technologies from unique perspectives, such as their use in high security environments and rapidly changing tactical environments.

5.2.4 Promote Standardization of Applicable Technologies

Based both on special naval and military needs and on the result of experimental use of new technologies, active promotion of new (and especially international) standards will be an important long-term component of any overall strategy to promote interoperability. Viewing these activities in terms of the basic and composite standardization strategies described in Section 3 of this report can help understand and in some cases predict whether a particular standardization effort is likely to produce results that will be truly useful for naval needs. Also, when new technologies that look promising are identified, the recommendations and warning signs of Section 3 can be used to help build new standardization efforts that are more likely to succeed.

5.3 Technologies of Special Interest

Although predicting “winners” in emerging technologies is always fraught with risk, this section summarizes several technologies from Section 4 that seem particularly likely to have both significant commercial support and strong impacts in interoperability over the next few years. These technologies are strong enough to merit inclusion in one or more new or ongoing naval research and development programs that are specifically devoted to tracking, testing, and prototyping the use of interoperability technologies for naval operations.

5.3.1 SOAP (Simple Object Access Protocol) and Other XML Technologies

The new SOAP protocol is probably the single most interesting new interoperability technology covered in this document. It is highly portable and supported both by a broad community of developers (the Internet Engineering Task Force, or IETF) and Microsoft, the leading vendor of PC operating systems. It is also sufficiently well defined for immediate use in exploratory naval applications and prototyping work. Finally, SOAP is associated with a suite of related XML-based technologies that are rapidly growing in use and worth tracking and investigating in their own right. One risk that should be noted with SOAP results from Microsoft’s investment in it. Microsoft could decide later to abandon cooperation with IETF and create a proprietary SOAP standard.

5.3.2 WAP (Wireless Application Protocol)

WAP is another very interesting and promising interoperability technology, since it stands a good chance of bringing the benefits of the Internet to mobile environments whose needs are strikingly similar to those of tactical military environments. Both environments share problems of limited bandwidth, uncertain access, unreliable connections, high latency, modest client-side processing capacities, and limited client-side battery life, and all these issues are issues that are addressed to one degree or another in WAP standards. WAP devices thus could prove highly cost-effective for solving many tactical interoperability needs, and naval support for WAP security could benefit both the defense and commercial sectors. The impact of WAP on software will be more indirect; it should encourage the growth and spread of more small, portable applications based on technologies such as XML and Java.

5.3.3 XML Metadata Technologies

XML technologies that specifically support metadata are of special interest due to their good fit to the fundamental problem of how to capture and use interoperability metadata. The RDF (Resource Description Framework) is perhaps the best off-the-shelf candidate for capturing interoperability metadata. However, RDF was originally conceived as a way to capture document retrieval metadata, not interoperability metadata. Using RDF to capture interoperability metadata thus would require refocusing it to the interoperability problem.

XML Schema is another XML metadata technology that could prove useful because of its ability to specify new XML data formats dynamically. XML metadata technologies could prove especially useful if combined with SOAP to help support automated distribution of interoperability-related data types to systems in an allied or coalition network.

5.3.4 Java and associated technologies

Although it is far from an new technology, Java and more recent associated technologies such as Jini and Enterprise Java Beans are well worth pursuing because they currently have the best overall potential as "universal" Internet application languages and system interfaces. The greatest problem with this suite of technologies is that they are proprietary to Sun Microsystems. Java and its associated technologies have also generally failed to reach the level of market saturation for which Sun had originally hoped, and for that reason are still better described as emerging technologies than as fully deployed ones.

5.3.5 Linux and open systems

Open source systems such as Linux are of special interest because they are far more adaptable than most comparable commercial products, and thus can be applied to special defense needs more readily. A good example of this is ongoing work by the National Security Agency (NSA) to produce a secure version of Linux. Open source methods are also interesting as ways of stabilizing important standards against capture of open standards by private corporations.

Linux in is likely to become more important as a direct consequence of the planned breakup of Microsoft into two corporations that will focus in one case on the operating system (Windows) and in the other on applications (Office and other Microsoft products). Due to the greatly increased corporate support Linux has received over the last couple of years, it will almost certainly be the single most important direct competitor of any such Windows-only company. Its low cost and ready availability could make it an especially difficult competitor for Windows.

Glossary of Acronyms

A

ASCII	American Standard Code for Information Interchange

B

BYOX	Bring Your Own X

C

COBOL	COmmon Business Oriented Language
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf

D

DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DII COE	Defense Information Infrastructure Common Operating Environment (often pronounced “die-co”)
DNA 2000	Distributed Network Architecture 2000

E

EBCDIC	Extended Binary Coded Decimal Interchange Code
EJB	Enterprise Java Beans

F

FORTRAN	FORmula TRANslation

G

GNU	GNU’s Not Unix [a recursive acronym]
GUI	Graphical User Interface

H

HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol

I

I2O	Intelligent I/O (Input/Output)
IETF	Internet Engineering Task Force
IIOP	Internet Inter-ORB Protocol
IP	Internet Protocol

J

Java	[Not an acronym; a marketing name selected and trademarked by Sun]
Jini	[Not an acronym; a marketing name selected and trademarked by Sun]

L

Linux	[Not an acronym; derived from first name of its creator, Linus Torvalds]

M

MPEG	Moving Picture Experts Group

N

NT	[Windows] New Technology [operating system]

O

ORB	Object Request Broker
OMG	Object Management Group

P

PL/I	Programming Language One [IBM had high hopes]
POSIX	Portable Operating System Interface [BYOX]

R

RDF	Resource Description Framework
RMI	Remote Method Invocation

S

SGML	Standard Generalized Markup Language [the parent of HTML]
SOAP	Simple Object Access Protocol

T

TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol / Internet Protocol

U

Unix	[Not an acronym; a play on the name of the Multics operating system]

W

WAP	Wireless Application Protocol

X

XML	eXtensible Markup Language
XUL	XML-based User Interface Language

Index