

Microsoft



Interoperability 101

A beginner's guide to understand interoperability.

Version ID: 2.2

Last Updated: February 2008

Author: Lorenzo Madrid

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

© 2006 Microsoft Corporation. All rights reserved.

Microsoft, Active Directory, BizTalk, Win32, Windows, Windows Server, and Windows Server System are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are property of their respective owners.

Table of Contents

Introduction	4
Portability	5
Cross-communication and interoperability	8
Interoperability – A simple software application case	11
Interoperability and gateways – Expanding the possibilities	12
Interoperability and convergence	15
SOA and WEB services	16
Open standard is not open source	18
Standards and interoperability – A great marriage	19
Conclusion	21
Glossary	23
Bibliography	24

Introduction

The objective of this paper is to provide an initial overview of what “interoperability” is, define some basic concepts, and present how interoperability has evolved over time since the initial stages of computing. We will also present some best-practice cases in which interoperability was a key element for leveraging successful e-government initiatives. Some basic recommendations will also be presented as a result of observing these cases.

Another objective of this paper is to clarify the difference between the meaning of portability and interoperability as seen by Microsoft, thereby serving as a reference to provide a clear understanding of these two concepts.

For an advanced understanding of interoperability, we recommend “Government Interoperability – Enabling the Delivery of E-Services,” a Microsoft white paper by Jerry Fishenden, Oliver Bell and Alan Grose.

In computer science, as in any other science, new terminology is continuously created to name specific technical issues. Later on, these words are used with different meanings by a larger population, without a clear or complete understanding of their initial purpose. Sometimes, a term starts to become used incorrectly, which can lead the decision-making process to happen under misinterpretation. In turn, this can have a severe impact on the future operational processes of information technology (IT) departments, which can then cause problems for the rest of the organization.

Nowadays, IT platforms tend to concentrate on a few popular vendors, for example, IBM mainframe operating system, UNIX and Linux flavors, and Microsoft Windows. Apple OS X plays a small part in the overall market. Many important systems of the past, such as Burroughs MCP, Control Data NOS & SCOPE, Digital VAX/VMS, or PDP 11 operating system, no longer play a relevant role either. In the past, due to the multitude of hardware and software platforms, there were two different needs that had to be solved in order to minimize the problems that arose from such a disorganized market (a situation known as the “IT Babel Tower”). They were:

- a) To have the same application program running on different platforms.
- b) To have one platform talk to another, exchanging data and messages between them.

In that regard, new terminology was created to address these issues:

- a) Program portability, or simply portability
- b) Cross-communication/interoperability

This paper explores both of these concepts in detail using specific technical and real-world examples.

Portability

In the early days of computing, programs and systems were developed for a specific platform, using programming languages that were not fully compatible. Each hardware vendor had their own unique assembly language, as well as implementation of FORTRAN and COBOL, then the predominant “standard” programming languages. The tools of each vendor had several “unique” features. The resulting programs could not be easily compiled on different platforms without major changes in its code. To add to the problem, there were also huge differences in the operating systems, and most of the systems were built to run only on one specific operating system. Most programmers often used internal and low-level commands to talk to the operating system functions, in addition to using file formats that existed only within one specific operating system, which made the whole process even more difficult.

Moving an application from one platform to another was a nightmare, even when based on a similar operating system, such as IBM's VSE to MVS, PDP 11 to VAX, or across the different flavors of UNIX. The task of making one program run on a different platform was called porting, and until today it is commonly used in UNIX environments.

Throughout the years, the adoption and acceptance of standards in the programming language arena, such as C, C++, and C#, among others, as well as development frameworks, such as J2EE and .NET, have significantly reduced the need for porting applications from one platform to another; however, this is still needed when moving an application across different operating systems, such as moving from UNIX to MVS or from UNIX to Windows.

The initial concept of portability was defined as the need to run the same application on a different piece of hardware from where it was originally defined. Therefore, portability was basically related to hardware constraints. The operating system was then tightly coupled to the hardware.

Let's look at a few of definitions of portability:

Microsoft Encarta® define portability as:

Computing easily converted to run on different computer operating systems.

<http://whatis.techtarget.com/> defines it as:

Portability is a characteristic attributed to a computer program if it can be used in operating systems other than the one in which it was created without requiring major rework. Porting is the task of doing any work necessary to make the computer program run in the new environment. In general, programs that adhere to standard program interfaces, such as the X/Open Unix 95 standard C language interface, are portable. Ideally, such a program needs only to be compiled for the operating system to which it is being ported. However, programmers using standard interfaces also sometimes use operating system extensions or special

capabilities that may not be present in the new operating system. Uses of such extensions have to be removed or replaced with comparable functions in the new operating system. In addition to language differences, porting may also require data conversion and adaptation to new system procedures for running an application.

<http://www.techweb.com/> defines it as:

Portability refers to software that can be easily moved from one type of machine to another. It implies a product that has a version for several hardware platforms or that has built-in capabilities for switching between them. However, a program that can be easily converted from one machine type to another is also considered portable.

Wikipedia (www.wikipedia.com) defines portability as:

[P]orting is the adaptation of a piece of software so that it will function in a different computing environment than that for which it was originally written.

Porting is usually required because of differences in the central processing unit, operating system interfaces, different hardware, or because of subtle incompatibilities in—or even complete absence of—the programming language used on the target environment.

Portability generally refers to one of two things. The first is a reference to the ability to compile code once (usually into some form of intermediary code that is then JIT-compiled at run-time) and run it on multiple platforms without any modification to the code. The second is a reference to the property of software that describes how easy it is to port. As operating systems, languages, and programming techniques evolve, software becomes increasingly simple to port between environments. One of the original objectives of the C programming language and the standard C library, for instance, was to ease porting of software by providing a common API to different and otherwise incompatible computing hardware.

Generally, using higher-level function calls instead of bare [operating system]-level APIs improves portability.

International standards, such as those promulgated by ISO, greatly facilitates porting because they specify the details of the computing environment in a way that varies little among platforms. Often, porting software between two platforms that implement the same standard (such as, POSIX.1) is simply a matter of recompiling the program on the new platform.

There also exist an increasing number of tools to facilitate porting—such as GCC, which provides consistent programming languages on different platforms, and autoconf, which automates the detection of minor variations in the environment and adapts the software accordingly before compilation.

Two activities related to, but distinct from, porting are emulating and cross-compiling.

Porting is also the term used when a computer game designed to run on one platform, be it a personal computer or video game console, is converted to run on another platform. Earlier video game ports were not true ports, but rather

complete rewrites. [Today,] more and more video games are developed using editing software which can output code for PCs as well as one or more consoles. Many early ports suffered from bad quality because the hardware of PCs and consoles differed greatly.

Encyclopedia Britannica defines portability as:

Usable on many computers without modification <portable software>

Taking into consideration all of the preceding definitions in which portability is essentially defined as the ability to use the same program on different computers without modifications, we can state that the Microsoft Windows platform offers the best portability advantage over any other platform, because any Windows-based application, can run seamlessly on hundreds of different computer vendors *without modification* or the need for porting.

Therefore, the need for portability shall only exist when there is an imperative need to migrate or convert one software application from one platform to a totally different one. Portability shall not be confused or mixed with interoperability. Two different systems can interoperate, without the need of either one being portable, as we will see in the next section.

Cross-communication and interoperability

Whereas portability is about moving an *application* to, and running it seamlessly on, different platforms, cross-communication and interoperability are about enabling different applications, devices, platforms, or components to connect and exchange data between them—that is, to “talk.”

There are a variety of complementary means to achieve interoperability, including: (1) development of software that is “interoperable by design” (for example, the inclusion of a particular technology or functionality in software to facilitate the easy exchange of data across different applications, or the creation of a “translator” or “gateway” that runs on one product and implements communications with another); (2) publication and licensing or cross-licensing of proprietary technologies and essential intellectual property; (3) specific collaborations with partners, competitors, customers, and governments; and (4) implementation of industry standards (including open standards and broadly accessible proprietary standards) in products and services.

While each of these ways to achieve interoperability is important and effective in various contexts, and while this paper touches on each of them at certain points, the focus is on how industry standards and gateways have been used to facilitate interoperability in the IT marketplace.

To address the cross-communication issues, a set of standards, such as ASCII, BCD, and EBCDIC, was developed to harmonize how data was stored in computers. Data telecommunications protocols—such as Poll Select, BSC1, BSC3, and SDLC—and network protocols—such as SNA, DECNET, ISO/OSI, and TCP/IP—were developed to facilitate communication between devices.

The existence of these protocols and standards first allowed devices and printers to connect to each other. Later, computers were able to exchange data using options such as offline magnetic tapes; computer-to-computer dialog; or online, real-time (tele) communications and channels.

During the initial stages, this cross-communication process was limited to data exchange only, and every application had its own “record format” specification to make it possible to “understand” the meaning of the data elements coming in or going out. Thus, in order to exchange data between systems, every application needed to write specific code to support its counterpart record definitions.

This proved to be an extremely inefficient way to work. Due to the complexity of the systems, their different requirements, and the ever-growing number of applications, a huge programming effort was required to maintain the code every time there was the need to change a single “record format” in only one application. Thousands of lines of code needed to be changed among the different programs using that data record. The

solution to this problem was to define additional protocol levels, not only to harmonize data *elements*, but also to harmonize *data formats*.

The solution to the problem was also based on describing data record formats and storing them as separate entities from the program logic. This simple need led to the appearance of database management systems (DBMS), the first leaders of which were IMS/DB and DL/1 from IBM and TOTAL from Cincom Systems. This significant new technology (DBMS) was an important step in allowing programs and systems to exchange information once data was stored and described in a single location. Therefore, if a change was needed in a data element, only the programs and systems dealing with that data element had to be modified. Intersystem communication was easily accomplished among programs and systems using the same DBMS. In addition, TOTAL was available for many different hardware vendors, making it much easier to *port* programs and systems from one vendor to another using the DBMS as the vehicle to transport data. Data could be passed from one system to the other, within the same computer, or, alternatively, to another computer, using an offline data-exchange mechanism—usually compatible magnetic tapes—as the transport media.

At the same time, it became clear that data needed to be sent back and forth between different computers on a real-time basis—that is, to have seamless interoperability. Many existing telecommunication protocols were available, but a record-format specification was needed to address industry-specific needs to exchange application data, such as for bank transactions, stock exchange markets, telecom, and manufacturing. A few protocols, based on industry-specific needs, appeared to support what are generically called EDI—Electronic Data Interchange—needs. Other pioneering message formats that experienced huge success were ISO 8583 and SWIFT, which connected financial institutions and EFT (electronic funds transfer) network operators.

The increased need to have systems automatically interoperate, the cost reduction in telecommunications rates, and the existence of reliable and well-defined network protocols were the key catalysts that drove the large adoption of the Internet as the preferred media to provide interoperability among systems. At this point in time, it became possible to have systems not only talk to other systems through external data-exchange mechanisms, but also to communicate among themselves in real time. Instead of transporting full data files from one system to another, it was now possible to send specific data-information elements from one system to another and receive an answer. Once again, sending data elements among non-homogenous systems required the development of common protocols so that any system could understand the data. The final result was ubiquitous computing.

As we have seen, the need to solve the interoperability issue among programs and systems was fundamental to the development of protocols. Their wide acceptance made them industry standards. Protocols now allow data elements to be independent from the program logic code, thus providing a common ground that allows multiple systems to understand data that is being sent from one system to the other, *regardless* of the platform originating or receiving the data.

In short, what began as a data exchange problem evolved to a program-to-program and then to a system-to-system communication need. In the program interoperability arena, many technical solutions were proposed and adopted, from the message-based protocols (in the beginning, RPC-based) to object-based and open systems initiatives, such as the ODBC (Open Database Connection) in the late '80s, to the component-based (in the '90s), and ultimately the XML-message based ('00s).

Currently, XML is the most accepted language that provides the level of data-exchange requirements needed to allow interoperability between systems and programs. XML includes in the preface of the data streams being transmitted the descriptions of the data elements in that stream.

Interoperability – A simple software application case

To illustrate interoperability, let's look at a simple example based on needs commonly found in public services agencies around the world.

System A is responsible for keeping driver's license records and was developed many years ago, while System B is needed now to record car accidents. It soon became obvious that System B needed access to the data available in System A. The alternatives were:

1. Duplicate the files from System A on to System B.
2. Have System B query the data on System A for the information it needs.
3. Have System B to query and update the data on System A (a more complex solution).

These three scenarios could be considered interoperability alternatives; however, only the third solution can be considered a full one-way interoperability case.

How is the third solution accomplished? We could use a batch solution in which files and records are processed from time to time, or we could process them in real time. In short, we have different levels of interoperability solutions, as shown in the following:

- Duplicate files from System A on to System B
- Transaction processing in batch mode
- Transaction processing in real time (query mode)
- Transaction processing in real time (update mode)
- Transaction processing in real time (bidirectional update mode)

In addition, it is understood that interoperability here is supported by a common set of base communication facilities that allow data to be transparently sent and read from one system to the other, regardless of the operating system, network protocols, and programming languages being used by the applications.

This is accomplished today mainly through the large acceptance and adoption of the TCP/IP protocol, which provides a foundation for the network and telecommunication infrastructure. Once the basic telecommunication process is in place (TCP/IP), new layers can be deployed on top as middleware components to allow a better and easier communication among applications.

Interoperability and gateways – Expanding the possibilities

In the previous example, we assumed that the developers of System B had enough knowledge and understanding of System A, such that they were capable of writing the necessary code to directly access data on System A. However, in actuality, many systems are currently in production where the source code is not always available and/or the original development team is no longer available to help.

In the real world, this is the usual scenario. Furthermore, it's not a good software engineering practice to write applications directly dealing with data, particularly if that data is owned by another system. Therefore, the solution that addresses both problems is to have an *intermediate* layer between applications in order to provide the required secure level of interoperability. Application A talks to an intermediate system, and then the intermediate system, in turn, talks to Application B. This “middleman” application acts like a polyglot, knowing all languages and dialects of all related applications. Sometimes, it will be necessary to write application-dependent “connectors” to be able to speak to the “middleman” application.

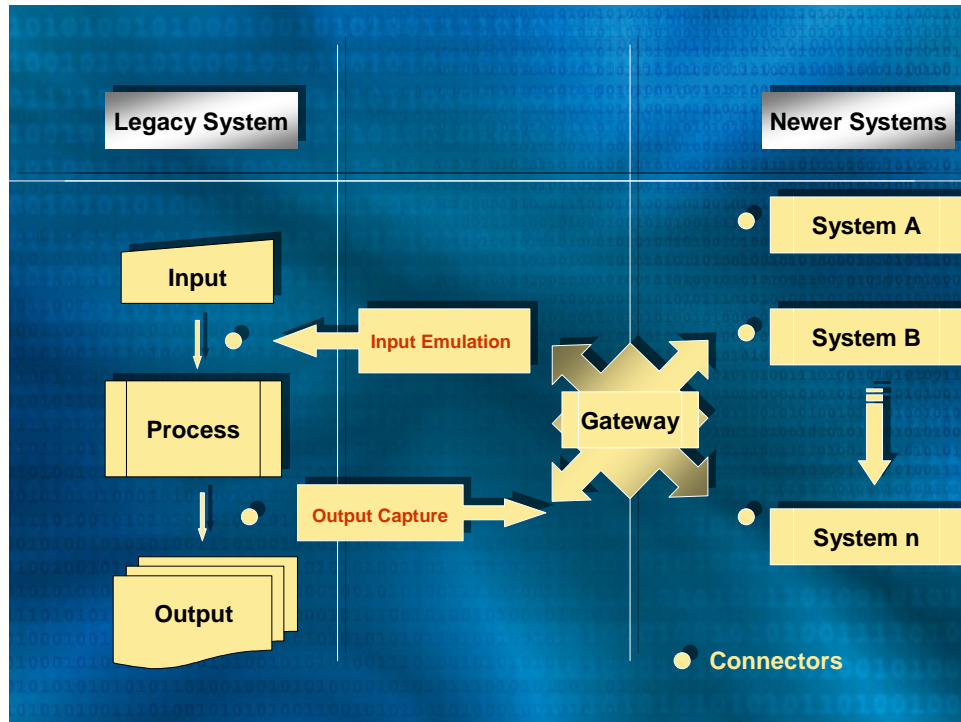
The great benefit of this concept is that we now have only one focal point where all “languages and dialects” in use are defined. If one system has changes to its internal data structures, only the “middleman” or its connector needs to know about it. Technically, this “middleman” application is called an *interoperability gateway*.

But there are additional benefits of having a gateway. Legacy systems represent a high portion of today's existing systems solutions. Many of these legacy systems were not designed to support current protocols and technologies; therefore, their interoperability capabilities are normally restricted to their original design intentions.

If there is a requirement that such a legacy system must interoperate with another system, this may lead to a decision to rewrite the legacy application or to write a parallel or alternative application or a bridge solution. However, this decision may not be necessary if we could provide a mechanism to emulate data formats for these legacy systems.

All systems are based on three basic functions: input, process, and output. If, using existing technologies, we could intercept data streams in the input/output phase; it would be possible to provide interoperability facilities to any system.

The best way to do that would be to produce the necessary input data in the format and with the input method used by the legacy systems. Going the other way, output data could be captured at the existing system's output device and then forwarded to any other system.



This method could then be extended to become a generic solution that allows interoperability among all systems. In that regard, the gateway solution applied is similar to the concept of a data translator dealer that receives data from one originating system and then forwards the data, in a translated format, to one or more destination systems. The destination system will then process the transaction requested and produce the desired results.

This is exactly the point where a *gateway* solution could be applied, because on top of being a translator of the languages and formats used by the applications, the gateway could also use emulation facilities to intercept data coming from or going out legacy systems.

A gateway and its appropriate connector can emulate an old video terminal and automatically input data into a legacy application as if it were a real person sitting in front of the device. At the same time, a gateway can look like an old printer device and capture data from the same legacy application and forward it to a newer system in the appropriate new format.

Let's take the following case:

System X is a legacy system built on IBM technology using CICS and ADABAS database management software. Its mission is to provide student registration and records for a university. One of the transactional functions using 3270 text terminals allows

querying the system using a student's ID number to find out in which classes that student is registered.

In the same university, the library has decided to develop a new system, using Microsoft .NET and SQL™ technologies, to keep track of library utilization by the students, using a Web-compatible application. One of the requirements for the new system is to know whether or not a student ID is still valid so a book can be made available to a borrower.

The new application will obtain a student number using an Internet Web form and send the data to the interoperability gateway. The gateway will convert the data from TCP/IP/XML/UNICODE standards into an SNA/LU2/EBCDIC standard and forward it to the IBM application. The IBM application will understand the data because it looks like a regular terminal 3270 query, and then the application will reply as it should. The gateway will intercept the returned data stream from the IBM application and convert it back into the appropriate format for the Web application.

Since the gateway serves as a middleman that understands and processes both sides of the transaction, both systems will fully interoperate, regardless of the fact the legacy system was not designed to do so.

However, there are additional benefits with the gateway concept. It is not uncommon in legacy systems to have duplicated data elements in different databases. This happens because applications are usually written independently, with no concern for an integration strategy. For example, let's look at public sector applications, such as ID cards, driver's licenses, or tax applications. It will most likely be the case that a person's address is stored and used by each of these apparently unrelated systems. Normally, this piece of information is not shared among the systems and there are no controls for accuracy or data quality assurance. Therefore, if a change to the address is needed, several requests from the citizen will have to be placed to correct the same information. In an interoperable architecture, the gateway could handle requests to update data in different systems, thereby reducing inconsistencies. Once a citizen requests his personal data to be updated, smart system architecture could automatically replicate that new information into other systems, providing a better service for the citizen and a more efficient use of resources. The result is an increased benefit for all parties involved.

Another benefit of a gateway is that it helps to introduce a single, comprehensive user-identification mechanism to heterogeneous systems. Due to the historical nature of legacy systems development, it's common to have each system "define" its own authentication mechanism in order to allow an individual to have access and permissions to use an application. Therefore, a user would have multiple IDs for accessing the applications he or she needs to run. By contrast, a gateway can store all IDs and provide the access rights to any system for the authorized user. The user only needs to have one ID, and the gateway will be the focal point of controlling all access to the applications. This is known as the ability to federate multiple IDs and provide a single sign-on facility.

Interoperability and convergence

Convergence was initially used to define the trend to have data processing and telecommunication as unique disciplines. Born in separate fields, these technologies are now completely married. However, as technology evolves, convergence is also happening in other areas. Processors and computers are being integrated into many types of devices and then integrated into networks. Cellular telephones can send and receive e-mail and interact online with other systems. Home appliances are being integrated. A refrigerator can now place an order to be replenished through an online store, using the Web, without any human interference. A car can automatically propose a better route to its driver using a combination of software, satellite GPS, cellular phone, and Web queries. This is convergence.

A traditional ERP (Enterprise Resource Planning) system, written to handle thousands of orders sent by customers and clients, was not originally built to receive a purchase order from a refrigerator or to report to a cellular phone number whether one specific order was shipped or not. These are necessary requirements of today's systems, and we need to have an easy and generic solution to address them.

Therefore, when we speak about interoperability, we cannot forget about convergence. Today, systems need to be able to interoperate not only with other systems, but also with many different devices and appliances.



Again, the key driver to facilitating convergence is powerful software solutions able to handle the kind of interoperability requirements that convergence solutions demand. Such capabilities are only possible using commonly accepted standards and protocols.

The Microsoft Windows family components (XP, Vista, Embedded, and Mobile), together with the .NET Framework, allow the necessary infrastructure to enable convergent, interoperable solutions.

SOA and WEB services

As we progress in the use of technology and interoperability becomes a reality, we will find that the strategy for system development may change completely. Systems today can interoperate, and the presence of networks is ubiquitous due to the adoption of a common telecommunication network protocol, TCP/IP.

A system no longer needs to be a complete and bulky piece of software, with millions of lines of code, running on a gigantic mainframe computer in one central location. Today, it is possible to have multiple systems totally integrated, with hundreds of subsystems running on computers that are located in different parts of the world.

If this is the case, why does someone need to rewrite an application component that is already successfully running somewhere on the network? Let's look at a simple case. A programmer develops an efficient application that accepts an entered postal code and returns all possible addresses in that postal code area. This functionality could be easily incorporated into any system requiring it, without the need to be hard-coded into the new application. Also, the actual program that provides that function could be running anywhere and the function offered as a service and be charged for on a per-use basis.

Systems can be built based upon a network of multiple service suppliers through the use of what is known as Web services. The overall architecture for these concepts is known as SOA: service-oriented architecture. SOA can be technically defined as "an approach to organizing information technology in which data, logic, and infrastructure resources are accessed by routing messages between network interfaces."

The basic value propositions are to provide consistent, stable interfaces in front of diverse or volatile implementations, thereby establishing context for information exchange across organizations. SOA requires an organizational commitment to build its application portfolio around a service component model.

One of the most important benefits of adopting an SOA/Web services strategy is a reduction in the complexity of system deployment, as well associated maintenance cost, once systems can be constructed in building blocks that will always communicate. Big and bulky pieces of software can now be reduced to basic modules comprising just the services they will provide to other members of the overall solution. This is not only true for new application blocks. This strategy also works well for existing applications or services implementing Web services façades.

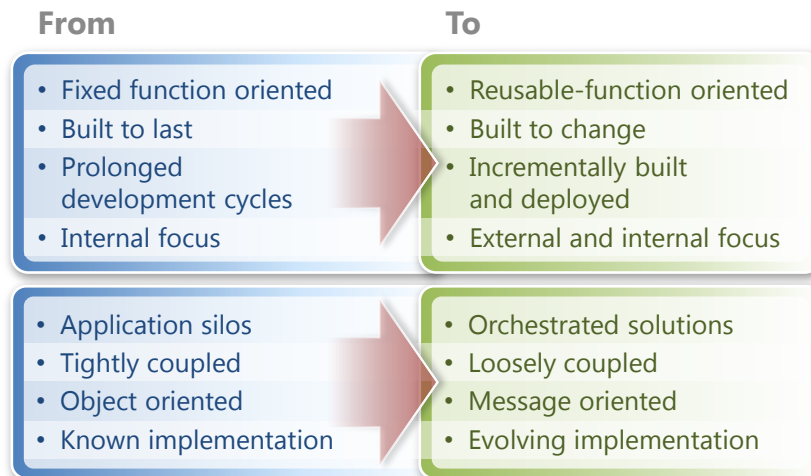
However, the challenges to introducing these kinds of technologies are significant. Web services are a great start, but we need to consider the following:

- How do I ensure reliable connections over networks that can fail?
- How do I secure my connections?
- How do I create applications that cross trust boundaries?
- Should I program to have objects or to provide services?
- Peer-to-peer and/or enterprise networks?

The change from a traditional development approach to SOA will require a new level of relationship among the parties providing the services. We can loosely compare that change to a decision to outsource services in an entity. There will be requirements to provide a minimum level of services among the components through an SLA (Service Level Agreements) for the services and infrastructure.

Although the core business applications may still be developed as usual, the overall applications architecture will face large conceptual changes. The complexity related to security, reliability, and routing will be moved to a new interface layer. Some of them are depicted in the following table:

Changing Applications Architecture Concept to SOA



Open standard is not open source

People often use “open standard” and “open source” terminology as if one was a synonym for the other. This is not the case, as many have concluded. For example, according to the document “Road Map for Open ICT Ecosystems,” produced by the Berkman Center for Internet & Society at Harvard Law School, “Open standards are not the same as open source software.” (pg. 6) <http://cyber.law.harvard.edu/epolicy> (2005).

While various parties have tried to use the term to mean different things, an open standard is a technical “specification” (that is, a set of technical instructions and requirements) that has the following characteristics:

- (1) Developed, maintained, approved, or affirmed by consensus in a voluntary, market-driven, standards-setting organization that is open to all interested and qualified participants
- (2) Published without restriction (in electronic or tangible form) in sufficient detail to enable a complete understanding of the standard’s scope and purpose (for example, potential implementers are not restricted from accessing the standard)
- (3) Publicly available without cost or for a reasonable fee for adoption and implementation by any interested party
- (4) Any patent rights necessary to implement open standards are made available by those developing the standard to all implementers on reasonable and non-discriminatory (RAND) terms (either with or without payment of a reasonable royalty or fee)

Sometimes open source software (OSS) is also erroneously equated with interoperability itself—that is, the use of OSS will ensure interoperability. *In fact, however, in certain cases, the opposite could be true.* Because all OSS source code may be modified by anyone, any OSS product that initially is standards-conformant and/or interoperable may be altered by any user in a manner that renders it non-conformant and incompatible with other users’ versions of the software. At the very least, the freedom to modify code necessarily encourages the creation of many permutations of the same type of software application, which could add implementation and testing overhead to interoperability efforts.

Microsoft implements hundreds of open standards and proprietary standards in its products to enhance their interoperability with other products and services. As a testament to Microsoft’s progress in this area, a 2003 study by Lawrence Associates/Forbes found that Windows provides a 102 percent improvement over competing open source products for standards compliance.

Standards and interoperability – A great marriage

As was mentioned previously, since the beginning of business computing in the 1960s, one of the first requirements was the need to have different hardware equipment talk to each other. Every hardware and software vendor had their own communication protocol specifications, and it was very difficult, for instance, to have Vendor A's printer attached to Vendor B's CPU.

This situation remained fairly constant through the advent of the personal computer, and even in MS/DOS it was cumbersome to attach a new printer to a PC system. Any application software needed to have its own unique driver to be able to do even a simple printout.

Windows 3.0 brought the WYSIWYG concept to the mass PC market and changed the requirement that applications must have proprietary drivers supporting particular pieces of hardware. With Windows 3.0, the driver essentially became a feature to be provided by the hardware vendor, and Windows offered a common protocol. This strategy made it easier to write applications since hardware and software vendors only needed to write one set of code using the standard interface. Hardware vendors didn't need to worry about supporting multiple proprietary application interfaces, and software vendors didn't have to support multiple proprietary hardware interfaces. Everything was taken care of through the Windows operating system.

However, it was still difficult to physically connect pieces of hardware to the PC. There were many technicalities and interrupt requests to be taken care of, memory incompatibility issues to avoid, and software drivers to be manually installed. It was not an easy task, and normally only hardware-savvy individuals were able to deal with this with some level of comfort.

With Windows 98, Microsoft made the Plug and Play (PnP) concept largely available. This set of standards for hardware and software vendors made it simple to install additional hardware on the PC. All you had to do was plug in the device and, after a few internal processing steps, the hardware was ready to go. The key was the unique code inside the device that was identifiable by the operating system, which enabled it to run the necessary programs to automatically install and configure the equipment.

But that wasn't the end of the problem. Several devices needed high-speed communication with the processor, which required internal boards to be installed on the PCs. The solution came in the form of a new industry standard specification called USB (universal serial bus), which uses a very simple external physical plug and allows high-speed communication between the CPU and the external hardware pieces.

Today, with very few exceptions, almost any equipment can connect to a PC and start working immediately, exchanging the desired information. Printers, scanners, digital cameras, sound devices, sensors, mobile telephones—almost anything can be easily connected to a PC thanks to the PnP concept and a few widely accepted industry standards, such as USB, Bluetooth, and IEEE 1394 (also known as “FireWire”).

Although this is by far the most successful interoperability case ever accomplished by any industry, it is also the least used to explain why standards are a key way to achieve interoperability across different systems, components, or platforms. Once things become simple and easy, people tend to forget how difficult they were before the problem was solved.

Conclusion

The current server-centric paradigm in the IT space is limited in its ability to unleash the incredible power of a distributed computing architecture. Legacy systems, in general, were not designed to work together, which explains why interoperability can, at times, be expensive, complex, and resource-intensive.

Possible alternatives include:

- Establish a new paradigm, a new programming model, based on an SOA using widely accepted industry standards.
- Offer a low-cost set of tools that implement this new programming model, taking into consideration existing skills, and at the same time providing productivity gains.
- Get the benefits of network gains, achieving interoperability on legacy and new applications through the use of Web services.

So far, we have seen that, while there are various complementary means for achieving interoperability (for example, through inherent software design, licensing of intellectual property, collaboration among companies, and adoption/implementation of industry standards), this paper focuses on how protocols and standards serve as key components for enabling interoperability among different systems and how the gateway is an important element of this process. Together, they allow us to abstract the complexity of individual systems and their internal languages, and permit us to focus on the core of the problem.

Although other languages are available, XML is becoming widely accepted as the “lingua franca” for data exchange and systems interoperability and, therefore, is an important industry standard. Many countries have already adopted XML as their national standard for interoperability purposes, but not many have yet understood the full potential and benefits that can be driven from the adoption of XML and gateway concepts.

SOA and Web services allow system deployment to be done at a lower cost and with simplified maintenance, and they also provide a solid foundation to increase interoperability among different and widely dispersed systems.

Governments can largely benefit from adopting an e-government/interoperability strategy supported by XML, SOA, and Web services technologies. The implications will be noticed on many dimensions, including government transparency and increased efficiency in public services—both in the internal management process and the external process, such as tax collection, health care, security and law enforcement, justice, and education. Through an efficient e-government infrastructure, a government will be able to provide better services to its citizens and constituents, as well as reduce the associated costs.

The implementation of these technologies is not expected to be easy. However, the successful deployment of SOA and Web services architecture is less of a technology problem than a management issue. Proper preparation is necessary to effectively reach this new level of interoperable systems and services. Several barriers, which come from the cultural and political aspects of every organization, need to be overcome in addition to the technical issues.

SOA is based on services contracts and message exchange. Therefore, contracts specification among services needs to be clear and message content precise to have a successful deployment and operation. Data needs to flow and be shared by an organization, but many times we see that data has different meanings across an organization. This is where most complex IT projects fail. An agreed-upon, common taxonomy needs to exist, with well-defined metadata, and governance models need to be accepted by all participant entities. These are not IT problems only, but they are a key element for the success of any IT project.

The future of these new technologies is promising. As was the case with Plug and Play and USB standards, we can foresee a close behavior in the systems software arena. In the near future, we will have the tools and technologies that will enable application development to be focused on the core of the service it needs to perform. All other requirements, such as security, data communication database services, and network access, will no longer be part of the application, but just a service—using a set of well-defined and accepted standards and SLAs—that we will “buy” for our applications. As with USB devices, we will develop an application and just plug it into the network and it will communicate smoothly with the other applications in the environment.

Microsoft is committed to providing full support to these new technologies. Web services and XML are at the core of all Microsoft products, and many other relevant IT standards are also supported. For example, on the gateway side, Microsoft offers BizTalk® Server, a generic solution for interoperability gateways. This combination of products and support aimed at widely accepted industry protocols positions Microsoft as the leading company to provide interoperable solutions.

Glossary

API	Application Programming Interface – The generic name used to describe the communication protocol to be followed between two application programs
ASCII	American Standard Code for Information Interchange
CICS	Customer Information Control System – IBM system for online transaction processing in mainframes
DBMS	Database Management System
EBCDIC	Extended Binary Coded Decimal Interchange Code
ERP	Enterprise Resource Planning
HTTP	Hypertext Transfer Protocol
LU.2	Logical Unit Type 2 – One of the SNA specifications for 3270 terminal communication
MVS	Multiple Virtual Space – An IBM mainframe operating system
ODBC	Open Database Connection
SDLC	Synchronous Data Link Channel – One of the SNA components that allows the transport of information across the network
SNA	System Network Architecture – The IBM standard for network architecture
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol – It provides the syntax for accessing services
TCP/IP	Telecommunication Control Protocol/Internet Protocol
UDDI	Universal Description, Discovery and Integration
UNICODE	Universal Code
USB	Universal Serial Bus
Web	Refers to the World Wide Web
WSDL	Web Services Description Language – It effectively provides the contract for Web services, setting out what inputs are expected and what outputs will be supplied
XML	eXtensible Markup Language – It provides vendor-independent data interoperability between systems

Bibliography

1. SNA, IBM's Networking Solution; James Martin, 1987.
2. *DNS and BIND*; Paul Albitz & Cricket Liu, 1997.
3. ABABAS Reference Manual; Software AG.
4. E-government – O governo eletrônico no Brasil; Florência Ferrer & Paula Santos, 2004.
5. América Latina Puntogob; Rodrigo Araya D. etc... 2004.
6. "Government Interoperability – Enabling the Delivery of E-Services"; Jerry Fishenden, Oliver Bell, Alan Grose. Microsoft white paper, April 2005.
7. "Road Map for Open ICT Ecosystems"; Berkman Center for Internet & Society at Harvard Law School, September 2005
8. BSA Statement on Technology Standards – February 2005.
9. Interoperability: Definition, How to Achieve, Choice as Best Policy, Microsoft's Commitment; Microsoft Corporation, December 2005.
10. Windows or Linux – Evaluate the total cost before you decide; Lawrence Associates ([http://www.lawrence-associates.com/Downloads/Public/Windows%20or%20Linux%20TCO_wp%20\(0903\).pdf](http://www.lawrence-associates.com/Downloads/Public/Windows%20or%20Linux%20TCO_wp%20(0903).pdf))
11. 2005 World Development Indicators Database; World Bank, April 16, 2005.
12. The Global Information Technology Report, 2004-2005; Soumitra, Dutta and Augusto Lopez-Claros, World Economic Forum Reports – INSEAD.
13. Three Unexpected Results in Open-Source Software Engineering; Stephen R. Schach, Vanderbilt University, Nashville, TN, United States.
14. WITSA – Digital Planet: The Global Information Economy (<http://www.witsa.org/news/99mar.htm>).
15. Gartner – 2005 Web Services Platform Magic Quadrant Report.