Learn the discipline, pursue the art, and contribute ideas at www.ArchitectureJournal.net

Resources you can build on.

THE ARCHITECTURE JOURNAL™

Input for Better Outcomes

Journal 11

Infrastructure Architecture

Microsoft®

Contents

THE ARCHITECTURE JOURNAL™
Input for Better Outcomes
Journal 11

Foreword

Dear Architect,

Welcome to Journal 11, the theme of which is "Infrastructure Architecture". A few years ago, one of my former colleagues, Pat Helland came up with an architectural analogy called Metropolis. His vision was to create a parallel using city planning and building architecture to give meaning to complex challenges in software architecture. One of the things that I always remembered from Metropolis is that as IT architects, we often spend proportionally too much time on our "own buildings". Many of us are guilty of obsessing over what a "building" will look like, and how it will perform. As a result, however, we sometimes forget the bigger picture. For example, how our building will fit into the larger city. For building architecture, a building must conform to city plans in a number of areas including road plans, utilities, transportation, etc. Taking this forward for this issue of the journal, I wanted to explore the same challenges as they relate to our industry.

To lead this issue, we have a group of authors—Shaun Hirschman, Matthew Baldwin, Tito Leverette, and Michael Johnson, who have written an article on Mass Hosting High Availability Architectures. In this article, the group shares key aspects for creating a hosting architecture that is both resilient and scalable. Following this we have Marc Holmes and Simon Cox with a new view of HPC they call High Productivity Computing. Marc and Simon have been working together on exploring the end-to-end and infrastructure considerations for HPC. Mario Cardinal follows with an article called Test Driven Infrastructures. Mario is a big proponent of Test Driven Development (TDD) and in this article outlines how TDD can be applied at an infrastructure level.

To continue our Architecture Journal Profile series for this issue, I had the unique opportunity of meeting with Don Ferguson to ask him about what it means to be an architect. For those that don't know, Don used to be an IBM Fellow, and the Chief Architect in the Software Group at IBM, before joining Microsoft recently. Following Don's interview is an article from Jim Wilt on Conquering the Integration Dilemma. Jim shares some of his integration challenges, including introducing us to a mapping pit of despair!

To round out this issue, we have two great articles on the subject of real world Service Oriented Architecture (SOA). Shy Cohen leads with an ontology and taxonomy of services that can reside in a SOA, many of which map well into the infrastructure space. Our last article for this issue is Versioning in SOA from Boris Lublinsky. Boris takes a challenge that many architects face today and looks at multiple approaches for service versioning.

Well, that is everything wrapped up for another issue. I hope the articles within help you not only think about the type of building you are putting together in your metropolis, but also the infrastructure that's needed to help people get there!

Simon Guest

Mass Hosting High Availability Architectures

by Shaun Hirschman, Mathew Baldwin, Tito Leverette, and Michael Johnson

Summary

Scalability and high availability are essential but competing properties for hosting infrastructures. Whether you are an open-source engineer, commercial solution consumer, or Microsoft IT engineer, no "silver bullet" solution appears to exist.

In exploring different aspects of a hosting infrastructure, we may find existing technologies that we can bring together to create the "silver bullet" solution. This process will also help expose the gaps that we need to bridge. The "silver bullet" platform needs to deliver an infrastructure that can allow for density in number of customer accounts as well as being scalable and highly redundant.

This article will concentrate on the required components needed to build an ultimate environment that is scalable, reliable, secure, and easy to maintain—all while providing high availability (HA). Solitary application providers, all the way to high density shared hosters, would benefit from such a solution. But does it exist? If not, what are the challenges that block us today and how close can we get to it?

History of the Hosting Industry and Current Problems

To fully understand the complexities of a Web hosting platform, we must first understand how the hosting market started and how it evolved into its current state. Before traditional hosting began to take shape, simple static Web sites were popular and relatively new to the public at large. The infrastructure built to support this movement was as equally basic and focused more on bringing as many customers as possible versus providing higher-end services, such as interactive applications and high availability.

Let's start by describing the classic architecture that Microsoft-based hosters have employed in the past. A single, stand-alone Web server with FTP services, running IIS with content hosted locally on the server. Each stand-alone system has a certain cost to it—hardware, software licenses, network, power, rack space, and others. Some hosters have even taken this to the extreme by hosting all services on a single server for *x* number of customers. For example, they have servers with IIS, SQL, third party mail server software, and local storage for hosted content. These boxes are easy to image and quick to deploy, especially if you are selling budget packages to customers who simply want to host a few pages and nothing more complex.

As this type of platform grew, numerous issues began to rear their ugly heads: backup and restore requirements, consumption of costly data center space, power per server, high-load customers, and general management. In addition to platform issues, there was the increasing demand from consumers and the advances in new Web technology. Technologies such as PHP, Cold Fusion, ASP, JSP, and ASP.NET emerged, providing more complex platforms that were driven by the consumer's desire for richer functionality and the desire for better business. This, in turn, seeded new applications requiring data stores like SQL and other related technologies. As new applications were created, the business potential around these applications became more evident and in demand.

Hosters, in an attempt to maximize the bottom line and simplify management, continued to operate all services required to support their customers on a stand-alone server. This created an increased load on a single server, encumbering it and reducing the number of Web sites that it could serve. Consumer demand rose more rapidly than hardware technology could support. Hosters now had to scale outwardly, by separating and isolating services across multiple servers instead of scaling vertically with faster hardware.

> **"THE PRIMARY GOAL WHEN DESIGNING A HOSTED PLATFORM IS TO OPTIMIZE SCALABILITY, AVAILABILITY, DENSITY, AND PRICE PARITY WHILE ADHERING TO AS GRANULAR A LEVEL OF SECURITY AS POSSIBLE WITH THE ISOLATION OF CUSTOMERS FROM EACH OTHER."**

The hosting industry continued to get saturated with competitive companies in response to the high demand for Web-based services such as dynamic Web sites and e-commerce shopping carts. This blooming market forced hosters to differentiate their services from others by creating service level agreements (SLAs). Not only are hosters required to provide services for low cost, but they must always be available. This is substantiated by consumers and businesses' growing dependency upon the Web and their demands for more interactive, complex applications. Producing highly available services usually translates into more servers to support redundancy as well as new performance, security, and management requirements. How can hosters scale and support these

services with the given software and hardware architecture?

In light of these industry changes, software technology companies providing the foundation for these services realized that their current operating systems were not capable of meeting the needs of hosters. A high level of system administrator contact was still required to keep operations running as smoothly as possible. Independent service vendors and industry engineers became conscious of the gaps in service related software/hardware and initiated their own focus on developing technologies to fill the gaps while profiting from them.

It is at this point that hosters started to look at building platforms that are more scalable and can achieve a higher per-box density. They must also be easier to manage. A good first step in building this type of architecture is to look at the various services that the hoster needs to support and manage.

Considerations when Planning Mass Hosting High Availability Architectures

When the hoster sits down and begins to think about the technologies they can put together and how to build a platform to support many Web sites and services, a number of key requirements come to mind. These requirements range from user or applications density to the type of features and services that should be supported and their impact on the underlying system—ASP.NET, PHP, SQL and more—and finally the cost per hosted site or application. The primary goal when designing a hosted platform is to optimize scalability, availability, density, and price parity while adhering to as granular a level of security as possible with the isolation of customers from each other.

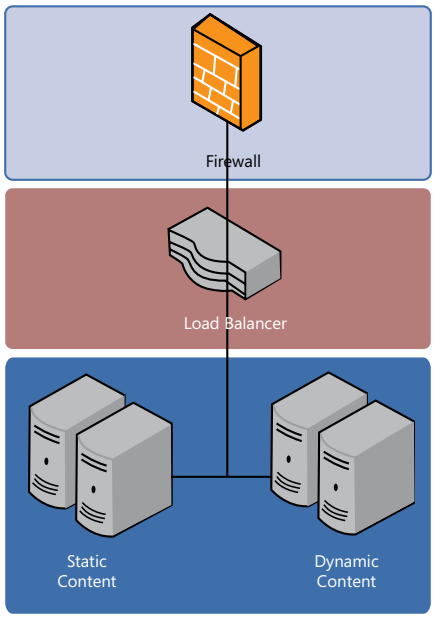
We separate these services into broad segments, with the principle pieces of the architecture being: IIS cluster(s), SQL cluster(s), support infrastructure services, such as Active Directory Services, System Center Operations Manager, centralized storage on either a storage area network or network attached storage, SSL offload clusters, FTP clusters, and other similar clusters.

Separating these services into multiple segments allows the hoster to scale the architecture in different ways. A hoster could scale a SQL cluster differently than a Web cluster and brings to the table a different set of architectural issues.

Another factor that informs the architecture is support for legacy requirements, the best example of which is the FrontPage Server Extensions (FPSE). These are still in use by thousands of customers and are necessary if the mass hosting platform hopes to attract them. These extensions are commonly used by mom-and-pop shops to author simple sites. They are still in use for development tools like Visual Studio and Visual Web Developer for their HTTP uploading functionality, despite the discouragement from Microsoft. Legacy components such as FPSE cannot simply be removed by large hosts without a loss of customer base.

Now let's dive into a few of these clusters within the architecture. The biggest piece is the Web cluster, with the second being the SQL cluster. It's important to remember that one key differentiator between what hosters do, versus other enterprises such as internal IT departments, is that they will attempt to put as many sites or databases on a cluster as possible. Because of this, certain enterprise solutions does not work for them. Additionally, the hosters do not always control what types of applications are placed on a server and so they cannot define capacity in the same way that typical enterprises can.

Figure 1: Application load distribution model



Load Distribution Models

Because there are multiple Web front-ends, the hosting architect has to consider many options for distributing the configuration across all web servers. This configuration is dependent upon the type of load distribution model that is chosen. There are several models for distributing load across multiple Web front ends. We will discuss two that are common to hosting scenarios. These are *application load distribution* and *aggregate load distribution*.

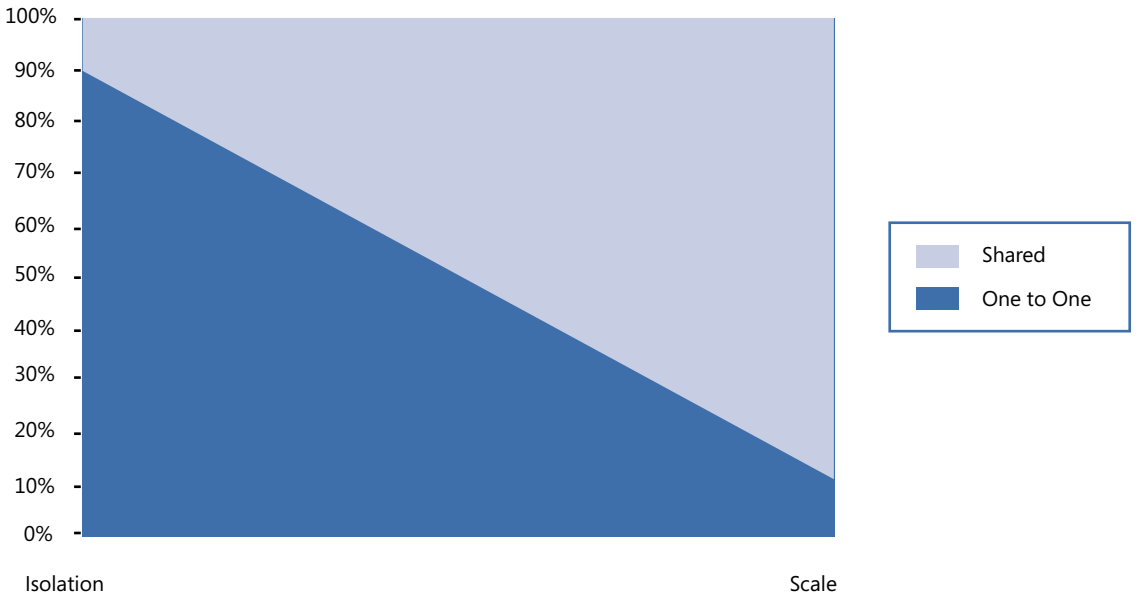
Application Load Distribution

Application load distribution describes a model where load is distributed across multiple Web front end nodes based upon the server's function. This model is typically request-based and leverages the application layer routing capabilities that many network load balancers now support. This model allows hosters to split the Web farm up based on server workloads. Looking at a typical implementation of this model you will see that servers separate from those that serve dynamic content such as ASP.NET or PHP are designated for static content (Figure 1).

Even more granularity can be added to this configuration by further splitting the dynamic servers by their specific function. This means creating smaller sub farms for each type of application. All ASP .NET traffic would be routed to an ASP.NET sub-farm and all PHP content would be routed to another sub-farm. Because dynamic content servers normally require more resources, this design allows the hoster to use a different class of hardware for those sites than would required be for static content.

Most hosters have to consider cost when designing their platform; therefore the application load distribution model may not always be feasible simply because it increases the number of servers involved. Application load distribution also increases the complexity of managing the servers and forces a heavy reliance on networking equipment.

Figure 2: One-to-one application pool scenarios



Aggregate Load Distribution

To keep cost at a minimum and also to make managing the platform less complex, a hoster might choose to implement an aggregate load distribution model. We refer to an aggregate load distribution model as one where all servers share the exact same configuration. Each server in the Web farm is also capable of servicing both static and dynamic content. On systems running IIS, application pool design is vital to maximizing scale in this type of model.

Scaling Microsoft IIS for Mass Hosting

Within a mass hosting company that focuses on the Microsoft platform, there is a constant push for a higher density on each server in the farm. With a complex, HA architecture the hoster increases their density model; however, performance bottlenecks still occur, in particular with application pools.

In IIS, application pool design is vital to achieving maximum density. Failing to take the time to properly plan application pool design may lead to unexpected performance and stability issues. Application pools are truly about isolation and there is direct correlation between the level of isolation you choose and the number of applications you can place on a server. When designing application pools, you must weigh your need for security versus your desired stability. Hosters must choose between two application pool scenarios, a one-to-one scenario or a one-to-many shared application pool scenario. Notice that in Figure 2 one-to-one application pool scenarios tend to trend more toward isolation but away from scale. The shared application pool scenario, on the other hand, trends toward higher levels of scale while trending away from isolation. Ideally, the hoster would choose a solution that allows them to maximize scale without sacrificing isolation and security.

Trending: Isolation vs. Scale

A one-to-one isolation scenario is defined by having an application pool assigned to a single application or in a shared Web-hosting scenario to a single Web site. It allows a hoster to achieve a high level of isolation because each Web site or application runs within a single process and does not share resources with others on the server. This is an optimal solution for a hoster or independent software vendor who must assure customers that others on the same server will not have access to their important data. However, this scenario is limited in mass hosting scenarios. Although it provides you with a desired level of isolation and security, because of memory requirements, it misses the mark in providing hosters the scale they desire. As each application pool runs it consumes memory, and eventually a bottleneck is reached.

Introducing dynamic code into the platform adds a whole new level of complexity. For example, ASP.NET applications increase the amount of memory needed for the application pool. This becomes a problem for the hoster because it limits the number of dynamic Web sites that can run on a server. They begin to see that they can only scale into the hundreds of sites rather than thousands, which is the benchmark for most improvements in hardware technology. Specifically the introduction of 64-bit architecture has afforded many hosters the luxury of adding immense amounts of memory to their servers. While this enables them to move beyond potential bottlenecks, it may also uncover other issues.

Shared Application Pool Scenario

A shared application pool scenario describes a situation where more than one application or website resides in the same application pool. There are two different shared application pool configurations. The first is one-to-N where the hosting provider dedicates a single application pool to a predefined number of websites. The second is a one-to-all configuration where the host places all websites in a single application pool. A shared application pool scenario allows for better scale

because it does not subject the system to the memory constraints imposed by a one-to-one app pool design.

There is some concern that websites and applications in a shared application pool may potentially have access to one another's data. Certain mitigations need to be taken to ensure that these applications and websites are secured. For example, each website needs to have a unique anonymous user and access control list should be applied to the web files. Also, ASP.Net applications should be configured with code access security and set to a medium trust level. These types of steps will help assure that applications and websites on the server are secure even in a shared application pool.

Because all applications run under the same process shared application pools lack the isolation many hosting providers require. This can be a concern in HA scenarios because a problem application affects all other websites and applications in the same pool. For instance a application could cause the application pool to recycle or worse shutdown completely. It is also more difficult for system administrators to identify a problem when there are multiple sites and applications in a single pool. Although there are tools available that allow host to isolate problems within an application pool this task is more easily accomplished when using a one-to-one application pool to website model.

Planning Your Application Pools for High Availability

Hosters are faced with striking a balance between achieving high levels of isolation while maximizing scale. Because of this, many are forced to use a hybrid of the aforementioned application pool scenarios. A typical scenario would include multiple application pools each serving a specific purpose. For instance Web sites that only contain static content might all be placed in a single shared application pool. This is possible because static content is not associated with the security and performance concerns that come with dynamic content. All other application pools would be dedicated to sites that contain dynamic content. This allows the hoster to allot greater resources to these sites. This configuration is more prevalent in shared hosting environments where a single platform must service customers who have both static and dynamic content.

Configuration Replication

Another fundamental requirement for mass hosting, HA environments, is maintaining state and configuration across all Web front ends in the farm. Although there are other configurations that must exist on each front end, the most important is that of the Web server. Some Web servers have support for a centralized configuration store. For those that do not, some software solution must be implemented to replicate the configuration across all servers.

Content Storage

One of the central principles to a highly scalable, mass-hosting platform that architects face is determining the location of the customer content that will be hosted. In most cases, you are dealing with content that either lives within SQL or on disk. Since the front end of the architecture is clusters configured with thousands of sites it is not practical to localize the content to on-board, directly attached disks. The below sections break-down the different storage architectures that are common among hosting companies.

Direct Attached Storage (DAS)

DAS is one of the most common and classic storage methods used by Web hosting companies. These are stand-alone Web-servers where content is stored locally, The primary benefit being that if a single stand-alone server goes down, the entire customer-base is not offline. One of the major downfalls is that the customers are susceptible to any type of hardware failures, not just a disk subsystem failure. Additionally, this type of configuration introduces issues such as density limits, migration problems, and lack of high-availability for the Web sites and load-distribution.

"THE INTRODUCTION OF 64-BIT ARCHITECTURE HAS AFFORDED MANY HOSTERS THE LUXURY OF ADDING IMMENSE AMOUNTS OF MEMORY TO THEIR SERVERS. WHILE THIS ENABLES THEM TO MOVE BEYOND POTENTIAL BOTTLENECKS, IT MAY ALSO UNCOVER OTHER ISSUES."

Network Attached Storage (NAS)

Most hosts that have gone with a highly scalable platform have opted to use NAS as their centralized remote storage for all of their customers. In a highly available Windows environment, the NAS is accessed via Common Internet File System (CIFS), usually across a dedicated storage network. Customer Web content is stored centrally on the NAS with the path to the NAS translated into a logical path using Distributed File System (DFS). The combination of NAS and DFS allows a Windows-based hoster to spread customers across multiple NAS storage subsystems, limiting the impact of a global issue to those customers.

Optimizing CIFS, TCP/IP, and the NAS plays heavily into how scalable the NAS is with the number of simultaneous customer sites for which it can be serving content. With poor optimization on the NAS, hosts can introduce issues that can affect the entire customer base. However, hosts also mitigate this by using multiple NAS subsystems for different segments of customers and dedicating storage networks and interfaces to this type of traffic.

Many NAS subsystems have mirroring and snapshot capabilities. These technologies are leveraged by the hoster to provide a solid process for disaster and recovery—especially considering the storage system could be holding content for tens of thousands of customers.

One issue with a pure NAS storage subsystem is that since technologies such as SQL do not support remote storage of their databases across CIFS, this limits the hoster in the types of services they can offer.

Storage Area Network (SAN)

Many enterprise storage systems are capable of operating as both a SAN and a NAS with the key difference being the method used to connect to them. In the case of a SAN, the principle methods are Fiber Channel (FC) and iSCSI.

Hosting companies are capable of building highly available, scalable SQL and mail systems, such as Exchange, by leveraging the capabilities of a SAN as the centralized storage for these types of clusters. Additionally, the more advanced the SAN, the more options the hosting company has to perform tasks such as snapshot and recovery management within SQL or Exchange.

One of the major detractors to an enterprise storage system is the cost the hoster incurs. Hosts are careful to ensure that the product they are about to deploy will have a high enough return on investment (ROI) to warrant the cost of a SAN.

> "THERE REALLY IS NO GOOD, COST EFFECTIVE WAY OF BUILDING A SCALABLE, HIGHLY AVAILABLE AND DENSE SQL CLUSTER PLATFORM. EACH CLUSTER TOPOLOGY HAS ITS DISADVANTAGES COMBINED WITH THE FACT THAT HOSTS HAVE NO CONTROL OVER CUSTOMER APPLICATION DESIGN."

SQL Cluster

SQL is a major service that many hosts offer to a majority of their customers. However, it is also one of the key areas that many hosts do not implement as a cluster. There are a variety of reasons for this, with the chief reason being cost and licensing. However, the hosts that do go with a highly available SQL cluster have to design their architecture so that they select the right type of clustering methodology that will support numerous databases.

Unlike other enterprises where the SQL cluster consists of a relatively small number of databases, hosting companies will deploy hundreds, if not thousands, of databases to a single database cluster. This cluster must be resilient in both performance and uptime. Since hosting companies have no control over how their customers write their applications, some unique issues are presented when designing a cluster for mass hosting. In a hosting environment, each customer has 1 – *n* databases assigned to them. These databases can be stored on a single cluster or distributed across multiple clusters. The most common cluster a hoster would build for mass SQL hosting is the standard active-passive SQL cluster. However, as hosts move into software as a service (SaaS) hosting, the requirements on the SQL cluster transform from node redundancy to redundancy of data. This introduces more issues since these same systems will still host numerous databases.

There really is no good, cost effective way of building a scalable, highly available and dense SQL cluster platform. Each cluster topology has its disadvantages combined with the fact that hosts have no control over customer application design. The ideal SQL cluster would allow hosts to do load distribution along with database mirroring without having to deal with the problems of transaction collisions while maintaining a very dense number of databases across the cluster(s).

Conclusion

Services providers must continually rise to the challenge to meet the growing customer demand for new sets of services that offer greater value to small- and medium-size businesses, developers, consumers, and designers. They must provide a high degree of service to maintain customer loyalty and achieve their business goals. Services providers seek ways to deliver the "silver bullet" Web platform that helps reduce total cost of ownership (TCO) and deliver customer satisfaction effectively and efficiently.

When referring to a web platform solution comprised of web serving, data storage, and database storage, a bottleneck inevitably occurs within one of these components. A solution is only as strong as its weakest link. Fundamentally, every component of a solution needs to scale outwardly and upwardly with redundancy and cost efficiency.

This article is not about what technologies (open source or closed source) can fill the gap or have already done so in some areas, but rather the underlying architectural problems that are not "top of mind" when technologies are being developed. It isn't sufficient, these days, to create technology for the sake of technology in an attempt to fill a niche. Only through strategy, planning and developing of large scale solutions, will we be able to support large scale requirements. The world is growing, needs are mounting, and it only makes sense infrastructures do as well.

About the Authors

Shaun Hirschman has worked in the hosting space for over eight years and is very in tune with the inherent business and technical challenges. Prior to joining Microsoft 2006, Shaun started out in the hosting industry as technical support and rose to a senior windows systems administrator position. He has considerable technical knowledge and experience with Microsoft-based systems. He mostly enjoys playing with the bleeding edge of software and hardware technology in an attempt to appease his thirst for knowledge.

Mathew Baldwin has been working within the hosted services space for the last ten years. In the past he has held roles as architect, senior systems engineer, regular troubleshooter,and consultant for Microsoft-based platforms with multiple hosted services companies. He was instrumental in bringing to market the first highly-available, clustered Windows 2003 shared hosting platform and has worked closely with Microsoft on a variety of initiatives. He currently works as a senior architect for Affinity Internet.

Tito Leverette has worked in the hosting space for more than eight years. He started his career with Interland Inc., now Web.com. Before coming Microsoft Tito was an architect on the Web platform team for SunTrust Inc., the ninth largest bank in the United States. Tito has years of experience designing, building, and deploying Web infrastructures for large enterprises as well hosting clients. He is a graduate of the Georgia Institute of Technology (Georgia Tech).

Michael Johnson spent more than seven years employed in the Web services provider space, where he developed and architected highly scalable applications and solution on Microsoft products. He is currently working as a Web architect evangelist for Microsoft.

Delivering End-to-End High Productivity Computing

by Marc Holmes and Simon Cox

Summary

Performing a complex computational science and engineering calculation today is more than about just buying a big supercomputer. Although HPC traditionally stands for "High Performance Computing", we believe that the real end-to-end solution should be about "High Productivity Computing". What we mean by "High Productivity Computing" is the whole computational and data handling infrastructure and also the tools, technologies and platforms required to coordinate, execute, and monitor such a calculation end-to-end.

Many challenges are associated with delivering a general high productivity computing (HPC) solution for engineering and scientific domain problems. In this article, we discuss these challenges based on the typical requirements of such problems, propose various solutions, and demonstrate how they have been deployed to users in a specific end-to-end environmental science exemplar. Our general technical solution will potentially translate to any solution requiring controlling and interface layers for a distributed service-oriented HPC service.

The solution considers the overall value stream for HPC solutions and makes use of Microsoft technology to improve this value stream beyond simply performance enhancements to hardware, software, and algorithms which—as is described—is not always a viable option. Key to our solution is the ability to interoperate with other systems via open standards.

Requirements of High Productivity Computing Solutions

In the domains of engineering and science, HPC solutions can be used to crunch complex mathematical problems in a variety of areas, such as statistical calculations for genetic epidemiology, fluid dynamics calculations for the aerospace industry, and global environmental modeling. Increasingly, the challenge is in integrating all of the components required to compose, execute, and analyze the results from large-scale computational and data handling problems.

Even with such diverse differences in the problems, the requirements for the solutions have similar features, due to the domain context and the complexity of the problem at hand.

Designed for a solution to a specific problem

Because the calculations and industry involvement are diverse, there are no particular solution providers for any given problem, resulting in highly individualized solutions emerging in any given research department or corporation requiring these calculations. This individuality is compounded by the small number of teams actually seeking to solve such problems and perhaps the need to maintain the intellectual property of algorithms or other aspects of specific processes. Individuality is not in itself an issue—it may be a very good thing—but given that the technical solutions are a means to an end, it is likely that these individual solutions are not "productized" and thus are probably difficult to interact with or obscure in other ways.

Long-running calculations and processes

The commoditization of the infrastructure required to perform large scale computations and handle massive amounts of data has provided opportunities to perform calculations that were previously computationally impractical. Development of new algorithms and parallelization of code to run across computing clusters can have a dramatic effect, reducing times for computation by several orders of magnitude. So a calculation that will complete "sometime shortly after the heat death of the universe" could perhaps run in several weeks or months. This success has enabled industries to achieve significant results and build upon this enablement to provide further direction for development.

Requirements for provenance information

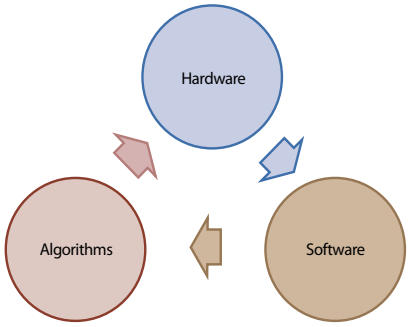
In areas of research, there is a critical need to ensure a useful trail of information for a variety of reasons. There may be simply a need to rerun algorithms and to ensure that the same result sets are produced as a "peace of mind" exercise, but more likely this will be required as part of proof and scientific backup for the publication of research. There may also be statutory reasons for such provenance information: in the aerospace industry, in the event of an air accident investigation, engineers making specific engineering choices may be liable for the cause of the accident and face criminal proceedings. Therefore, the need to recreate specific states and result sets as required, and to follow the decision paths to such choices is of extreme importance. The complexity of such tasks is compounded when one considers that the life cycle of an aircraft design could be 50 years.

Significant volumes of data and data movement

Deep Thought performed one of the largest HPC tasks in recent

(fictional) memory. Given a simple input ("What is the answer to life, the universe, and everything?"), he provided a simple output ("42") albeit after several million years of processing and a small embarrassed pause at the end of the process.

Figure 1: Efforts to improve the speed of computation fall in three interrelated zones forming a cycle.



The reality, however, is that significant calculations requiring significant amounts of processing are very likely to involve significant amounts of data throughout the life cycle of the calculation. Even with Deep Thought-style input and output simplicity, the operational data sets within the problem space during calculation may be significant. Strategies need to be developed to handle this data, and its metadata. Given the need for provenance information, these strategies need to be both flexible and robust and integrated into the workflow processes used to coordinate the calculations.

Enhancing Value of High Productivity Computing Solutions

Because of the complexity of the domain problems, and the requirements described above, the net result is that solutions are typically designed to get the job done. This in itself is something of an achievement, and that is leaving aside the intellectual effort involved in the development of the computational algorithms and conceptual solutions in the first place.

There is, of course, a simple way to improve the value of an HPC solution: Make it quicker. However, for problems of sufficient computational complexity, at a certain point it becomes fruitless to continue to attempt to improve the solution because of technology constraints. The efforts to improve the speed of computation can be represented by the cycle shown in Figure 1.

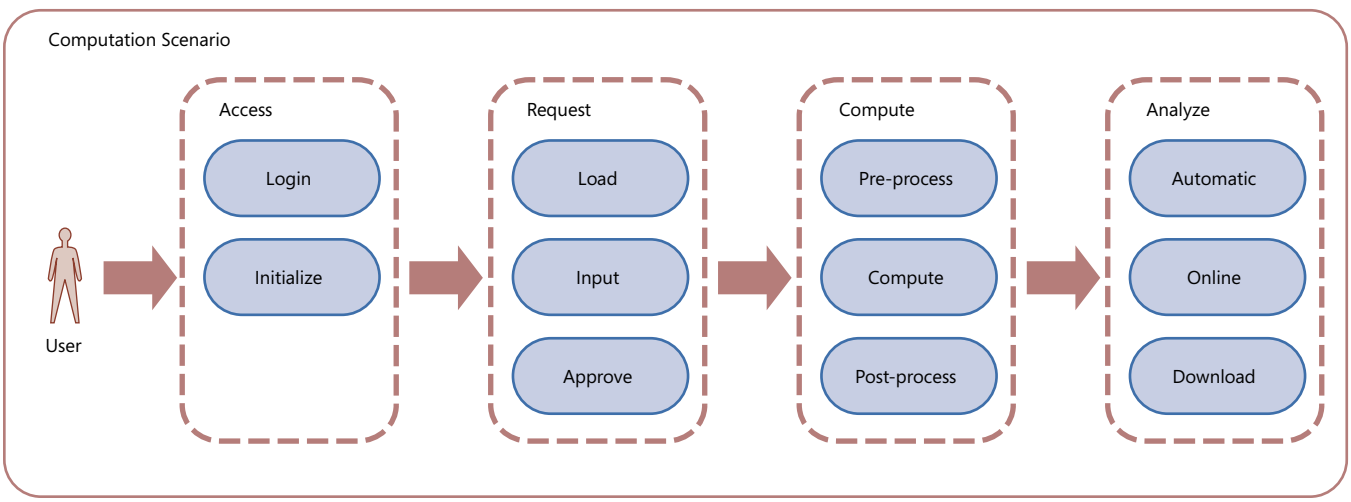
To improve the speed of calculation, a development team may:

- Use better or more hardware. The benefits of faster processors, faster disks, more memory, and so on, could be limited by the capability of software or algorithms to utilize the available hardware. It is also limited by the release cycles of next-generation hardware and is probably severely limited by budget.
- Use better or more software. The algorithms themselves are more likely to be a bottleneck than the underlying software, but from time to time, there will be improvements in data handling or other such features to provide a useful enhancement. This may also suffer from budgetary constraints.
- Use better algorithms. Better algorithms require invention that simply might not be possible and is probably less predictable than software or hardware improvements, although when it does occur may provide the most significant improvement of all.

So the challenge for continued platform enhancement on a fundamental level is straightforward to understand, but not easy to achieve. As a result, the teams using HPC solutions tend to be pragmatic about the length of time it takes to complete calculations as they understand that they are pushing available technology and, as previously mentioned, can be quite happy to be completing the calculation at all.

Once computational times have been reduced as far as practical, enhancing the value of such solutions is then about consideration for the overall process and the implications of performing the calculations to further reduce the overall time to new scientific or industrial insight. Given the research/engineering nature of the problems, then the overall process is typically a human workflow on either side of the computational task. It is also about the development of a solution set to provide

Figure 2: General computation scenario



interfaces and controls to enable an effective translation of the overall process, allowing researchers and engineers to carry out their tasks more quickly and efficiently.

Defining the Problem and Solution

Utilizing the available portfolio of Microsoft technologies allows us to create a fuller architecture for a HPC solution and enables the fulfillment of features that drive additional value to teams using the solution. Furthermore, use of technologies can build seamlessly and interoperate with existing infrastructures. This section of the paper will describe a possible architecture and some of the features of that architecture and the value they provide.

Problem Scenarios

It is difficult to fully generalize an HPC solution, due to the specific needs of an individual domain and the business processes that emerge as part of the engineering challenges that are typically present. However, we could consider three scenarios for the solution. The first is the core user scenario: completing a computational job. Two other scenarios, administration and authoring, support this process.

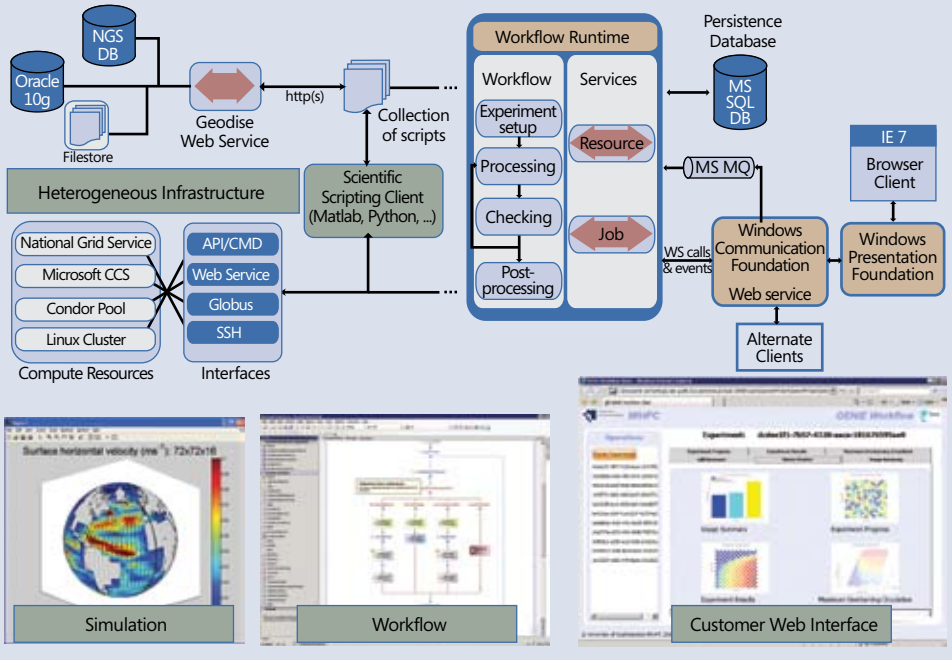
End-to-End High Productivity Environmental Science

The Grid-ENabled Integrated Earth (GENIE) system model project provides a grid-enabled framework that facilitates the integration, execution and management of constituent models for the study of the Earth system over millennial timescales. Simulations based on the GENIE framework need to follow complicated procedures of operations across different models and heterogeneous computational resources.

The GENIE project has developed a framework for the composition, execution and management of integrated Earth system models. Component codes (ocean, atmosphere, land surface, sea-ice, ice-sheets, biogeochemistry, and so on) of varying resolution and complexity can be flexibly coupled together to form a suite of efficient climate models capable of simulation over millennial timescales. The project brings together a distributed group of environmental scientists with a common interest in developing and using GENIE models to understand the Earth system. Earth system simulations are both computationally intensive and data intensive. The GENIE framework has been designed to support running of such simulations across multiple distributed data and computing resources over a lengthy period of time. We exploit a range of heterogeneous resources, including the U.K. National Grid of parallel computing resources (running both Linux and Windows Compute Cluster Server) and desktop cycle stealing at distributed sites. Our back-end data store uses Oracle 10G to store metadata about our simulations and SQL Server to coordinate the persistence tracking of our running workflows. (See Figure 1.)

At Supercomputing 2006 in Tampa, Fla., we showed how applying the workflow methodology described in the article can provide the GENIE simulations with an environment for rapid composition of simulations and a solid hosting environment to coordinate their execution. The scientists in the collaboration are investigating how Atlantic thermohaline circulation —sea water density is controlled by temperature (*thermo*) and salinity (*haline*), and density differences drive ocean circulation—responds to changes in carbon dioxide levels in the atmosphere and seek to understand, in particular, the stability of key ocean currents under different scenarios of climate change.

Figure 1: High Productivity Environmental Science end-to-end support: (a) Customer Web Interface using Windows Presentation Foundation (b) Windows Workflow Foundation to author and coordinate simulations, and (c) Heterogeneous Infrastructure consisting of Windows and Linux Compute Resources, and Oracle and SQL Server data storage.



Microsoft Institute of High Performance Computing: Matthew J. Fairman, Andrew R. Price, Gang Xue, Marc Molinari, Denis A. Nicole, Kenji Takeda, and Simon J. Cox
External Collaborators: Tim Lenton (School of Environmental Sciences, University of East Anglia) and Robert Marsh (National Oceanography Centre, University of Southampton)

Figure 3: General authoring scenario

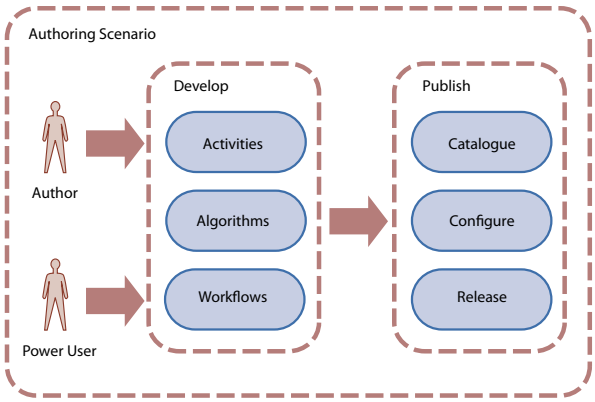
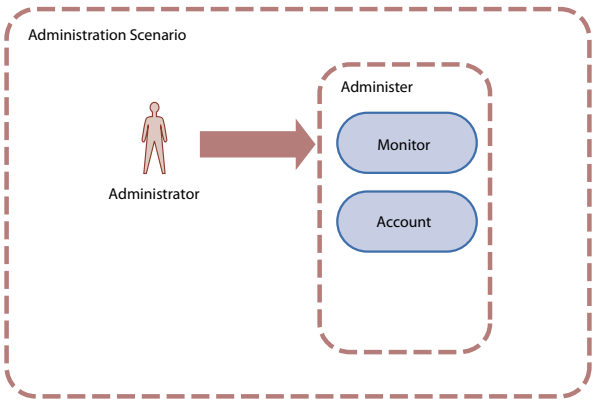


Figure 4: General administration scenario



Computation Scenario. The computation process is divided into four core steps for an average user, which will ensure task completion: access, request, compute, and analyze. (See Figure 2.)

The Access step:

- *Login.* A user must be able to login to the solution and be identified as a valid user. They are likely then to be able to see only the information that is relevant to their role and/or group.
- *Initialize.* Prior to making a job request, there may be a need to set up a dashboard for the execution of the job. Given the nature of a long-running computation, one run of the process may be considered a "project."

The Request step:

- *Load.* A user must be able to upload or access data sets that are intended to be used as part of the job. Data sets may be large, nonhomogeneous or sourced externally, so a specific step may be necessary in the workflow to obtain that data and partition it across the cluster node in a way that balances the expected workload.
- *Input.* A user must then be able to add parameters and associated metadata to ensure that the job will run successfully. These parameters are captured for reuse and auditing.
- *Approve.* Following the input of parameters and any other required information, it is likely that approval will be required before a job is submitted. Presumably then, some kind of typical approval workflow will ensue, following a user job submission.

The Compute step:

- *Preprocess.* The preprocessing step for a job may do several things. It will likely move the input data sets to the cluster nodes and may perform an initial set of computations, such as the generation of analytical measures for use in the main computation. It may also initialize data structures and assess all data for validity prior to execution.
- *Compute.* This phase represents the actual parallel computation. Each node will execute a portion of it and may be required to pass intermediate results to other nodes for the whole process to complete (message passing jobs). Alternatively, the data may lend itself to be computed into totally isolated portions, such as in parametric sweeps, or what-if scenarios. This phase may take place over a potentially significant duration. Ideally, some feedback is provided to the end user during this time.
- *Post-process.* The final step in the actual computation is similar to preprocessing in that it is now likely that some work is completed with the results data. This may involve data movements to warehouses, aggregation of the separate node data, clean-up operations, visualization tasks, and so on.

The Analyze step:

- *Automatic.* Although the results from a job are likely to need specialist intervention to truly understand, some automatic analysis may be possible, particularly in the case of statistical analysis where patterns are important and are easy to automate.
- *Online.* A user should be able to perform some core analysis without requesting the entire result set. This may be presented as tools with typical business intelligence paradigms—slicing and dicing, and so on.
- *Download.* Finally, a user should be able to retrieve result sets for advanced offline manipulation as required.

Authoring Scenario. The authoring scenario covers the aspects of the overall process to provide the capabilities for an end user to complete a computation job. Broadly, a process author should be able to develop new capabilities and processes and publish those for end user consumption. Figure 3 shows the authoring scenario for two actors, authors and power users. The difference between these roles is that an author will likely develop code, whereas a power user is likely to be configuring existing code. The authoring scenario can be described as follows:

The Develop step:

- *Activities.* A developer may wish to construct discrete process elements or activities that can be used as part of an overall computation job process. An example may be a generic activity for performing FTP of result sets.
- *Algorithms.* A developer may be developing the actual algorithms for use in the compute step of the end user process.

- *Workflows.* A developer or power user may define specific computation workflows by joining together activities and algorithms to create an overall compute step consisting of pre- and post-processing as well as the actual compute task.

The Publish step:

- *Catalog.* Following development of activities, algorithms, and workflows, the author will want to catalog the development for use by another author or the end user.
- *Configure.* Some configuration of activities or workflows may be required, such as access restrictions, or other rules governing the use of the development work.
- *Release.* Finally, the development work should be exposed by the author when it is ready for use. Ideally, this will not require IT administration effort.

Administration Scenario. Finally, administration of the HPC platform is required. In this case, we are considering that the main tasks to support the end user involve monitoring and accounting and are excluding other core administration tasks such as cluster configuration and other general IT infrastructure activities. (See Figure 4.)

The administration scenario can be described as follows:

- *Monitor.* Clusters are typically shared resources, managed by specialized personnel. They will monitor the installations for functionality and performance, e.g. analyzing the job queues and the resource consumption trends.
- *Account.* It is desirable to account for usage of resource utilizations by certain accounts. This helps when the clusters are funded and used by several groups.
- *Billing.* It is possible to build charge-back models to explicitly bill for usage.

Solution Scenarios

Having described a general series of scenarios that may represent the requirements for an HPC solution, and considering the values described at the outset in the construction of an HPC solution, we can now consider some solution scenarios to satisfy these requirements.

High Performance Computing with Microsoft Cluster Compute Server Edition

The core service required for the solution is the actual High Performance Computing cluster capability. Microsoft Cluster Compute Server Edition (CCS) provides clustering capabilities to satisfy the Compute step of the problem scenarios. (See Figure 5.)

CCS provides clustering capabilities to a cut-down version of Windows Server 2003 allowing, for example, the use of parallelized computations, taking advantage of many processors in order to complete particularly complex, or intensive, calculations such as those found in genetical statistics.

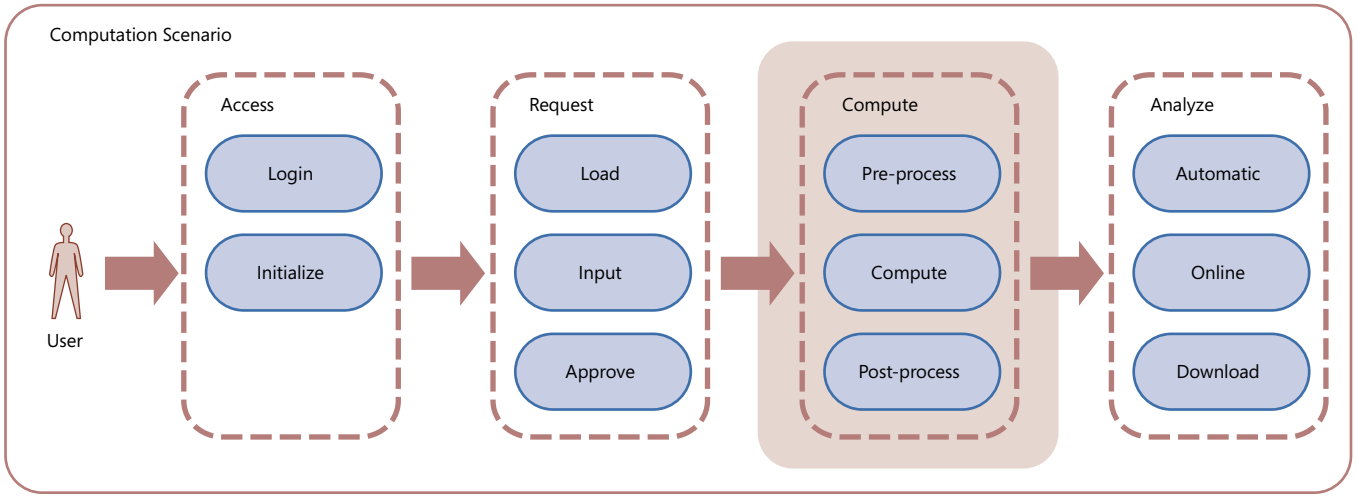
Control of a CCS cluster is handled by a "head node" which communicates with "cluster nodes" to issue instructions and coordinate a series of tasks on a particular job. The cluster nodes may exist on a private network—for optimal performance, this will likely be a low-latency network to ensure that network latency does not impact on computation times. The resources section at the end of this document contains links to technical articles on deploying and configuring CCS.

The head node is equipped with a "job scheduler" to allow the insertion of a computation job into a queue for execution when the desired or required number of processors become available to complete the job. A small database is provided for the head node to keep track of the current job queue. The job scheduler can be accessed via an interface on the head node, via command line—which can be parameterized or executed with an XML definition of the parameters, or via an API (CCS API) for programmatic interaction.

A CCS cluster can handle several types of job. These are:

- *Serial job.* This is a typical compute task of simply executing a required program.
- *Task flow.* This is a series of serial jobs consisting of potentially multiple programs/tasks.

Figure 5: Microsoft Cluster Compute Server Edition satisfies the Compute step.



Considering the Value Stream

Typically, the users of high-productivity computing solutions are highly skilled and highly prized assets to the research and development teams. Their skills and expertise are needed to provide the inputs to the processes, the algorithms for the calculations, and the analysis of the resulting output. However, these assets should be allowed to work as effectively and efficiently as possible.

The creation of obscure (in one sense or another) solutions will affect this in a number of possible ways:

- The user may need to have some understanding of the system that goes beyond what would typically be expected. For instance, I have to understand the basic principles of a remote control in order to operate my TV, but I don't need to understand the voltage required to power its LCD screen to watch it.
- Experts who have been involved in an individual solution may find it hard to escape. For example, if Alice has coded an algorithm for the solution, she may find that she becomes heavily involved in the day-to-day operation of the solution because only she understands the best way to interact with the system to use the algorithm. She should really be put to use creating even better algorithms. The result after one or two generations of improvement is both a high cost of solution operation and a risk problem since Alice has become a key component of the solution.
- The solution may also be a problem to administer for core IT teams. An HPC solution crafted by domain experts could easily be analogous to the sales team "linked spreadsheet solutions" that can be the stuff of nightmares for any in-house development and support teams.

The diagram in Figure 1 represents a high-level process for the provision and use of an HPC solution.

Figure 1: Original value stream

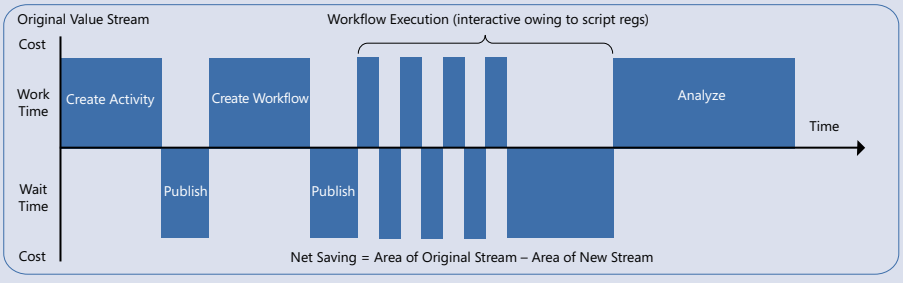
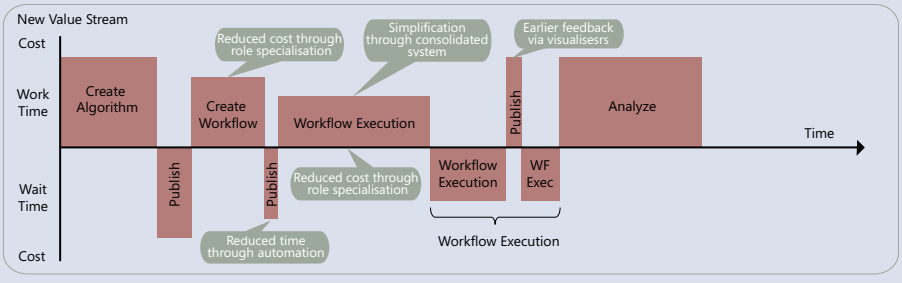


Figure 2: Improved value stream



The compression of the value stream in terms of time and cost should be the primary concern for the provision of a solution. Given the issues described with speed improvements to the computational work, then improvements should be identified outside of this scope and in other areas of the process.

Suitable candidates for the cost reduction in the process are, broadly, all aspects of the value stream with the probable exception of the algorithm creation, which is the invention and intellectual property of the solution. Reduction of cost is in two parts: the cost to use the solution and the cost of administration. Improving the process for the publication and use of algorithms and providing analytical support could reduce the overall process cost.

Based on the consideration of the value stream, a solution set should consist of tools providing two core features: simplification of task completion and opportunities for improved interaction.

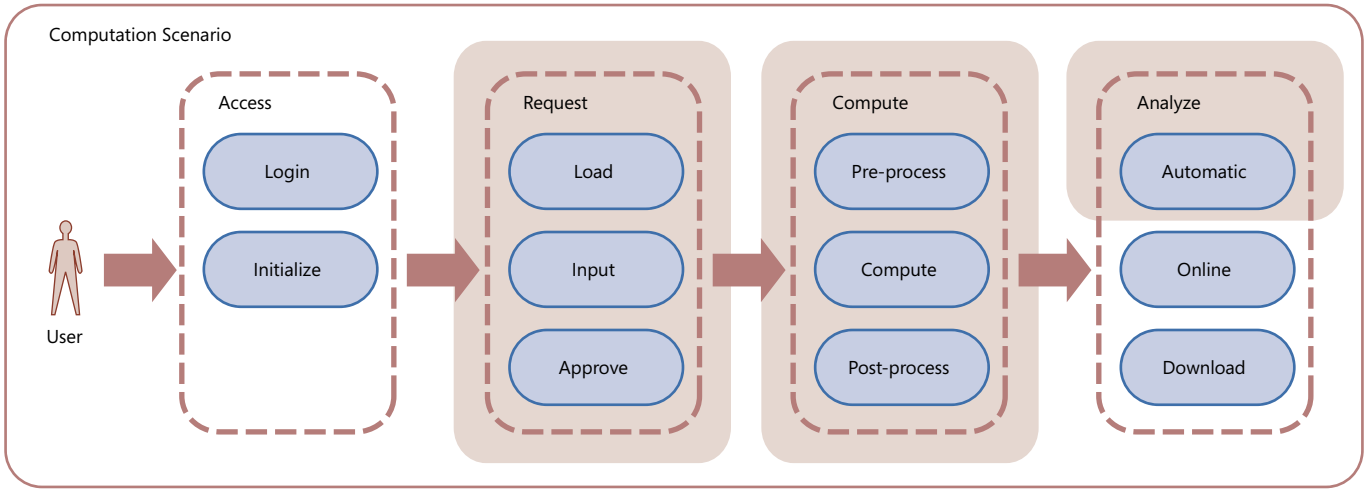
An Improved Value Stream

Providing an HPC-based solution using architecture similar to that described may have several benefits for the overall value stream (Figure 2).

There are several areas that could be initially affected:

- *Planning for a long-running job*. Reductions of both time and cost are possible through the use of intuitive interfaces, validation, and other logic to ensure correct capture of information. In addition, specialist coders or administrators may no longer be needed at this point in the process.
- *Publishing a long-running job*. Both cost and time may be reduced because the information is captured by the system, and consequently, the publication of the job can be fully automated.
- *Cost of computation*. Cost is reduced through role specialization—monitoring can occur from an end-user perspective—and the overall cost across many jobs may be reduced through improved recognition of failing or erroneous jobs which can be cancelled before they have completed execution. This improved recognition is again provided by the use of feedback mechanisms and monitoring presented to the user interface.
- *Analysis*. Analysis can begin more quickly if the user interface presents information during the job execution; some automatic analysis may be available, reducing initial costs before specialist intervention is required.

Figure 6: Windows Workflow Foundation can provide a complete application layer covering most areas in the Request, Compute, and Analyze steps.



- *Parametric sweep.* This type of job would execute multiple instances of a single program but with a series of differing parameters, allowing for many result sets in the same time frame as a single task.
- *Message Passing Interface job.* The most sophisticated job would be the use of a Message Passing Interface (MPI) to parallelize an algorithm and split the computation job amongst many processors, coordinating to provide speed increases for long and complex computations.

Any or all of these types of jobs may be found inside a single computation process. CCS should be able to provide a powerful platform for any of the pre- or post-processing tasks as well as the main computational task.

Application Logic with Windows Workflow Foundation

Because we can consider the overall computation process to be an instance of a long-running human and machine workflow, then we can consider Windows Workflow Foundation (WF) as a suitable starting point for the construction of an application layer and framework for the HPC solution. In fact, WF has several components which can used to provide a complete application layer for the HPC solution. The particular (but not exhaustive) areas that WF can cover are shown in Figure 6:

WF forms a core part of the .NET 3.0 framework and provides a full workflow engine that can be hosted in a variety of environments. The resources section of this document contains several links to more information on Windows Workflow Foundation.

Some of the core features of WF for consideration are as follows:

- *Sequential and state workflows.* The WF runtime can handle sequential and state-driven workflows, so a huge variety of processes can be described. These workflows can also have retry and exception handling.
- *Activity libraries.* Workflows consist of "activities" such as decision making, looping, and parallel execution, as well as arbitrary "code" activities, and several of these are ready-to-use in .NET 3.0. Additionally, because WF is used to power server products (for instance Microsoft Office SharePoint Server 2007), these products also have base activities for consumption inside WF. Finally, activities can be constructed as required to create a bespoke library to suit a specific requirement.
- *Rules engine.* WF has a rich forward-chaining rules engine which can be used for decision making inside a workflow, but can also be executed outside of workflow instances. Activities are designed to work with this rules engine.
- *Designer and rehosting.* WF also has a full drag and drop design surface used inside Visual Studio 2005 but can also be re-hosted inside (for example) a Windows Forms application.
- *Runtime services.* The WF runtime can have services added prior to workflow execution to intercept the execution of a workflow and perform actions such as persistence or tracking. Services can also be constructed as required.
- *Extensible Application Markup Language (XAML).* Finally, WF makes extensive use of XAML to describe workflows and rulesets, meaning that serialization is trivial, and the generation of truly bespoke design surfaces and rules managers is possible.

Given these features, we can see how WF can provide capability to the architecture.

Activity Libraries

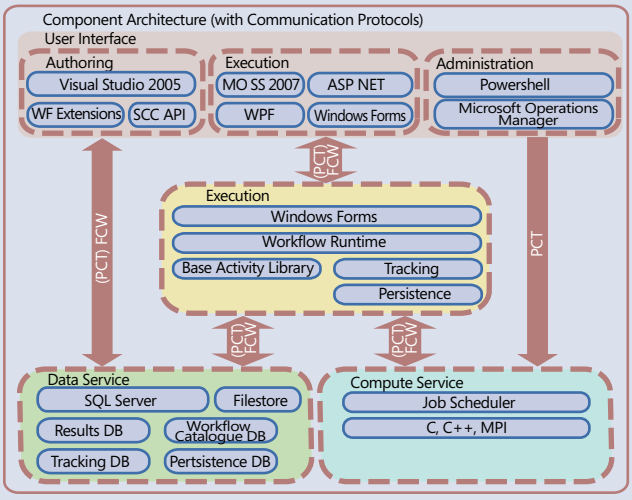
Activity libraries are the building blocks needed for the HPC solution. Examples are:

- Cluster communication. It is possible to build a custom activity that uses the CCS application programming interface to communicate with the job scheduler and create or cancel jobs on the cluster. This custom activity can then be used by any workflow needing to communicate with the job scheduler and can provide a higher level of abstraction to "power users" who may need to author new workflows or amend existing workflows without detailed programming knowledge.

Component Architecture

The core architecture looks like a familiar n-tier application architecture and, broadly speaking, this is indeed the case. The organization of functionality is quite logical. Figure 1 shows the required components for the architecture.

Figure 1: Components



User Interface

The user interface experience can be broken into three parts. Each performs a different function of the overall solution:

- The main user experience for planning and executing a comput ation can be provided through various technologies. Examples could be Windows Forms, Windows Presentation Foundation, ASP.NET or Microsoft Office SharePoint Server. The technology choice should be appropriate to a particular environmental characteristic (for example, the use of Web technologies where wide availability is required or WPF where rich interaction is required).
- The workflow authoring experience is provided by Visual Studio 2005 using the Windows Workflow Foundation (WF) authoring design surface. Using the "code beside" model of developing workflows enables the use of a new Source Code Control API interface to submit WF "XAML" files to a central database repository, for example, for execution in the user interface.
- For IT administrators, Microsoft Operations Manager can be used to monitor the health and status of the HPC cluster, though for end users of an HPC system, more user friendly feedback can be made available through the use of Windows PowerShell.

Application Layer

The application layer consists of the WF runtime hosted as an NT Service. This layer is used primarily to communicate with the cluster job scheduler but offers a variety of functions:

- An activity and workflow library for controlling the cluster and performing data movements and other pre- and post-processing steps.
- A rules and policy engine for managing access to a cluster and potentially providing priorities and "meta-scheduling" capabilities along with security.
- Tracking information for job execution providing provenance information as required.
- Persistence information allowing scaling of the solution and providing robustness to long running workflows and potentially the ability to re-execute a workflow from a particular state.

Data Service

The data service contains supporting databases such as tracking and persistence stores for the workflow runtime. In this architecture, it is also likely that there is a store containing results or aggregates of results. Input and output files are likely to be held as files for easier movement around the cluster to the respective compute nodes. Finally, there is a database for holding a catalog of available workflows for execution

Compute Service

The compute service is the "simplest" part of the architecture, consisting of a cluster of x nodes in a desired configuration

Communication

Communication throughout the architecture is handled using Windows Communication Foundation through a TCP (Remoting) or HTTP channel. This keeps the architecture decoupled and scalable, and offers advantages for differing user interfaces based on user requirements (authoring, execution, reporting).

Security

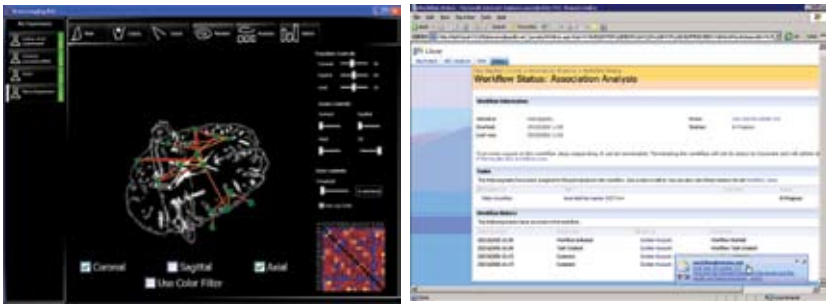
In a distributed service-oriented framework potentially crossing multiple security domains, federation of identities to permit single sign-on and user level access control to all aspects of the component architecture is essential. Technologies offering such capabilities include those based around certificates, active directory federation (along with extensions to interoperate with Unix systems), and Microsoft's Windows CardSpace.

- Higher level, "strongly typed" cluster job activities targeting specific (rather than arbitrary) executables
- Activities for data movement around the network—perhaps using WF to control SQL Server Integration Services
- Data upload and FTP activities, for results movement
- Retry behaviour in the event of some failure of the process
- Notification activities for events during the process
- Rules-based activities to govern automated optimization of the process, or control access privileges.

Workflows for specific computation scenarios can then be assembled using a combination of these generic activities and some specific activities to enable the desired scenario.

Workflow activities conform to all the typical rules of inheritance so it is worth noting that in a heterogeneous environment with multiple clusters of differing types, activities could be written to a general interface and then selected (at design time or run time) to allow execution on a given cluster. A useful example of this is to create an activity that allows the execution of an MPI computation on a local machine—perhaps making use of two processors—for testing purposes.

Figure 7: Using WF-based application layer, Windows Presentation Foundation (left) and Microsoft Office Sharepoint Server (right) offer diverse user experiences.



Services

WF provides several services which can be used as required by attaching the service implementation to the workflow runtime. Three of the most useful for HPC are:

- *Tracking service.* This uses the notion of tracking profiles and a standard database schema to audit occurrences in a workflow. This can be extended as required, or replaced in a wholesale fashion with a custom implementation. The out-of-the-box service is generally sufficient to provide quite a detailed audit trail of workflow execution.
- *Persistence service.* This uses a standard database schema to dehydrate and rehydrate workflow instances as required by the runtime, or as specified by host. Rehydration of a running workflow happens when an event on that workflow occurs, making WF an ideal controller for long-running workflows, such as week, or month, long computations. Persistence also provides scalability across multiple controlling applications, and enables the use of "change sets" to rollback a workflow to a previously persisted position and re-execute from there. This has potentially significant advantages for multistage complex computations and for rerunning scientific research computations in a given state.
- *Cluster monitor service.* This service can register for events and callbacks from the cluster and then perform new activities within a workflow. For instance, the monitoring service may listen for an exception during the computation, or be used to cancel a running computation on the cluster.

Additional services could be constructed alongside these. An example may be a billing service which monitors the durations of processor usage and "bills" a user accordingly.

Security and Policy

WF has a powerful forward-chaining rules engine that can be used inside a workflow instance. Rules can be created in code, or declaratively, and are then compiled alongside the workflow for execution.

Rules and policy could have several uses:

- *User identity.* Access to system resources or types of resources could be granted on the basis of identity, or additional services and workflow steps could be added and removed on the basis of identity.
- *Optimization.* Some automation of optimization or business process could be provided through the use of rules combined with business intelligence or other known parameters again to complete or remove workflow steps as required.
- *Metascheduling.* CCS does not currently have a metascheduler, but WF could be used to designate particular clusters for execution based on parameters, such as requested number of processors or current status of the available clusters, or user identity.

So there is significant power and flexibility in WF. We can also see later on how we can use the features of WF to provide an authoring experience.

User Experience with Microsoft Office SharePoint Server

Using WF to provide the application logic and control of the process means that we can use any technology to host the application logic—from ASP.Net to Windows Forms to Windows Presentation Foundation (WPF). The screenshots in Figure 7 show the use of WPF and Microsoft

Figure 8: Computation scenario with Microsoft Office SharePoint Server

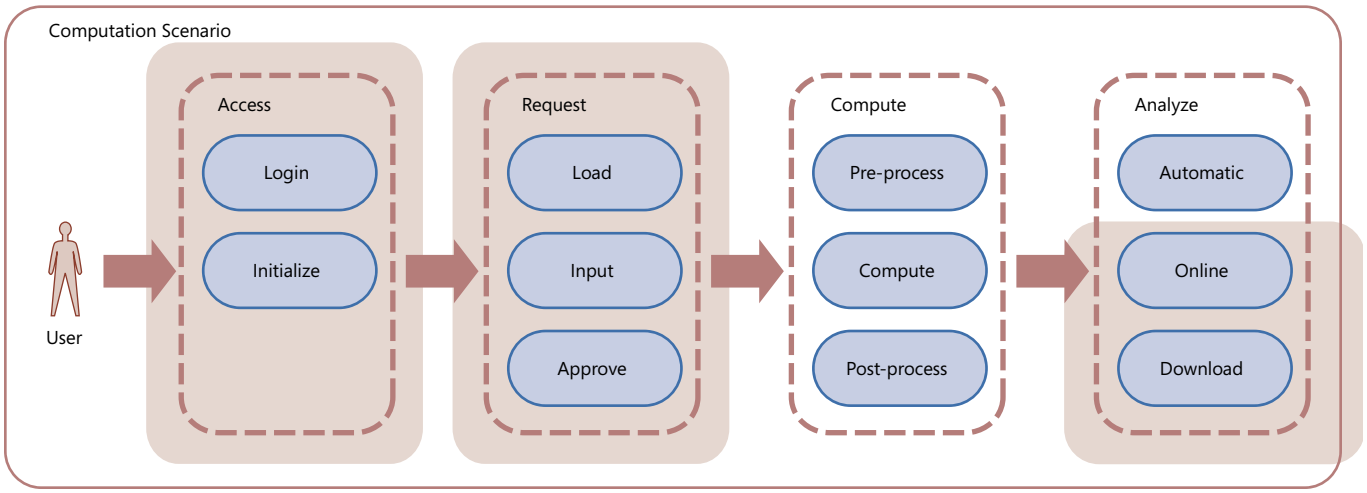
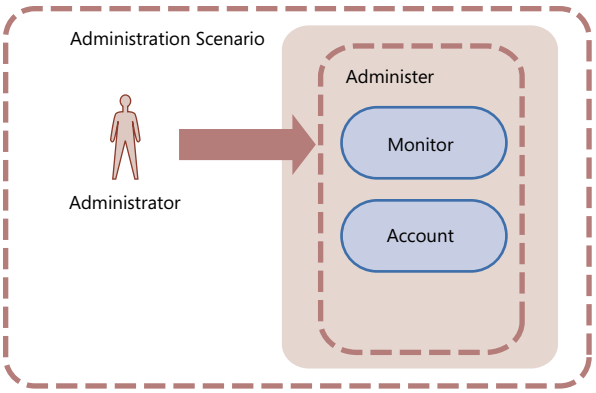


Figure 9: Microsoft Operations Manager can monitor the system health of the CCS cluster.



Office SharePoint Server (MOSS), respectively using the same WF-based application layer, though offering very different user experiences.

Use of Microsoft Office SharePoint Server (MOSS) provides a solution footprint covering areas in the Access, Request, and Analyze steps (Figure 8). The reasons are:

- *Collaboration.* Research and other typical uses of HPC revolve around collaboration activities, and MOSS is extremely feature-rich in terms of enabling collaboration around the concept of "sites."
- *Access.* Research groups may be geographically separated, so a Web-based interface helps ensure that the platform is available to all users. Significant business value could be generated from providing access to an HPC platform to remote users.
- *Extensibility.* MOSS can be extended in a variety of ways, from search and cataloging of information to the construction of Web parts to be "plugged in" to the main application. This extensibility also provides a strong user paradigm and development platform and framework for the overall solution.
- *Workflow.* MOSS is also powered by WF so the synergy between the two technologies is quite clear. An advantage here is that the skill sets of teams handling HPC workflow and those handling other business workflow are fully transferable.

To cover the required footprint, we can take advantage of several features of MOSS:

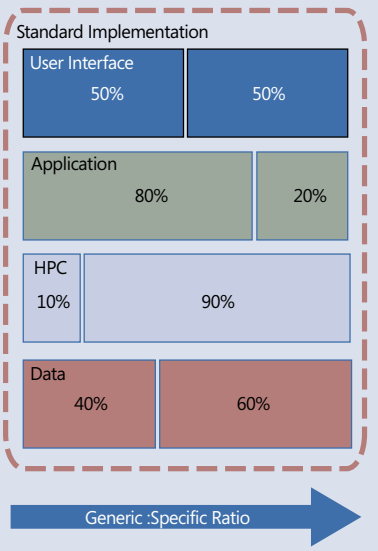
- *Access and initialize.* The single sign-on and user capabilities of MOSS will effectively provide off-the-shelf access functionality. Initialization can be made extremely easy through the creation of bespoke "sites" representing computation job types. A user can then generate a site representing a project or a job and receive the required configuration of Web parts in order to request, execute, and analyze on a computation job. Additionally, collaboration features such as presence controls and wiki can also be exploited.
- *Request and execution.* Specific Web forms could be designed to enable parameter input and requests to be made, but a better solution, in terms of capability and flexibility, may be to use InfoPath. InfoPath provides sophisticated Web form capabilities and a well-developed authoring capability. Because it uses XML to maintain the form definition and the data for the form, it is also well positioned to take advantage of Web services for submission, and the use of Web service enabled workflows to trigger job request approval and job execution. WF itself can be used to define the required approval workflows and is embedded as part of MOSS.
- *Analysis.* MOSS is also capable of hosting reporting services and Business Scorecard Manager (soon to be PerformancePoint) for dashboard-style reporting. Additionally, Excel Services can be accessed via MOSS, making it an ideal platform for performing the initial analysis.

A generic implementation?

Although aspects of the proposed architecture are intended for reuse, it is unlikely that the architecture can be significantly generic to simply be reused from scenario to scenario. That the architecture represents a "shell" or "ball-and-socket" approach is more likely: Development will be required in each scenario, but with various reusable components and approaches. A speculative ratio (based on development experience) for generic to specific components is shown in Figure 1.

These ratios could be explained as:

- *User interface 50:50.* Core aspects of the user interface—file movement controls, job execution controls and monitoring—would likely be immediately reusable and applicable to many scenarios. Specialization would be required—as with any UI—for any given process, particularly with such diverse scenarios as HPC computations.
- *Application layer 80:20.* The application layer is made up of a series of workflows and activities which can be rebound and reorganized for a particular process. With some work, it is likely that a lot of these activities and logical elements can be reused given new data. Examples may include activities for file movements, CCS integration, policy, and metascheduling. Also, various authoring tools would be applicable for power users in any scenario.
- *HPC layer 10:90.* Very little of the HPC layer has any reuse because every algorithm will be unique. However, monitoring and installation processes are reusable and match existing Microsoft paradigms.
- *Data layer 40:60.* Again, unique data structures will be required for any given computation, but tracking, persistence, and billing databases (for example) will all be applicable from scenario to scenario.



Monitoring with Microsoft Operations Manager and PowerShell

The main product for monitoring system health is Microsoft Operations Manager (MOM). We can use MOM to monitor the health of the CCS cluster in the same way as for other server environments. In addition, we can provide simple user monitoring for end users with PowerShell. (See Figure 9.)

Providing an Authoring Experience with Visual Studio 2005

Finally, we can provide a first-class authoring experience through Visual Studio 2005, some of which is off-the-shelf capability, and some extension (Figure 10).

Develop

Coding is required for authoring activities and algorithms, so Visual Studio 2005 is an ideal environment for using the .NET framework to extend the activities or amend the user interface, as well as for using the Microsoft MPI and C++ stacks for the coding of parallel algorithms.

For the development of workflows, Visual Studio 2005 is also the preferred option, although it may not be as suitable for power users as for developers, due to the complexity of the interface. For power users, we have other options:

- *Designer rehosting*. The entire workflow designer can be rehosted within a Windows Forms application. This application could have a series of specific, cut-down functions to ensure that a power user can easily construct and configure workflow templates. It can also be used to provide some additional quality assurance to the workflow constructs by limiting the activities available to a power user.
- *Designer development*. Alternatively, because workflow and rule definitions can be described in XAML, an entirely new design surface could be constructed to represent workflow construction. A new design surface is a great option for a domain that has a particular way of diagramming processes or where other hosting options are required, for example, making use of Windows Presentation Foundation or AJAX capabilities of the Web. The resources section contains a link to an ASP.NET AJAX implementation of the design surface.

Figure 10: Authoring scenario with Visual Studio 2005

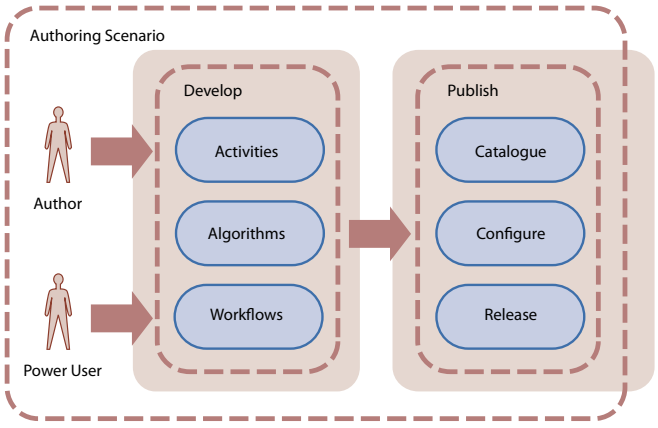


Figure 11: Overlapping roles reduce costs

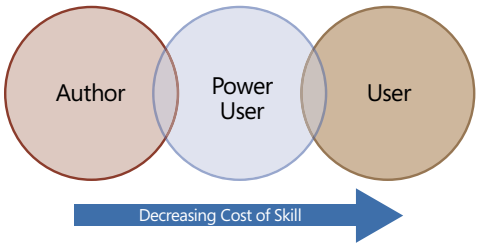
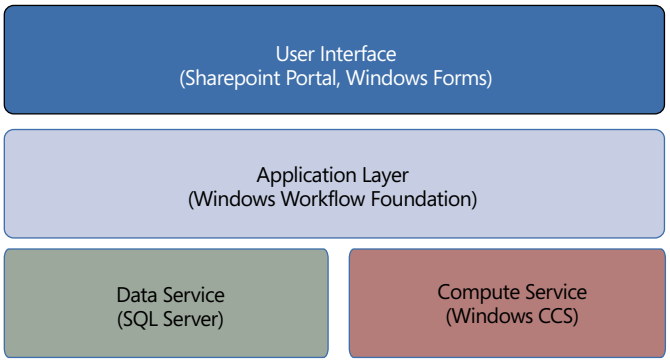


Figure 12: HPC technical architecture



From a development perspective, we can effectively consolidate all platform development into one environment and specialize certain aspects through the use of XAML for the definition of workflows. The use of XAML also presents opportunities for publishing.

Publish

In the case of activities and algorithms, the publishing paradigm is effectively a software release, as binaries need to be distributed accordingly, though because of the very specific software that is being generated, a distribution mechanism could be built to specifically retrieve the libraries as required.

The Microsoft ClickOnce technology may also be worth investigating for the distribution of a "thick client" power user authoring application for automating library updates.

In the case of workflow authoring, publishing could be as simple as transmitting the resulting XAML definitions to a central database and cataloging these definitions. The workflow can be retrieved from the same database by an end user—or simply invoked as the result of a selection on a user interface—and then compiled and executed as required. Described in the next section, implementing this concept means new workflows could potentially be published very quickly because no code release is required (therefore, there are no release cycle or administrative requirements).

This capability could be provided as part of a bespoke power-user application, or as an add-in to Visual Studio 2005.

Workflow Compilation

WF workflows are typically compiled into assemblies, and are then executed by the workflow runtime by passing a type that conforms

to a workflow into the runtime to create an instance of the workflow. However, a workflow can be executed from a declarative markup file—XAML file—if required, or compiled on the fly from XAML, with rules alongside.

This is useful because it allows a workflow author—someone with knowledge of the coding of workflows—to create a workflow that is not entirely valid—perhaps it is missing some parameters—and save the workflow into a markup file. The markup file can then be amended by a power user to add the required parameters (but perhaps not edit the actual workflow). Now that the workflow is valid, it can be submitted, compiled and executed by the application layer.

These steps can reduce the cost of process execution by enabling roles to overlap (Figure 11); expensive authoring skills do not have to be involved at the execution step of the process where less expensive configuration skills can be applied.

Technical Architecture

The solution scenarios can be aggregated into a technical architecture (Figure 12). Conceptually, CCS should be considered a service gateway in the same way that SQL Server provides services to an application—the solution is not centered on CCS but utilizes its capabilities as required.

Similarly, these services are typically abstracted away from a user interface through some sort of application control or business logic layer. Given the nature of interactions with the HPC solution—human workflows—then WF is a suitable choice as a controller for these services.

WF also has other advantages. It is designed to be hosted as required, so the user interface layer could be a Web or Windows application written specifically for the task. In the instance described above, we described Microsoft Office SharePoint Server as a suitable interface because it has strong hooks for WF and the integration with Office (including InfoPath) and the collaboration features of MOSS could be useful in the scientific and engineering domains. The choice, of course, should be based on the requirements of a given domain.

Conclusion

Architecting for high productivity computing is not simply a case of ensuring the "best" performance in order to compute results as quickly as possible—that is more of an expectation than a design feature. In the context of the overall value stream, the architecture must drive value from other areas, such as ease of access and decreasing cost of specialist skills to operate the system.

A successful architecture for high productivity computing solutions involves consideration of the overall process alongside the computationally intensive activities, and therefore, may use several integrated components to perform the individual aspects of the process. Microsoft Cluster Compute Server Edition is easy to include as a service gateway inside a general n-tier application structure and is simple to integrate via command-line or API hooks.

Other available technologies can provide the basis for an HPC solution. In particular, Windows Workflow Foundation is well-suited to providing an application interface to CCS because the features and extensibility of WF, such as persistence and tracking, lend themselves to the requirements of HPC-based solutions. The use of WF also opens up the available choices of user experience technologies to be applied in a given domain.

Further Resources

The following resources may be of help when considering the technologies comprising the solution proposed in this document:

- Microsoft Compute Cluster Server 2003, http://www.microsoft.com/hpc
- Microsoft SQL Server 2005 (including SQL Server Integration Services), http://www.microsoft.com/sql
- Microsoft Windows Workflow Foundation, http://wf.netfx.com
- Microsoft Windows Communication Foundation, http://wcf.netfx3.com
- Microsoft Windows Presentation Foundation, http://wpf.netfx3.com/
- Microsoft Office SharePoint Server 2007, http://www.microsoft.com/sharepoint/
- Microsoft Operations Manager, http://www.microsoft.com/mom
- Microsoft Windows PowerShell, http://www.microsoft.com/powershell

Other resources referred to in the document:

- CCS 2003 Technical Articles, http://technet2.microsoft.com/WindowsServer/en/library/9330fdf8-c680-425f-8583-c46ee77306981033.mspx?mfr=true
- Essential Windows Workflow Foundation, http://www.awprofessional.com/bookstore/product.asp?isbn=0321399838&rl=1
- Implementing the WF design surface in ASP.NET (with AJAX), http://www.netfxlive.com/
- Whitepaper on WF Performance, http://msdn2.microsoft.com/en-us/library/Aa973808.aspx

Acknowledgments

Peter Williams (Microsoft Consulting Services, Reading)

About the Authors

Marc Holmes is an Architect Evangelist for the Microsoft Technology Centre at Thames Valley Park in the U.K. where he specializes in architecture, design, and proof of concept work with a variety of customers, partners and ISVs. Prior to Microsoft, Marc most recently led a significant development team as Head of Applications and Web at BBC Worldwide. Marc is the author of "Expert .Net Delivery with NAnt and CruiseControl.Net" (APress) and maintains a blog at http://www.marcmywords.org. He can be contacted at marc.holmes@microsoft.com.

Simon Cox is Professor of Computational Methods in the Computational Engineering Design Research Group within the School of Engineering Sciences of the University of Southampton. An MVP Award holder for Windows Server System - Infrastructure Architect, he directs the Microsoft Institute for High Performance Computing at the University of Southampton and has published over 100 papers. He currently heads a team that applies and develops computing in collaborative interdisciplinary computational science and engineering projects, such as computational electromagnetics, liquid crystals, earth system modeling, biomolecular databasing, applied computational algorithms, and distributed service-oriented computing

Test Driven Infrastructures

by Mario Cardinal

Summary

IT shops must fulfill two roles: "build" and "run" software. Each role requires a different set of skills. The gap between "build" and "run" is almost always clearly visible in the organization chart. At the architecture level, on one side, there are the application architects involved in software development (build), and, on the other side, the infrastructure architects involved in software operation (run). Being an application architect, I believe that both teams should learn from each other's best practices. One best practice that the infrastructure team should learn from the software development team is to express architecture decisions using test scripts.

Architecture decisions

The software architecture discipline is centered on the idea of reducing complexity through abstraction and separation of concerns. The architect is the person responsible for identifying the structure of significant components, which are usually thought of as hard to change, and for simplifying the relationships between those components. The architect reduces complexity by dividing the problem space into a set of components and interfaces that will be more and more difficult to change as the project evolves.

The only way to simplify software is to structure all the main component interactions through interfaces and accept the fact that these interfaces are now almost impossible to change. If you pick any one component of software, then you can make it easy to change. However, the challenge is that it is almost impossible to make everything easy to change without increasing the level of complexity, as Ralph Johnson said in a paper that Martin Fowler wrote for IEEE Software: "Making something easy to change makes the overall system a little more complex, and making everything easy to change makes the entire system very complex. Complexity is what makes software hard to change."

Establishing overall component interactions through interfaces is making irreversible design decisions. This set of design decisions about logical and physical structure of a system, if made incorrectly, may cause your project to be cancelled. The key to success is to protect the system against instability. Good architects know how to identify the areas of change in existing requirements and protect them against the architecture's irreversibility.

Documenting architecture is explicitly communicating without ambiguity the set of irreversible design decisions.

Nonambiguous documentation

Documenting architecture decisions facilitates communication between stakeholders. They are the persons that have a legitimate interest in the solution. There are two classes of stakeholders with different needs in regard to architecture specifications. The first class of stakeholders is the deciders who need to know about the architecture in order to understand the constraints and limitations of the solution. Examples of such stakeholders are managers, customers, and users. The second class of stakeholders is the implementers who need to know about those same decisions in order to build and run the solution. Examples of such stakeholders are developers, system engineers, and system administrators.

A narrative specification written as a document is the perfect documentation for the first class of stakeholders. For them, even a nonformal specification published as a Microsoft Powerpoint is good enough. It provides a high-level view of the architecture with almost no ambiguity in regards to the constraints and limitations of the solution.

However, a narrative specification is not the appropriate documentation for the second class of stakeholders. It does not provide concise information about design decisions to implementers. And, if it does so in a very thick formal specification, this document will be of no interest for the deciders. That first class does not care at all about the inner structure of significant architecture components. However, implementers do.

In my experience as an application architect, I have discovered that the easiest way to communicate all the intricacy of an irreversible design decision is not to write a narrative document but instead to write a test that explains how I would validate a good implementation. When you use only words, it is very difficult for the architect and the implementers to understand each other with confidence. There is always a mismatch in how each party understands the meaning of a specific word.

Here is a simple example to demonstrate my point. Let's say that I wrote, in the architecture specification, the following design decision:

"We will restore data only from the previous day in case of hard disk crash or data corruption. Users will need to retype lost data for the current day."

Obviously, implementers need clarification about my intent in order to implement correctly my specification. Here is the same decision written for them:

"We will backup the SQL Server database every night. In case of recovery, we will restore the latest backup".

This is better than the original one but there are still a lot of grey zones. I could rewrite the description over and over and provide more and more details for the implementers. I will end up with a formal detailed specification. However, a narrative documentation does not provide an explicit consensus about what a successful implementation is. Even with the best intent, usually the implementers misunderstand subtleties of my formal specification. For example, let's say that the solution that I was envisioning was to back up on tape and I forgot to make this explicit. What happens if the implementers back up the SQL Server database on disks? Unless I discovered it during design review, I won't know about it until both the SQL Server disk and the backup disk crash the same day.

Instead, if I wrote a test script, I would establish an explicit consensus about the compliance. A test script either succeeds or fails. Tests enable the implementers and the architects to explicitly define the compliance against the architecture specification. Implementers looking to learn about the architecture can look at the test scripts. Test scripts are operational artifacts. They are less susceptible to drifting from the implementation and thus becoming outdated.

Explicit consensus

A test script is the combination of a test procedure and test data. Test scripts are written sets of steps that should be performed manually or automatically. They embody characteristics that are critical to the success of the architecture. These characteristics can indicate appropriate or inappropriate use of architecture as well as negative behaviors that are to be trapped.

As a concrete example, Figure 1 shows two simple test scripts that validate the following irreversible design decision:
"We will restore data only from the previous day in case of hard disk crash or data corruption. Users will need to retype lost data for the current day."
A test script either succeeds or fails. Because it validates the intent of the architect, it provides an explicit consensus about compliance.

Figure 1: Sample test scripts for validating design decisions.

Test script 1
Intent: Recover from SQL Server hard disk crash
Step:
1. Replace defective hard disk
2. Install SQL Server database
3. Restore yesterday's backup from tape (create schema and data)

Success criteria: I can find at least one order with its update field set to yesterday in table "order"

Test script 2
Intent: Recover from data corruption
Step:
1. In an existing SQL Server database, restore the last day's backup from tape (overwrite data)

Success criteria: I can find no orders with the update field set to today and at least one order with its update field set to yesterday in table "order"

Protecting against change

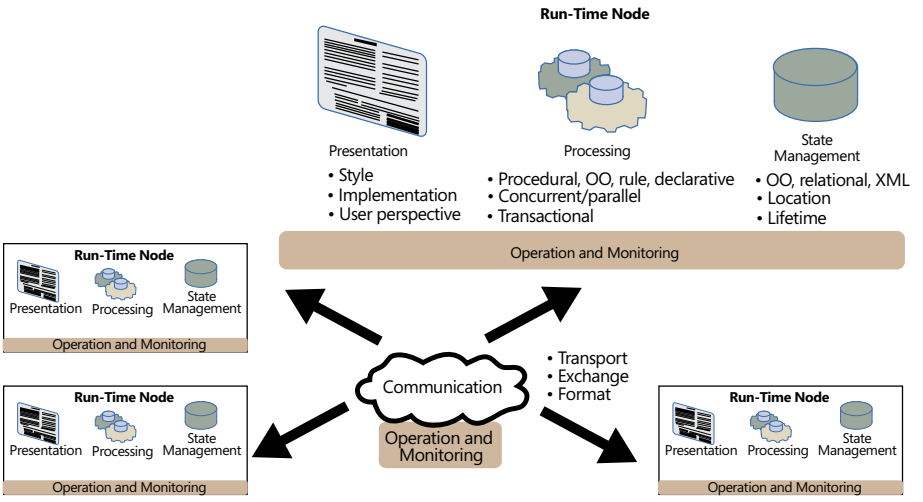
Test scripts expressing the key architecture decisions are always related to things that change. Main areas of change can be divided in three categories:

1. **Execution:** In this category, variations about operations and monitoring are the main concern for infrastructure architects. They crosscut all the other execution concerns such as presentation, processing, communications, and state management:

- *Presentation:* Stakeholders interacting with the system. How do we implement the UI? Do we need many front ends? Are quality factors such as usability, composability, and simplicity important?
- *Processing:* Instructions that change the state of the system. How do we implement processing? Do we need transactional support? How about concurrency? Are quality factors such as availability, scalability, performance, and reliability important?
- *Communication:* State distribution between physical nodes of the system. How do we interact? What format does the data travel in? Over what medium is the communication done? Are quality factors such as security and manageability important?
- *State management:* Location, lifetime, and shape of data. Do we need a transient or durable state? How do we save the state?
  Are quality factors such as availability, recoverability, integrity, and extensibility important?

Figure 2 is a diagram showing all the potential sources of run-time variations.

Figure 2: Potential sources of run-time variations



Architects protect the system against run-time instability by building abstractions. Tests should validate these abstractions. For example, in high availability requirements, to protect against defects of a physical server, a proven solution is to put a failover in place. Failover is the capability of switching over automatically to a redundant or standby computer server upon the failure or abnormal termination of the previously active server.

2. **Deployment:** These are the load-time concerns. In this category, infrastructure architects should pay attention to configuration and registry variations. For example, to protect against change in authentication mechanisms when using Windows as a platform, a proven solution is to store user credentials in Microsoft Windows Active Directory.
3. **Implementation:** The most important design-time variations are related to instability in tooling and organizations (people). For example, infrastructure teams have learned how to protect against unstable development teams. Deployment of bad components in production can be disastrous for service level agreements. A proven solution is to establish a fallback mechanism to rapidly restore the system to a previously stable state. Not only should we document fallback processes but we should also write tests to express compliance without ambiguity.

The process of discovering irreversible design decisions consists of reviewing all of these areas susceptible to change and building the appropriate tests.

Operational artifacts

Today, test scripts can be manual, automated, or a combination of both. The advantage of automated testing over manual testing is that it is easily repeatable. It is therefore favored when doing regression testing. After modifying the software, either for a change in functionality or to fix defects, a regression test reruns previously passing tests. It ensures that the modifications haven't unintentionally caused a nonrespect of design decisions.

An automated test script is a short program written in a programming language. Infrastructure architects do not need to be programmers to write automated test. Scripting languages can efficiently do the job. When using Windows as a platform, PowerShell, Microsoft's new scripting language and command shell, is well-suited for such purposes given its support for control structures, exception handling, and access to .NET system classes. Here are some of the testing scenarios PowerShell could be used for:

- *Deployment testing.* Create a script that verifies your release went as expected; check running processes, services, database metadata, database content, application file versions, configuration file contents, and so on.
- *Infrastructure testing.* Create a script that verifies the hardware, operating system, running services, running processes, and so forth.

For example, here is a PowerShell small function to test connectivity to a server:

```
function test-connection
{
        $pingtest = ping $args[0] -n 1
        if ($pingtest -match 'TTL=')
        {
                write-host $true
        }
        else
        {
                write-host $false
        }
}
Usage:

test-connection MyServer01.domain.com
```

Contrary to a narrative document, a set of automated test scripts is an operational artifact. It is always kept up to date and it continuously validates compliance with the intent of the architect.

Design for testability

Testability means having reliable and convenient interfaces to drive the execution and verification of test scripts. You cannot achieve testability if you write tests after the design and implementation. Building tests during design is the only viable approach to achieve testability.

By abstracting the implementation, writing tests first greatly reduces the coupling between components. The irreversible design decisions can now be more thoroughly tested than ever before, resulting in a higher quality architecture that is also more maintainable. In this manner, the benefits themselves begin returning dividends back to the architect, creating a seemingly perpetual upward cycle in quality. The act of writing tests during the design phase provides a dual benefit:

1. It validates compliance with the intent of the architect.
2. It does so explicitly.

Testing is a specification process, not only a validating process.

Design for automation

Experience has shown that during software maintenance, reemergence of faults is quite common. Sometimes, a fix to a problem will be "fragile"—that is, it does not respect characteristics that are critical to the success of the architecture.

Therefore, it is considered good practice that a test be recorded and regularly rerun after subsequent changes to the program. Although this may be done through manual testing, it is often done using automated testing tools. Such tools seek to uncover regression bugs. Those bugs occur whenever software functionality that previously worked as desired stops working or no longer works in the same. Typically, regression bugs occur as an unintended consequence of program changes.

Regression testing is an integral part of the agile software development methodology, such as eXtreme Programming. This methodology promotes testing automation to build better software, faster. It requires that an automated unit test, defining requirements

Figure 3: PowerShell automated test

```
set-psdebug -strict -trace 0

# Ensure that group "App_Setup" is defined in Windows Active Directory
function TestActiveDirectoryGroup
{
   $objGroup =[ADSI]"LDAP://localhost:389/CN=App_Setup,OU=HR,dc=NA,dc=org1,dc=com"
   AssertNotNull $objGroup "App_Setup group is required to install application xxx."

      RaiseAssertions
}

# run the function library that contains the PowerShell Testing Functions
# the functions are defined as global so you don't need to use dot sourcing
if (!(Test-Path variable:_TESTLIB)) { ..\src\TestLib.ps1 }

# run the function defined above that demonstrates the PowerShell Testing Functions
TestActiveDirectoryGroup
```

of the source code, be written before each aspect of the code itself. Unit tests are used to exercise other source code by directly calling routines, passing appropriate parameters, and then, if you include `Assert` statements, testing the values that are produced against expected values. Unit testing frameworks, such as xUnit, help automate testing at every stage in the development cycle.

The xUnit framework was introduced as a core concept of eXtreme Programming in 1998. It introduced an efficient mechanism to help developers add structured, efficient, automated unit testing into their normal development activities. Since then, this framework has evolved into the de facto standard for automated unit testing frameworks.

The xUnit frameworks execute all the test scripts at specified intervals and report any regressions. Common strategies are to run such a system after every successful build (continuous integration), every night, or once a week. They simplify the process of unit testing and regression testing.

It is generally possible to perform testing without the support of xUnit frameworks by writing client code that tests the units and uses assertions, exceptions, or early exit mechanisms to signal failure. However, infrastructure architects should not need to write their own testing framework. Familiar xUnit testing frameworks have been developed for a wide variety of languages including scripting language commonly used by system administrator.

On the Windows platform, "PowerShell Scripts for Testing," a function library that implements the familiar xUnit-style asserts for Microsoft's new scripting language and command shell, can be downloaded for free from CodePlex, a Web site hosting open source projects (see Resources). A sample Powershell automated test is shown in Figure 3.

Automated tests seek to discover, as early as possible, non-respect of design decisions that are critical to the success of the architecture. Automated tests are:

- Structured
- Self-documenting
- Automatic and repeatable
- Based on known data
- Designed to test positive and negative actions
- Ideal for testing implementation across different machines
- Operational documentation of configuration, implementation, and execution.

Data-driven testing

Test automation, especially at the higher test levels such as architecture testing, may seem costly. The economic argument has to be there to support the effort. The economic model can be weakened if you do not simplify the process of writing tests.

Data-driven testing is a way of lowering the cost of automating tests. It permits the tester to focus on providing examples in the form of combinations of test input and expected output. You design your test by creating a table containing all the test data (inputs and expected outputs) and then write a test fixture that translates a row in the table into a call to the system under test (SUT).

You achieve a high performance ratio by virtue of never having to write the script beyond writing the test fixture. The framework implicitly provides the script—and runs it.

In the latest version of "PowerShell Scripts for Testing," this xUnit framework makes it possible to use Excel as the source of the data to support data-driven testing. There is a new function in DataLib.ps1 called `start-test` that takes a workbook name, a worksheet name, a range name, and a set of field names as input, and runs the tests that are described in that range. It does this by calling a function that has the same name as the range.

Using `start-test` is demonstrated in a sample performance test on the companion blog for PSExpect. Figure 4 shows the Excel table and the test script provided for the weather Web service.

Figure 4: Excel table and test script for sample weather Web service

TestCase	Remark	Zip	Units	Title
TC-1	Success in Celsius	35801	C	Conditions for Huntsville, AL
TC-2	Success in Fahrenheit	35801	F	Conditions for Huntsville, AL
TC-3	Success - Bad Units	35801	Z	Conditions for Huntsville, AL
TC-4	Success - Default Units	35801	$null	Conditions for Huntsville, AL
TC-5	Failure - No Zip	$null	F	City not found
TC-6	Failure - Invalid Zip	2	F	City not found

```
# Exercises the target of the test - the get-weather web service
# and then verifies the results against the expected values that are also on the
# worksheet (in this simple case, the $title parameter)
function TestGetWeather()
{
    param(  [string]$TestCase, `
            [string]$Remark, `
            [string]$Zip, `
            [string]$Units, `
            [string]$Title)

    if ($Units -eq "$null") { $Units = $null }
    if ($Zip -eq "$null") { $Zip = $null }

    [string]$urlbase="http://xml.weather.yahoo.com/forecastrss"
    [string]$url=$urlbase + "?p="+$Zip+"&u="+$Units

    write-host Connecting to $url

    # create .NET Webclient object and call the web service
    $webclient = new-object "System.Net.WebClient"
    $targetBlock = {[xml]$weather=$webclient.DownloadString($url)}

    # measure the results and verify
    $targetBlock | AssertFaster -MaximumTime 300 -Label $TestCase
}

# run the function library that contains the PowerShell Testing Functions
# the functions are defined as global so you don't need to use dot sourcing
if (!(Test-Path variable:_XLLIB)) { ..\src\DataLib.ps1 }
if (!(Test-Path variable:_TESTLIB)) { ..\src\TestLib.ps1 }

# run the test by calling DataLib\start-test function
$FieldNames = ("TestCase","Remark","Zip","Units","Title")
start-test ((get-location).ToString() + "\TestGetWeather.xls") "Sheet1" `
    "TestGetWeather" $FieldNames
```

Conclusion

Automated testing combined with regression testing not only documents the architecture decisions at the appropriate level for implementers but also continuously validate compliance with the intent of the architect.

The benefits of writing automated tests are the following:

- *Enable freedom to change:* As software systems get bigger, it becomes harder and harder to make changes without breaking things. The business risk to this situation is that you may find yourself in a situation where customers are asking for things or the market shifts in some way that causes the need for change. A large battery of automated test scripts frees the architects to do lots of cool things. Architects will be less afraid to change existing design decisions. With the automated testing as a "safety net," they can refactor the architecture, or add or change features, for example, without losing sleep. What you will find, by investing in automated testing is that your organization actually moves faster than it did before. You can respond to market change quicker, you roll out features faster, and you have a stronger organization. Automated testing helps you:
- *Reduce costs:* Automated testing finds problems effectively as early as possible, long before the software reaches a customer. The earlier problems are found, the cheaper it is to fix them because the "surface area" of change is smaller (that is, the number of changes since the last test will be limited).
- *Ensure reliability:* Tests perform precisely the same operations each time they are run without ambiguity. It either succeeds or fails. It provides an explicit consensus about compliance and continuously validates the intent of the architect. It eliminates human error and negates the fatigue factor of manual testing as deadlines approach.
- *Engender confidence:* Automation provides concrete feedback about system because you can test how the software reacts under repeated execution of the same operations.
- *Prevent risks:* Automated testing demonstrates system correctness within a sensible deadline. Tests can be run over and over again with less overhead.
- *Improve maintainability:* Writing tests first influences structure and creates modular systems. It greatly loosens the coupling between components, thus reducing the risk that a change in one component will force a change in another component.
- *Provide up-to-date documentation consistently:* Tests are an operational artifact that cannot be left out of sync. If tests drift from reality, it is impossible to run them with success. Outdated tests will always fail.

Resources

"Who needs an architect?" Martin Fowler, IEEE Software Magazine, Volume 20, Issue 5 (September 2003)
http://www.martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf

CodePlex PowerShell Scripts for Testing
http://www.codeplex.com/psexpect

Companion blog for PSExpect:
http://testfirst.spaces.live.com

About the Author

Mario Cardinal is an independent senior consultant specialized in enterprise application architecture. He spends most of his time building well-designed enterprise .NET applications with agile processes. He possesses over fifteen years of experience in designing large-scale information systems. For the second year in a row, he has received from Microsoft the Most Valuable Professional (MVP) award in the competency of Software Architect. MVP status is awarded to credible technology experts who are among the very best community members willing to share their experience to helping others realize their potential. He leads the architecture interest group at the Montreal Visual Studio User Group and the Montreal's chapter of the International Association of Software Architects (IASA). He is also the architecture track tech chair for DevTeach Conference. Furthermore, he hosts an audio Internet talk show about software development with Microsoft .NET (Visual Studio Talk Show). Mario holds Bachelor of Computer Engineering and Master of Technology Management degrees from the Ecole Polytechnique in Montreal, Canada. He also holds the titles of Certified ScrumMaster (CSM) and Microsoft Certified Solution Developer (MCSD.Net). (Contact Mario through his Web site, www.mariocardinal.com.)

Architecture Journal Profile: Don Ferguson

For this issue, as part of the Architecture Journal Profile series, we had the chance to catch up with Don Ferguson, a Technical Fellow at Microsoft. We asked Don some questions about his career and what advice he had for people who wanted to become architects or who are interested in architecture today.

AJ: Who are you, and where are you from?

DF: I'm Don Ferguson. Before I joined Microsoft earlier this year, I was an IBM Fellow. There are about 200,000 engineers at IBM and about 50 Fellows, so it was quite an honor to be one of them. When I first joined IBM Research I never thought that someday I would be a Fellow. Maybe one day I would become a project leader—that would be quite an accomplishment. There are a lot of smart, talented people, and I just hoped that I could be one of them.

AJ: How did it feel to become an IBM Fellow?

DF: The day I became a Fellow was one of the best days of my life. In truth, I did feel a little bit awkward about the award, however. I felt that there were some people that deserved it more than I did. So shortly after the announcement I became an advocate for these people—and over the next few years was able to help get three or four of them to Fellow. In some ways, I was happier on their award days than on mine. When it happens to you, it's a blur, but when it happens to others you have a lot of satisfaction.

AJ: What did a typical day look like?

DF: I was the Chief Architect in the Software Group in IBM. There were hundreds of products and projects. Being the Chief Architect means that you can't design every component. There are just too many products and products for this to be a realistic goal. A large portion of my day would be spent doing a lot of review work. I would review several projects' or products' designs, focusing on some top-level things. I placed special emphasis on new products. Typically, I would focus on cross-product integration, common themes, etc. For example, does the portal product have the right interface to connect to this security product? Is the product supporting the right standards? I used to jokingly refer to this as "herding the cats." On the title slide of presentations that I gave, I often used the title of "Chief Cat Herder."

In the early days, this work was very unsatisfying—too abstract, too shallow, too broad. But then I had an epiphany. Working on pulling the different parts together a little bit more had a significant effect, especially across products. A little bit of improvement in a lot of places makes a difference. An example is also true of the work that I was able to do on the Web Services standards. I think I helped make them a little more coherent, composable, easier to understand, etc.

AJ: With such responsibilities, how do you stay technically savvy?

DF: Good question! The first thing I do is work really hard. When I'm not with the kids, it's often eat, sleep, work, karate—and sometimes sleep is optional. I definitely put in a lot of hours. Secondly, I do a lot of work on airplanes where there are no conference calls or disruptions. Whenever possible, I consciously stay away from email. I also try not to learn new technical information by reading a paper or presentation. The best way is to is to download, install, use and code.

Over the years, one of the things I've learned to do well is to synthesize how things should work with only a little bit of information. By and large, people are smart. With a little bit of information, I can often figure out what smart people must have done. How would smart people have designed this? With this process, although I may not know the topic in depth, I can often work out how things must work and contribute to the discussion.

In IBM, I tended to have around 10-12 initiatives that I was working on at any time. I would spend a few months each project, multiplexing between the initiatives. Eventually, each initiative would either take on a life of its own or not pan out. You'll see a lot of this in the standards work I participated in. I was listed as contributor for many of the early papers, but over time I disappeared. A lot of good people started driving and doing deep technical work, and I moved on to something else.

AJ: What advice would you give someone who wants to become an architect today?

DF: I think there are four pieces to this answer.

Firstly, at IBM we had an architect board in the Software Group, which helped me form a network. It took me a while to understand the importance of a network. Being a New Englander, I tend to be a little taciturn and by myself. You should never underestimate the importance of a social network. You don't know what you don't know. You don't know what someone may say to you that can push the reset button in your brain and make you think differently.

Secondly, as a software architect, never stop coding. I write code. It's not good or particularly deep code, but I do code. A lot of it is educational and related to my spare time activities. For example, recently I've been working on a portal that connects my family using TikiWiki and PHP. I installed the products, but they didn't work for me right out of the box.

So, I had to go in and hack them. It was cool. Another example is the nursery school that my daughter attends. They asked me to set up a Web site using FrontPage, which was another learning experience.

Thirdly, communication skills matter. They really do. It's really important to understand how to write well and how to present well.

Finally, the most important thing is to connect with customers. Spend as much time as possible with them. Learn what they are trying to do, and look at what works and what doesn't. Help them use your products. There's no substitute for spending time with customers and helping them solve problems. Doing this, we often learned that customers used things in ways that we never dreamed they would. We also came up with amazing new ideas.

AJ: Who is the most important person you've ever met and what made that person so significant?

DF: When I started out in academia and the Research Division, my thesis adviser (Yechiam Yemini) was really important. He taught me two things:

People who are driven tend to underestimate the quality of what we do. We are too hard on ourselves. We think our papers aren't very good or our research isn't novel. My adviser helped with perspective.

Secondly, he taught me that this stuff can be fun—just have a good time, and enjoy what you are doing. He had a very contagious, childlike enthusiasm.

In addition to my adviser, there are many more people at IBM who became great friends and role models. Tony Storey was the most important one. I would look at the ones that I respected, and often try to "reverse engineer" how they do their job. I would consciously try to be like them.

AJ: What is the one thing in your career that you regret? What did you learn from it?

DF: Wow! I don't even know where to begin! I'll give you two: My old group gave me a "Darth Vader" helmet, because of the way I was operating in the early days. "Do it my way or I will blow your planet up." The "Darth Vader" approach doesn't work. You can't make smart people do things that they don't want to do. People do well what they want to do. For me that was an epiphany. Why was I so curt and domineering? I was busy, and all I could think about was "I have these 20 things to do—why won't this person just do what I want?" I came to realize that this approach never works, no matter how hard you try. People deserved my time and I owed them the time.

Secondly, many people who work in technology suffer from the "end game fallacy." We are all pretty bright. We see a lot of customers. We see what they are doing and then plot a trajectory for where they will be in a few years. Once you do this however, it's too tempting to build what they will need in five years, and not what they need next or are ready for. Sometimes I say there is no point in building the Emerald City without building the Yellow Brick Road—and you have to build the road first. Often I would make things too complicated, anticipating their future needs. One of the senior executives used to say that "vision without execution is hallucination," and it's true.

AJ: What does Don's future look like?

DF: To answer this, I have another piece of advice. When I ask myself that question, I always put personal fulfillment first and then job fulfillment second. If you are not fulfilled personally, you will never be fulfilled in your job. I have a black belt in Karate (Kenpo) and want to get better at that. I want to take another discipline—Jujitsu. I would like to teach Kenpo. Personal fulfillment has to come first.

For job fulfillment and what excites me, I imagine a point in the future where everyone can program. This isn't so much about using Fortran and C#, but instead another definition of "program." It sounds strange, but let me explain:

Everyone who graduates from high school has done some programming—even if it's something simple like a PHP Web site. These are basic skills now—just like I learned how to do long division by hand (although I would argue that students of today can't do long division anymore). These casual programmers will enter the workforce.

Maybe we need to broaden our definition of "program." What's the number one "programming tool" for business professionals? PowerPoint. If business professionals can use PowerPoint, I wonder whether we can nudge them into another tool to do business modeling. They already do business modeling, but they do it using documents that get handed to programmers. Programmers guess what the documents mean, and bad things happen when programmers guess. I wonder whether we can do something cleverer. In business schools, they teach a discipline called structured English and various diagramming techniques. Maybe these could form the basis for programming business services.

Job fulfillment for me would be thinking about how we change the fundamental nature of the Web. Web 2.0, mashups, and feeds are interesting, but they are step one of a two-step process. The Web today is a push model. People write content that is pushed out to people who read it. Programmers write Web applications that end users "use."

People write mashups or scripts to access the "pushed" content. These applications run on the PC today, but I believe that they could migrate into the Internet "cloud." The Internet now becomes the programmable Internet, and can run my applications for me. The tag line is "The Internet is the computer." We see this in nascent places already with Google, Amazon, MSN, etc. All now having callable services. Together with a broad range of programming skills for everyone, I see a blurring between the personal and business environment. That's what I really care about.

Dr. Donald Ferguson
Microsoft

Dr. Donald Ferguson is a Microsoft Technical Fellow in Platforms and Strategy in the Office of the CTO. Don focuses on both the evolutionary and revolutionary role of information technology in business. Microsoft expects that Don will be involved in a variety of forward-looking projects.
Prior to joining Microsoft, Don was an IBM Fellow and Chief Architect for IBM's Software Group (SWG). Don provided overall technical leadership for WebSphere, Tivoli, DB2, Rational and Lotus products. He also chaired the SWG Architecture Board (SWG AB). The SWG AB focused on product integration, cross-product initiatives and emerging technology. Some of the public focus areas were Web services, patterns, Web 2.0 and business driven development. Don guided IBM's strategy and architecture for SOA and Web services, and co-authored many of the initial Web service specifications.

Conquering the Integration Dilemma

by Jim Wilt

Summary

Extensible framework-based packages. They are everywhere. From portals to e-commerce. From content management to messaging. Effective? In many cases, yes, absolutely. I can think of many successful applications I've built based on the frameworks these products provide. They boost productivity, enhance quality, increase feature richness, and reduce the time to market greatly.

So, why don't integration solutions experience the same improvements? Regardless of how I plug at an integration solution with a given tool or framework, I don't seem to progress at the pace I experienced with my web application or portal solution. This is what I define as *The Integration Dilemma*.

Integration in and of itself, by definition, is a difficult problem to solve. We will examine the contributing factors so that they can be:

- recognized and categorized
- understood with their resulting repercussions
- proactively addressed.

The 90/10 Rule

Ninety percent of the activity centered on integration solutions is in setting up, configuring, and fine tuning the infrastructure environment and security properly. Operating Systems, Web Servers, Directories, and Application Settings, as well as Application Pools, Host Process users, Single Sign On, Directory Permissions, Security, and so on—these all play a part in the execution and contribute to the frustration often associated with integration implementation.

Many application and extensible package developers are protected from having to worry about strong name keys, the GAC, certificates, encryption, in-process hosts vs. isolated host processes, and the many other factors in which an integration developer must gain a great deal of expertise and mastery. The benefit is that integration development experience will greatly enhance a developer's approach moving forward with future [normal] application solutions. That is to say, a developer's skills are strengthened by completing an integration solution.

The remaining 10 percent in the 90/10 rule is left for the integration developer to solve the actual integration problem itself. This is the cause of much frustration to both development teams and management alike as the 90 percent commitment generally is not in any way applicable to the actual integration problem at hand. It is further magnified, as we will see in the next sections, as the actual integration problem is generally more difficult than anticipated requiring far more time to solve than the remaining 10 percent allows.

Integrating in frameworks like Microsoft BizTalk Server is much like going bowling with bumpers in the gutters. Their interfaces often keep you from doing too much harm to your solution, steering you with feature rich design and implementation interfaces. In contrast to the infrastructure and security challenges, this tool will actually help direct the solution. This positive integration experience, however, constitutes only 10 percent of the effort put forth.

Conquering the 90/10 Rule

Your infrastructure and operations teams are your new best friends:

- Keep a close working relationship with your infrastructure and operations teams from the project's onset as they may preemptively identify potential security and operational issues
- Many of the stifling issues an integration developer encounters are commonplace for an infrastructure resource, so utilize their experience to expedite problem identification and resolution.

> "INTEGRATION DEVELOPMENT GENERALLY HAS ZERO TOLERANCE: YOU ITERATE UNTIL IT IS PERFECT. PARTIAL FUNCTIONALITY AND PHASES ARE USUALLY NOT AN OPTION. THUS, INTEGRATION BECOMES THE SOURCE FOR MANY PROJECT DELAYS AND BUDGET OVERRUNS."

Make your development environment mimic your production environment:

- Replicate your LDAP/Active Directory and install applications, servers, and packages using the same security model as production (or as close to as you can)
- Never develop or run your solutions as an administrator
- When you must make a change to the environment during development, review the change with your infrastructure/operations team and document your modification in your deployment documentation.

The best starting place for resolving integration infrastructure problems is security in the form of permissions and accessibility.

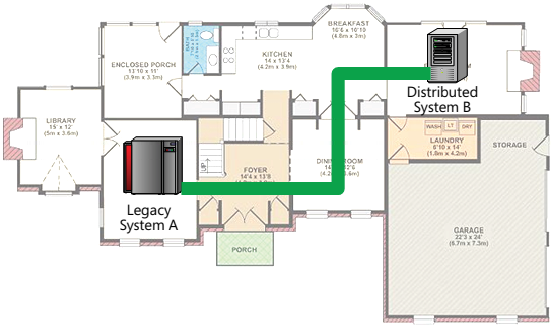
A to B vs. A to C

Integration solutions generally involve diverse system scenarios. It is good to understand the two major types of integration problems, how they are solved, and their relative complexities.

A to B Problems: One house, two systems

The characteristics of A to B problems are when a System B wishes to communicate with a System A. They generally are in the same infrastructure, but are not limited to it (they can span a WAN, VAN, the Internet).

Figure 1: A to B problem: Two systems, one house

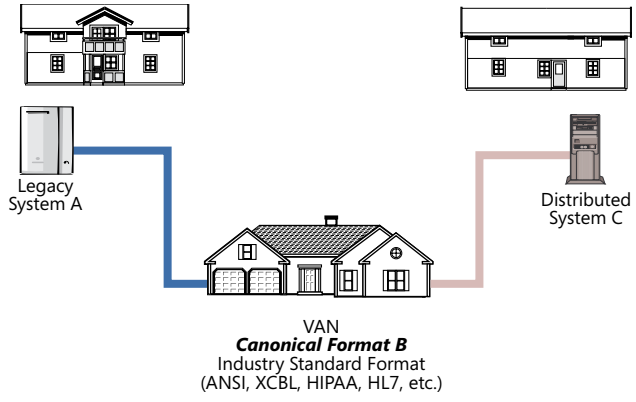


For example, System B could be a distributed system accessing information stored in System A, a legacy system (Figure 1). Direct mapping of information from A to B requires intimate knowledge of both A and B, but because domain knowledge of both systems is generally available in-house, there is less guess work on which data fields and elements in A relate to B.

A to C Problems: Multiple houses, multiple systems

An A to C problem describes scenarios in which Trading Partner A wishes to communicate with Trading Partner C, but they both must do so through a sometimes external intermediate, common, or industry standard format B. The houses are generally in separate infrastructures. (See Figure 2.)

Figure 2: A to C problem: Two systems, two houses



An example of this common situation is when trading partners use an ANSI X.12 , HIPAA, or XCBL common industry format to communicate with each other, sometimes through a VPN. Often, the intermediary schema is 10 to 100 times larger than the internal schema used by System A or C. These intermediary schemas can be 1-2 MB, resulting in performance and stability issues when introduced to packaged integration framework tools—especially frustrating when the average message payload is only 20K (Figure 3).

Aside from these more mechanical issues, the two trading partners may have to perform guesswork to decide where in the intermediary format B they are to place their information and where their trading partner will place theirs. Coupled with the fact that an intermediary format often contains bloated redundancies invariably leading to further confusion in the proper placement of information, these imprecisions may cause you to question the value in this form of integration (of course, there is value, but at times it may seem dubious).

These factors tend to make A to C problems significantly more difficult to solve, requiring far greater communication between trading partners to resolve interpretation differences with the intermediary format B.

Conquering the A to B and A to C Problems

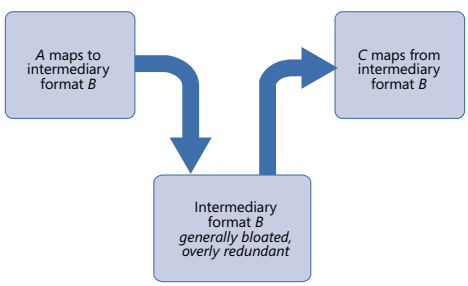
- Clearly document your intermediary format B expectations, providing many samples.
- Reduce the size of intermediary format B by working with your trading partner(s) to agree on a subset schema that includes only those components utilized by all trading partners.
- Double or triple your trading partner test projections to compensate for intermediary format B misinterpretations.

The Mapping Pit of Despair

Many integration solutions find themselves in a form of scope-creep that is never planned for but heightens frustration from overruns in time management to the budget. This is known as the *Mapping Pit of Despair.*

Once all security, infrastructure, and operating environment issues are resolved to the point actual data moves from point A to point B (or C), the interpretation of the hundreds to thousands of fields from one schema to the next becomes the primary focal point. Too often, data required by one point is not readily available at another or the intended meaning of one field is used for something entirely different. The ugliest secrets usually turn up during the mapping phase.

Figure 3: A to C scenarios often involve a bloated intermediary format B



The following examples illustrate how this might happen:

Misused Fields in Data Repositories

Trading partner A must supply identity data to schema B in the format shown in Table 1.

Simple enough, but trading partner A creatively uses the Middle Initial field to store years of service data, where 0-9 represents the number of years and A represents 10 or more years. When a middle initial is used

Table 1: Identity data format

First Name	Char *
Middle Initial	Char 1
Last Name	Char *

by an identity, trading partner A simply places it as part of the first name. Table 2 represents how data may be stored by trading partner A.

Not so simple anymore, is it?

- Trading partner A must parse the *First Name* field to determine if a middle initial exists.
- The parsing algorithm must distinguish between a two word first name, Tory Ann, and a real middle initial, Mary A.
- Improper parsing could result in the confusion between Mary A. Anderson with 2 years of service and Mary Anderson with 10 or more years of service.
- A bad choice of field to hold years of service data (likely made years prior to any intention of sharing this data) has just doubled or even tripled the time to map from trading partner A's schema to trading partner B's.

Real World Mapping

When mapping is demonstrated by integration software vendors, it usually looks something like Figure 4.

Real world mapping is a far different problem involving:

- Hundreds to thousands of fields
- Hierarchical differences in data formats that may require complicated looping algorithms to properly position data
- Large data streams requiring complicated loops to break apart messages
- Fancy integration tools sometimes lead developers to pursue a graphical solution to a problem that is far too complicated for that tool
- Pulling data from multiple internal sources (sometimes asynchronously)

Table 2: Data table stored by trading partner A

First Name	Middle Initial	Last Name
John	5	Smith
Tory Ann	8	Wilson
Mary A	2	Anderson
Mary	A	Anderson

- Manufacturing or calculating information that simply doesn't exist in any internal data repositories
- Interruptions to other team members working on other parts of the solution when a map breaks

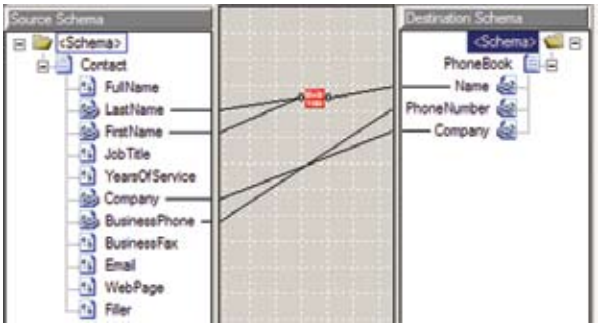
Real world maps look more like Figure 5. Significant innovations in tools for real-world maps, such as BizTalk's new XSLT Mapper, have been demonstrated and are forthcoming to alleviate this tedious task, but the challenge often stems beyond what any given tool is positioned to perform (see Resources.)

Never Ending Iterations

In normal application development, iterations are a good thing. You often can determine how many iterations will be allowed or decide on an acceptable form/function tolerance that will trigger moving on. Partial functionality is generally acceptable and you can utilize phases to introduce missing functionality at a later date.

Integration development, however, generally has zero tolerance: You iterate until it is perfect. Partial functionality and phases are usually not an option. Thus, integration becomes the source for many project delays and budget overruns.

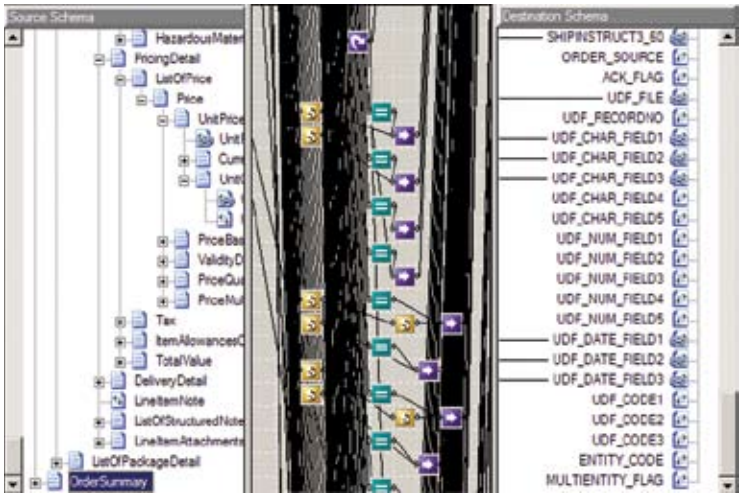
Figure 4: Integration software mapping demo



Conquering the Mapping Pit of Despair

- Make no assumptions about the difficulty in mapping; always research sources thoroughly to identify those dirty little secrets
- Establish a test/debug contract with trading partners specifying frequency for testing and issue turn-around metrics *with appropriate escalation paths*. Especially when working with external trading partners, it is imperative to define and establish testing agreements with appropriate escalation paths so that when interruptions to necessary testing procedures occur (which they most certainly will), everyone affected is notified as early as possible to appropriately communicate potential delays to the solution delivery.
- Utilize *Test Driven Development* best practices to be able to thoroughly test and defend your side of the integration and reduce/prevent downtime to other team members
- For complicated maps, consider using scripts and code over fancy UI paradigms (easier to read & debug)
- Break messages apart before mapping
- Use managed code in place of maps when necessary (for example, for better performance and complicated hierarchies). This is accomplished by using serialized classes.

Figure 5: Real world data mapping



- Understand and honor performance ramifications related to mapping (for example, because maps are XSLT scripts, never call managed code from a map)

Performance Matters

Performance always matters, especially when told it doesn't matter.

Conquering Performance Matters (BizTalk)

- Minimize trips in/out of the Message Box
- Minimize dependency on Orchestrations—which can cause your solution to execute a magnitude slower
- Performance-driven development means measuring metrics during all phases of development to identify bottlenecks as early as possible
- Utilize product performance tuning guidelines—some excellent BizTalk guidelines have been published (see Resources)
- Out-of-Box is not optimized for your solution. There are many places to tweak performance so it is important to understand all components of the solution and appropriately tune them for optimal operation
- Serialized classes and managed code may be faster than messages and maps, consider using them for performance critical components in the solution

Think of your tools as a Bunch of Legos

Once a team adopts a toolset or package to assist in the implementation of their integration solutions, a common error is to utilize *every* tool in their suite for every integration solution.

In most cases, not every tool needs to be used. In fact, using every tool may adversely affect performance of the overall solution.

A best practice is to think of your suite of tools as a bunch of Legos. Legos come in many sizes and colors. You don't need to use all sizes and all colors in all your creations. Sometimes you may wish to only use only green and white Legos while other times you may concentrate on blue and red. It all depends on your desired result.

Treat your integration tools the same. Not every solution needs an *Orchestration*. There is a performance price to pay when using an *Orchestration*. Maps can be utilized in *Ports* as well as *Orchestrations*. Sometimes, it is good to experiment by implementing several solution prototypes using various combinations of the suite's tools to understand the performance, maintenance, and deployment differences.

Know your tools and use only those you need (BizTalk)

- Make better use of the Port based *publish/subscribe* model; because there's no orchestration, this model is too often overlooked.
- Orchestrations are most effective for workflow; although they carry some overhead, Orchestrations have great purpose and are very effect when utilized for workflow.
- Map inside Ports, not just Orchestrations; this is another much overlooked capability.
- Pipelines can be fast and effective, consider a custom Pipeline Component over an Orchestration —it minimizes trips in/out of the message box and eliminates Orchestration overhead.
- Serialized classes and managed code are sometimes an effective alternative to messages and maps.

Bottom line?

Think of your integration tools as a Ferrari (with *manual* transmission) in your garage. As long as you can only drive an *automatic*, this Ferrari will be the slowest, most frustrating vehicle you've ever known. However, once you master a *manual,* it will show itself for the finely tuned high-performance racing machine it truly is.

The right tools teamed with the right skills and practices most certainly can conquer the integration dilemma.

Resources

Eddie Churchill, "BizTalk's sexy new XSLT Mapper"
http://channel9.msdn.com/Showpost.aspx?postid=126990

BizTalk guidelines
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/bts_2004wp/html/87f447a2-09ce-4a30-9f94-584684310051.asp

About the Author

Jim Wilt focuses his experience and problem solving skills toward helping customers architect the best possible solutions to succeed with their needs related to system design, collaboration, data integration, and business intelligence. He is a *Microsoft Certified Architect – Solutions* and has received several industry awards, including the *1993 Industry Week Technology of the Year Award* and the *Burroughs Achievement Award for Excellence*. He also is a *Microsoft Most Valuable Professional - Visual Developer - Solutions Architect*, member of the *Microsoft MCA Board of Directors*, the *Central Michigan University College of Science and Technology Alumni Advisory Board*, and is a *Central Michigan University Distinguished Alumni*.

Ontology and Taxonomy of Services in a Service-Oriented Architecture

by Shy Cohen

Summary

This paper describes an ontology and taxonomy for services in a Service-Oriented Architecture (SOA); it discusses the nature of and interrelationships between different service types, describes a generic blueprint for SOA-based systems, and provides some guidance on the construction and management of services. The ontology and taxonomy described here provide a common language for architects, engineers, and business decision makers, and facilitates better communication within and across different disciplines and organizations.

A Service Taxonomy

Service-oriented economies thrive by promoting composition. In SOAs, new solutions can be created by composing together new application-specific business logic and functionality with existing, recombinant business capabilities. Today, these business capabilities are mostly built in-house, or purchased for on-site deployment as a packaged solution. As we look into the near future we see continued growth in the Software as a Service (SaaS) model as an option to provide organizations with the ability to "lease" componentized solutions and business capabilities, thus enriching the set of components which can be included in the organization's composite applications.

An ontology is a data model that represents a set of concepts within a domain and the relationships between those concepts. A taxonomy is a classification of things, as well as the principles underlying such a classification. A hierarchical taxonomy is a tree structure of classifications for a given set of objects. This paper defines the service categories in an SOA, the relationships between these categories, and the principles underlying the classification of different service types into the different categories. In addition to defining, classifying, and providing the classification principles, it touches on both the software architecture and the business-related aspects relevant to the construction and management of composite applications in SOAs.

Looking at composition from a software architecture standpoint, a common ontology and taxonomy enables us to identify the common characteristics of services that fall into a particular category. These characteristics affect the architecture and design of SOA-based solutions from the individual service level up to the entire composite-application. Categorization supports composability by clarifying the roles of the different components, thus helping reason about component inter-relationships. Categorization also assists with the discoverability of services (for example, searching for existing services by using a service repository) which can further promote reuse.

Looking at composition from the business standpoint, a proper common ontology and taxonomy can help in making business-related decisions, such as how to obtain a capability (build vs. buy vs. "lease"), how to manage services (central management vs. per application), and so on.

Service Categories and Types

As we examine service types we notice two main types of services: those that are infrastructural in nature and provide common facilities that would not be considered part of the application, and those that are part of the application and provide the application's building blocks.

Software applications utilize a variety of common facilities ranging from the low-level services offered by the operating system such as the memory management and I/O handling, to the high-level runtime-environment-specific facilities such as the C Runtime Library (RTL), the Java Platform, or the .NET Framework. Solutions built using an SOA make use of common facilities as well, such as a service-authoring framework (for example, Microsoft's Windows Communication Foundation) and a set of services that are part of the supporting distributed computing infrastructure. We will name this set of services **Bus Services** (these are sometimes referred to as infrastructure services).

Bus Services further divide into **Communication Services**, which provide message transfer facilities such as message-routing and publish-subscribe, and **Utility Services**, which provide capabilities unrelated to message transfer such as service-discovery and identity federation.

The efficiency of software applications development is further increased through reuse of coarse-grained, high-level building blocks. The RAD programming environments that sprang up in the component-oriented era (such as Borland's Delphi or Microsoft's Visual Basic) provided the ability to quickly and easily compose the functionality and capabilities provided by existing building blocks with application-specific code to create new applications. Examples of such components range from the more generic GUI constructs and database access abstractions, to more specific facilities such as charting or event logging. Composite applications in an SOA also use building blocks of this nature in their composition model. We will name these building blocks **Application Services.**

Application Services further divide into **Entity Services**, which expose and allow the manipulation of business entities; **Capability Services** and **Activity Services,** which implement the functional building blocks of the application (sometimes referred to as components or modules); and **Process Services**, which compose

and orchestrate entity, capability, and Activity Services to implement business processes.

Figure 1 shows a sample composition of services in the service categories in order to implement a business process. In this sample scenario, a Process Service is shown to be orchestrating three Activity Services and two Capability Services. It is fairly typical for Process Services to tie together multiple other services in order to implement a complex business process. In the diagram, one of the Activity Services (Activity 2) is using a Capability Service (Capability 3) as indicated by the line connecting them. One possible reason for this is that Activity 2 may be implementing a multi-step activity over that Capability Service, or exposing a different service interface than that of the underlying Capability Service. Capability Services 2 and 3 are implemented over Entity Services 2 and 3 (respectively). It is interesting to note that Entity Services 2 and 3 are accessing the same underlying data store. This may be in order to expose different entities from underlying store, or expose the same entity in different ways in order to adhere to different requirements that Capability Services 2 and 3 have regarding the data model. In this sample scenario we can also see that Capability 1 resides outside of the organizational boundary (as indicated by the Internet cloud), and acts as an external resource that is woven into the business process implemented by the Process Service.

Bus Services

Bus Services are common facilities that do not add any explicit business value, but rather are part of the required infrastructure for the implementation of any business process in an SOA. Bus Services are typically purchased or centrally built components that serve multiple applications and, as a consequence, are typically centrally managed.

Communication Services

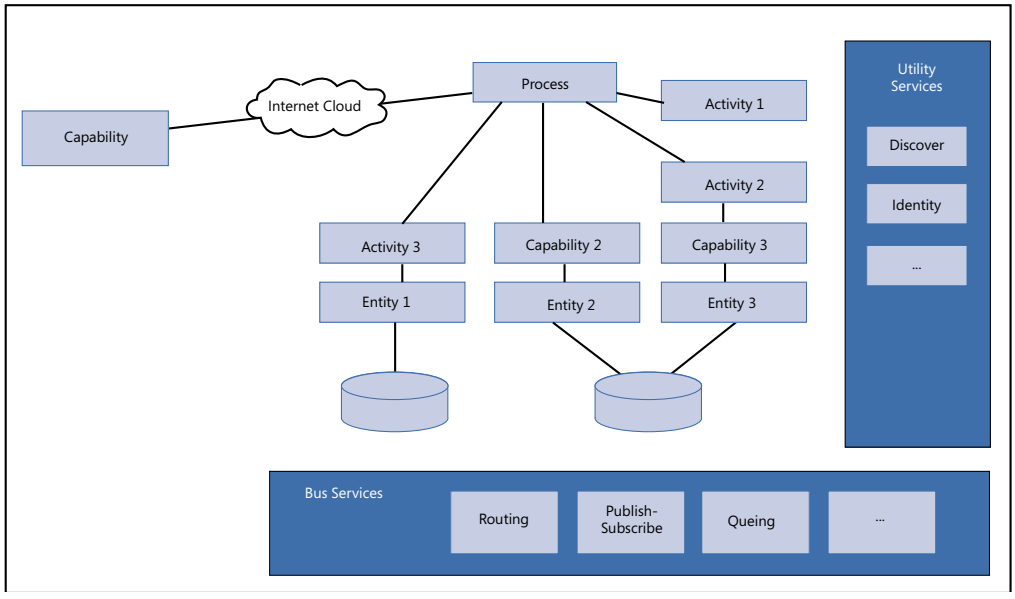
Communication Services transport messages into, out of, and within the system without being concerned with the content of the messages. For example, a bridge may move messages back and forth across a network barrier (that is, bridging two otherwise-disconnected networks) or across a protocol barrier (such as moving queued messages between IBM's WebSphere MQ queuing system and Microsoft's MSMQ queuing system). Examples of Communication Services include relays/bridges/routers/gateways, publish-subscribe systems, and queues.

Communication Services do not hold any application state, but in many cases they are configured to work in concert with the applications that use them. A particular application may need to instruct or configure a Communication Service on how to move the messages flowing inside that application such that intercomponent communication is made possible in a loosely coupled architecture. For example, a content-based router may require the application to provide routing instructions such that the router will know where to forward messages. Another example may be a publish-subscribe service that will deliver messages to registered subscribers based on a filter that can be applied to the message's content. This filter will be set by the application. In both cases, the Communication Service does not process the content of the message but rather (optionally) uses parts of it as instructed in advance by the application for determining where it should go.

In addition to application-specific requirements, restrictions imposed by security, regulatory, or other sources of constraints may dictate that in order to use the facilities offered by a particular Communication Service, users will need to possess certain permissions. These permissions can be set at the application scope (allowing an application to use the service regardless of which user is using the application), at the user scope (allowing a specific user to use the service regardless of which application that the user is using), or at both scopes (allowing the specific user to access the service while running a specific application). For example, a publish-subscribe service may be configured to restrict access to specific topics by only allowing specific users to subscribe to them.

Other application-level facilities that may be offered by Communication Services pertain to monitoring, diagnostics, and business activity monitoring (BAM). Communication Services may provide statistical information about the application such as an analysis of message traffic patterns (how many messages are flowing through a bridge per minute), error rate reports (how many SOAP faults are being sent through a router per day), or business-level performance indicators (how many purchase orders are coming in through a partner's gateway). Although they may be specific to a particular application, these capabilities are not different than the configuration settings used to control message flow. This information is typically provided by a generic feature of the Communication Service,

Figure 1: Sample composition of services in the service categories



which often needs to be configured by the application. The statistical information being provided typically needs to be consumed by a specific part of the application that knows what to do with it (raise a security alert at the data center, or update a BAM-related chart on the CFO's computer screen, for example). Table 1 summarizes the characteristics of Communication Services.

Table 1: Summary of the Communication Services category

Communication Services	
Main Purpose	Message transportation
Interface	Not processing application messages. May have Management and Monitoring interfaces.
State Management	No management of application state
Transactions	Not applicable (since they are not processing application messages)
Error Handling	No application-related error handling
Security	User/App/Both
Management/Governance	Centralized
How Built	Centrally built or purchased

Utility Services

Utility Services provide generic, application-agnostic services that deal with aspects other than transporting application messages. Like Communication Services, the functionality they offer is part of the base infrastructure of an SOA and is unrelated to any application-specific logic or business process. For example, a discovery service may be used by components in a loosely coupled composite-application to discover other components of the application based on some specified criteria (for example, a service being deployed into a preproduction environment may look for another service which implements a certain interface that the first service needs and that is also deployed in the preproduction environment). Examples of Utility Services include security and idEntity Services (for example, an Identity Federation Service or a Security Token Service), discovery services (such as a UDDI server), and message transformation services.

As in the case of Communication Services, Utility Services may also be instructed or configured by a particular application on how to perform an operation on their behalf. For example, a message transformation service may transform messages from one message schema to another message schema based on a transformation mapping that is provided by the application using the message transformation service.

Although Utility Services do not hold any application state, the state of a Utility Service may be affected by system state changes. For example, a new user being added to the application may require an update to the credential settings in the Security Token Service. Unlike in the case of Communication Services, Application Services directly interact with the Utility Services that process and (if needed) respond to the messages that the Application Services send them.

Users of Utility Services may require that a permission be configured for them in order to use the service, be it at the application, user, or the application-user scope. For example, a discovery service may only serve domain-authenticated users (users who have valid credentials issues by a Windows domain controller).

Like Communication Services, Utility Services may provide application-level facilities for monitoring, diagnostics, BAM, and so on. These may include statistical information about usage patterns (how many users from another organization authenticated using a federated identity), business-impacting error rates (how many message format transformations of purchase orders failed due to badly formatted incoming messages), and so forth. As with Communication Services, these facilities are typically generic features of the Utility Service and need to be configured and consumed by the particular solution in which they are utilized. Table 2 summarizes the characteristics of Utility Services.

Application Services

Application Services are services which take part in the implementation of a business process. They provide an explicit business value, and exist on a spectrum which starts with generic services that are used in any composite-application in the organization on one end, ends with specialized services that are part of a single composite-application on the other end, and has services that may be used by two or more applications in between.

Entity Services

Entity Services unlock and surface the business entities in the system. They can be thought of as the data-centric components ("nouns") of the business process: employee, customer, sales order, and so on. Examples of Entity Services include a customers service that manages the customers' information, or an orders service that manages the orders that customers placed.

Entity Services abstract data stores (such as SQL Server or Active Directory) and expose the information stored in one or more data stores in the system through a service interface. Therefore, it is fair to say that Entity Services manage the persistent state of the system. In some cases, the information they manage transcends a specific system and is used in several or even all the systems in the organization.

It is very common for Entity Services to support a create, read, update and delete (CRUD) interface at the entity level, and add additional domain-specific operations needed to address the problem-domain and support the application's features and use cases. An example of a domain-specific operation is a customers service that exposes a method called `FindCustomerByLocation` that can locate a customer's ID given the customer's address.

The information that Entity Services manage typically exists for a time span that is longer than that of any single business process. The information that Entity Services expose is *typically* structured, as opposed to the relational or hierarchical data stores which are being fronted by the service. For example, a service may aggregate the information stored in several database tables or even several separate

Table 2: Summary of the Utility Services category

Utility Services	
Main Purpose	Generic (non-application-specific) infrastructural functionality
Interface	Service interface for exposing the service's functionality. May also have Management and Monitoring interfaces.
State Management	No management of application state. May have their own state
Transactions	Typically not supported
Error Handling	No/limited application-related error handling
Security	User/App/Both
Management/Governance	Centralized
How Built	Typically purchased

databases and project that information as a single entity.

In some cases, typically for convenience reasons, Entity Service implementers choose to expose the underlying data as data sets rather than strongly-schematized XML data. Even though data sets are not entities in the strict sense, those services are still considered Entity Services for classification purposes.

Users of Entity Services may require that a permission be configured for them in order to use the service, be it at the application, user, or the application-user scope. These permissions may apply restrictions on data access and/or changes at the "row" (entity) or "column" (entity element) level. An example of "column" level restriction would be that an HR application might have access to both the social security and home address elements of the employee entity while a check-printing service may only have access to the home address element. An example of "row" level restriction would be an expense report application that lets managers see and approve expense reports for employees that report to them, but not for employees who do not report to them.

Error compensation in Entity Services is mostly limited to seeking alternative data sources, if at all. For example, if an Entity Service fails to access a local database it may try to reach out to a remote copy of the database to obtain the information needed. To support system-state consistency, Entity Service may support tightly coupled distributed atomic transactions. Services that support distributed atomic transactions participate in transactions that are flowed to them by callers and subject any state changes in the underlying data store to the outcome of these distributed atomic transactions. To allow for a lower degree of state-change coupling, Entity Services may provide support for the more loosely coupled Reservation Pattern, either in addition to or instead of supporting distributed atomic transactions.

Entity Services are often built in-house as a wrapper over an existing database. These services are typically implemented by writing code to map database records to entities and exposing them on a service interface, or by using a software factory to generate the mapping code and service interface. The Web Services Software Factory from Microsoft's Patterns & Practices group is an example of such a software factory. In some cases, the database (Microsoft's SQL Server 2005, for example) or data-centric application (SAP, for instance) will natively provide facilities that enable access to the data through a service interface, eliminating the need to generate and maintain a separate Entity Service.

Entity Services are often used in more than one composite application and thus they are typically centrally managed. Table 3 summarizes the characteristics of Entity Services.

Table 3: Summary of the Entity Services category

Entity Services	
Main Purpose	Expose and manage business entities
Interface	Entity-level CRUD and domain specific operations
State Management	Managing application state is the primary purpose of the service
Transactions	Atomic and/or Reservation Pattern
Error Handling	Limited error compensation (affects SLE and SLA)
Security	User/App/Both
Management/Governance	Centralized
How Built	In house/Purchased/Leased

Capability Services

Capability Services implement the business-level capabilities of the organization, and represent the action-centric building blocks (or "atomic verbs") which make up the organization's business processes. A few examples of Capability Services include third-party interfacing services such as a credit card processing service that can be used for communication with an external payment gateway in any composite application where payments are made by credit card, or a value-add building block like a rating service that can process and calculate user ratings for anything that can be rated (usefulness of a help page, a book, a vendor, and so forth) in any application that utilizes ratings. Capability Services can be further divided by the type of service that they provide (for example, third-party interfacing or value-add building block), but this further distinction is out of scope for this discussion.

Capability Services expose a service interface specific to the capability they represent. In some cases, an existing (legacy) or newly acquired business capability may not comply with the organization's way of exposing capabilities as services, or even may not expose a service interface at all. In these cases, the capability is typically wrapped with a thin service layer that exposes the capability's API using a service interface that adheres to the organization's way of exposing capabilities. For example, some credit card processing service companies present an HTML-based API that requires the user to complete a web-based form. A capability like that would be wrapped by an in-house-created-and-managed-façade-service that will provide easy programmatic access to the capability. The façade service is opaque and masks the actual nature of the capability that's behind it to the point where the underlying capability can be replaced without changing the service interface used to access it. Therefore, the façade service is considered to be the Capability Service, and the underlying capability becomes merely an implementation detail of the façade service.

Capability Services typically do not directly manage application state; to make state changes in the application, they utilize Entity Services. If a Capability Service does manage state, that state is typically transient and lasts for a duration of time that is shorter than the time needed to complete the business process that this Capability Service partakes in. For example, a Capability Service that provides package shipping price quotes might record the fact that requests for quotes were sent to the shipping providers until the responses come back, thereafter erasing that record. In addition, a Capability Service that is implemented as a workflow will manage the durable, transient execution state for all the currently running instances of that workflow. While most of the capabilities are "stateless," there are, of course, capabilities such as event logging that naturally manage and encapsulate state.

Users of Capability Services may require that a permission be configured for them in order to use the service, be it at the application, user, or the application-user scope. Access to a Capability Service is typically granted at the application level. Per-user permissions are typically managed by the Process Services that make use of the Capability Services to simplify access management and prevent midprocess access failures.

Error compensation in Capability Services is limited to the scope of meeting the capability's Service Level Expectation (SLE) and Service Level Agreements (SLA). For example, a Shipping Service which usually compares the rates and delivery times of four vendors (FedEx, UPS, DHL, and a local in-town courier service, for example) may compensate for a vendor's unavailability by ignoring the failure and continuing with the comparison of the rates that it was able to secure as long as it received at least 2 quotes. This example illustrates that compensation may result in lowered performance. This degradation can be expressed in terms of latency, quality of the service, and many other aspects, and therefore it needs to be described in the SLE and SLA for the service.

Capability Services may support distributed atomic transactions and/or the Reservation Pattern. Most Capability Services do not manage resources whose state needs to be managed using atomic transactions, but a Capability Service may flow an atomic transaction in which it is included to the Entity Services that it uses. Capability Services are also used to implement a Reservation Pattern over Entity Services that do not support that pattern, and to a much lesser extent over other Capability Services that do not support that pattern.

Capability Services can be developed and managed in-house, purchased from a third party and managed in-house, or "leased" from an external vendor and consumed as SaaS that is externally developed, maintained, and managed.

When developed in-house, Capability Services may be implemented using imperative code or a declarative workflow. If implemented as a workflow, a Capability Service may be modeled as a short-running (atomic, nonepisodic) business activity. Long-running business activities, where things may fail or require compensation, often fall into the Process Service category.

A Capability Service is almost always used by multiple composite applications, and is thus typically centrally managed. Table 4 summarizes the characteristics of Capability Services.

Table 4: Summary of the Capability Services category

Capability Services	
Main Purpose	Implement a generic value-add business capability
Interface	Service interface for exposing main functionality
State Management	Typically holds no application specific state
Transactions	No state implies that there is no need for transactions. May implement atomic and/or Reservation Pattern over an Entity service
Error Handling	Limited error compensation (affects SLE and SLA)
Security	Application-level
Management/Governance	Centralized
How Built	In house/Purchased/Leased

Activity Services

Activity Services implement the business-level capabilities or some other action-centric business logic elements ("building blocks") which are unique to a particular application. The main difference between Activity Services and Capability Services is the scope in which they are used. While Capability Services are an organizational resource, Activity Services are used in a much smaller scope, such as a single composite application or a single solution (comprising of several applications). Over the course of time and with enough reuse across the organization, an Activity Service may evolve into a Capability Service.

Activity Services are typically created to facilitate the decomposition of a complicated process or to enable reuse of a particular unit-of-functionality in several places in a particular Process Service or even across different Process Services in the application. The forces driving the creation of Activity Services can stem from a variety of sources, such as organizational forces, security requirements, regulatory requirements, and so forth. An example of an Activity Service which is created in a decomposition scenario is a vacation eligibility confirmation service which, due to security requirements, separates a particular part of a vacation authorization application's behavior in order for that part to run behind the HR department's firewall and access the HR department's protected databases to validate vacation eligibility. An example of an Activity Service used for sharing functionality would be a blacklist service which provides information on a customer's blacklist status so that this information can be used by several Process Services within a solution.

Like Capability Services, Activity Services expose a service interface specific to the capability they implement. It is possible for an Activity Service to wrap an existing unit of functionality, especially in transition cases where an existing system with existing implemented functionality is being updated to or included in an SOA-based solution.

Like Capability Services, Activity Services typically do not directly manage application state; if they do manage state, that state is transient and exists for a period of time that is shorter than the lifespan of the business process that the service partakes in. However, due to their slightly larger granularity and because in some cases Activity Services are used to wrap an existing system, it is more likely that an Activity Service will manage and encapsulate application state.

Users of Activity Services may require a permission to be configured for them in order to use the service, be it at the application, user, or the application-user scope. As in the case of Capability Services, access to an Activity Service is typically granted at the application level and managed for each user by the Process Services that are using the Activity Service.

Activity Services have the same characteristics for error compensation and transaction support as Capability Services. Activity Services are typically developed and managed in-house, and may be implemented using imperative code or a declarative workflow. As in the case of a Capability Service, if implemented as a workflow an Activity Service may be modeled as a short-running business-activity.

Activity Services are typically used by a single application or solution and are therefore typically managed individually (for example, at a departmental level). If an Activity Service evolves into a Capability Service, the management of the service typically transitions to a central management facility. Table 5 summarizes the characteristics of Activity Services.

Process Services

Process Services tie together the data-centric and action-centric building blocks to implement the business processes of the organization. They compose the functionality offered by Activity Services, Capability Services, and Entity Services and tie them together with business logic that lives inside the Process Service to create the blueprint that defines the operation of the business. An example of a Process Service is a purchase order processing service that receives a purchase order, verifies it, checks the customer blacklist

Table 5: Summary of the Activity Services category

Activity Services	
Main Purpose	Implement a specific application-level business capability
Interface	Service interface for exposing main functionality
State Management	Typically holds no app specific state (unless when wrapping and exposing an existing system)
Transactions	May support atomic transactions and/or implement the Reservation Pattern over an existing system
Error Handling	Limited error compensation (affects SLE and SLA)
Security	Application-level
Management/Governance	Per application
How Built	In house

service to make sure that the customer is okay to work with, checks the customer's credit with the credit verification service, adds the order to the order-list managed by the orders (entity) service, reserves the goods with the inventory (entity) service, secures the payment via the payment processing service, confirms the reservation made with the inventory (entity) service, schedules the shipment with the shipping service, notifies the customer of the successful completion of the order and the ETA of the goods via the email gateway service, and finally marks the order as completed in the order list.

Process Services may be composed into the workflows of other Process Services but will not be recategorized as capability or Activity Services due to their scope and long-running nature.

Since Process Services implement the business processes of the organization, they are often fronted with a user interface that initiates, controls, and monitors the process. The service interface that these services expose is typically geared towards consumption by an end-user application, and provides the right level of granularity required to satisfy the use cases that the user facing front-end implements. Monitoring the business process will at times require a separate monitoring interface that exposes BAM information. For example, the order processing service may report the number of pending, in-process, and completed orders, and some statistical information about them (median time spent processing and order, average order size, and so on).

Process Services typically manage the application state related to a particular process for the duration of that process. For example, the purchase order processing service will manage the state of the order until it completes. In addition, a Process Service will maintain and track the current step in the business process. For example, a Process Service implemented as a workflow will hold the execution state for all the currently running workflow instances.

Users of Process Services may require that a permission be configured for them in order to use the service. Access to a Process Service is typically granted at the user level.

Process Services very rarely support participation in a distributed atomic transaction since they provide support for long-running business activities (long-running transactions) where error compensation happens at the business-logic level and compensation may involve human workflows. Process Services may utilize distributed atomic transactions when calling into the services they use. They may also implement the reservation pattern.

Process Services are typically developed and manages in-house since they capture the value-add essence of the organization, the "secret sauce" that defines the way in which the organization does its business. Process Services are designed to enable process agility (that is, to be easily updatable), and the processes that they implement are typically episodic in nature (the execution consists of short bursts of activity spaced by long waits for external activities to complete). Therefore, Process Services are best implemented as declarative workflows using a workflow server (such as Microsoft's BizTalk Server) or a workflow framework (such as Microsoft's Windows Workflow Foundation).

Process Services are typically used by a single application and are therefore managed individually (for example, at a departmental level). In some cases a reusable business process may become a commodity that can be offered or consumed as SaaS. Table 6 summarizes the characteristics of Process Services.

Table 6: Summary of the Process Services category

Process Services	
Main Purpose	Implement a business process by orchestrating other services
Interface	Service interface targeted at user-facing applications
State Management	Manages process state
Transactions	No support for atomic transactions. May use atomic transactions with the services it uses. May implement a Reservation Pattern
Error Handling	Errors handling is implemented as part of the business logic
Security	User
Management/Governance	Per application
How Built	In house

Conclusion

For the architect, having a good grasp of the different categories assists in classifying existing or new services, as well as helps define the appropriate functionality to include in a particular service in order to promote composition and reuse. The architectural blueprint defined here can be used for the design of new systems as well as the refactoring of existing systems.

For the business decision maker, understanding the business value of a component and its commoditization level makes it easier to make build vs. buy vs. lease decisions, and may expose business opportunities for making a service available for others.

About the Author

Shy Cohen is a program manager in the Distributed Systems Group at Microsoft. Shy joined Microsoft in 1996 and has worked on different technologies at different groups in the company, always staying true to his decade-long passion for distributed computing.

For the past five years, Shy has focused his time on designing several of the technical and architectural aspects of Windows Communication Foundation (WCF). Today, he directs his attention to several aspects relating to the creation of distributed systems and provides guidance on distributed systems, SOA, Web Services and WCF, and workflow technologies.

Versioning in SOA

by Boris Lublinsky

Summary

Service-Oriented Architecture (SOA) is taking a center stage in enterprise architecture. SOA enables parallel development by multiple disparate teams, each with its own delivery and maintenance schedule. In this article, I will examine service versioning approaches, allowing service implementations to evolve without breaking existing consumers, leading to more loosely coupled SOA implementations. The basic idea of service versioning is fairly simple, but its implementation requires strict governance; I will discuss units of versioning; service changes, constituting a new version; service version lifecycle considerations; and version deployment/access approaches. Method-based service versioning proposed here allows to minimize the impact of versioning and to reduce amount of deployed code. Semantic messaging for service interface definitions makes service implementation more resilient to change.

If there is a constant in IT implementation, it is change. Business conditions change, consequently requiring IT implementations to change. New techniques and patterns allow for better implementation of qualities of service, such as load balancing and failover, security, and so forth. Information technology itself changes continually with the introduction of new operating systems, programming languages, and application servers, for example, that simplify creation and maintenance of new solutions. In fact, a driving force in the implementation of an IT solution is its ability to cope with these inevitable changes.

In the era of monolithic applications, changes were dealt with on an application-by-application basis. Implementation of change, whether for a new business or infrastructure—for example, the introduction of a security policy or requirement, or moving an application to a new software platform—was done for an application as a whole, consuming significant amounts of time and money to complete. On the other hand, because each application was developed by a single team and independent, this approach allowed changes to be contained. As a new version of an application was introduced, the previous version came out of use and could be disposed.

One of the main advantages of a service-oriented architectural style is its ability to efficiently deal with changes. SOA is based on a decomposition of enterprise IT assets and separation of "stable" IT artifacts (services) from "changeable" artifacts (business processes), orchestrating services into IT solutions (processes). As a result, business requirements changes can often be satisfied by either changes to existing processes or creation of new enterprise business processes based on the existing services. This approach allows for much better (faster, cheaper) support for required changes through (re)assembling of a solution based on the reusable enterprise services. Business services become resources, shareable by multiple business solutions, enabling massively parallel autonomous development of services by multiple disparate teams, each with its own delivery and maintenance schedule.

Because every service, in this case, is used simultaneously in multiple enterprise solutions, a change in a business service can have a significant impact on many existing implementations and may consequently require changes in each of them. This is not only extremely expensive (requiring a tremendous amount of coordination between development teams and testing, ensuring that none of the multiple service consumers are affected), but also goes against one of the fundamental SOA tenets: Services are autonomous. Autonomy is the fundamental concept behind service orientation. Services should be deployed and modified/maintained independently from each other and the systems that use them.

"AUTONOMY IS THE FUNDAMENTAL CONCEPT BEHIND SERVICE ORIENTATION. SERVICES SHOULD BE DEPLOYED AND MODIFIED/ MAINTAINED INDEPENDENTLY FROM EACH OTHER AND THE SYSTEMS THAT USE THEM"

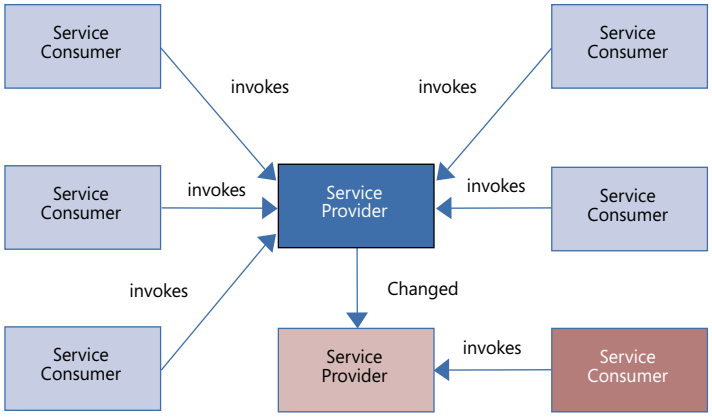
The problem of dealing with shared, potentially changing, components is not a new one. For example, Windows operating system and Windows-based applications rely on the shareable COM/ ActiveX components. In the COM case, components are considered immutable and any functional change requires introduction of new component with a new globally unique identifier (GUID). Such approach allows for simultaneous coexistence of multiple components/versions.

In this article, I will discuss service versioning approaches, allowing service implementations to evolve without breaking existing consumers.

Introducing Service Versioning

One the most popular approaches for dealing with changes is versioning. Versioning assumes simultaneous existence of

Figure 1 Coexistence of multiple service versions



multiple (different) implementations of the same thing, with every implementation distinguishable and individually addressable. In the case of SOA, service versioning equates to coexistence of multiple versions of the same service which allows each consumer to use the version that it is designed and tested for (see Figure 1). In this case, a new version of a service is created based on the requirements of one or more consumers, which can start using this new version immediately. The other consumers of this service do not need to switch to using the latest version immediately, but can continue to use the versions of the service they were designed for and tested with. They can switch to the latest version of service, based on their own development and testing schedule. Multiple coexisting versions of the same service in the system allows for the independent life cycles of services and their consumers and minimizes the overall impact of the introduction of changes. Although the necessity of such versioning mechanism may be obvious to anyone who has ever dealt with services, this topic still has not penetrated the mainstream of SOA publications and implementations. The basic idea of service versioning is fairly simple and straightforward, but its implementation requires definition of the following:

"METHOD-BASED SERVICE VERSIONING PROVIDES ENHANCED FLEXIBILITY AND BETTER ALIGNS SERVICE VERSIONING WITH THE VERSIONING PRACTICES PREVALENT FOR PROGRAMMING LANGUAGES"

- Units of versioning
- Service changes, constituting a new version
- Service version lifecycle considerations
- Version deployment/access approaches.

Units of Versioning

There are two major options for defining units of versioning, each with its own benefits and liabilities.

Based on the current Web Services practices (the most popular SOA implementation technology today), many practitioners propose *versioning of the service (including all its operations) as a whole*. Although this approach is well aligned with the current object-oriented (OO) and component-based development (CBD) practices, it does not always seem to be appropriate in the case of coarse-grained services. (For further comparison of SOA and OO, see Resources, "Defining SOA as an Architectural Style.")

When people talk about versioning in their everyday conversations, they usually talk about changes and versioning of the methods, not a service itself. For example, consider an account service that implements three operations: withdrawal, deposit, and transfer. Typically the conversation revolves around changes of the individual operations (i.e., withdrawal), not the account service itself.

Therefore, another option is to *define individual service operations (methods) as a unit of versioning*. Versioning service operation independently has the following advantages:

- It allows for immutable services. The service can provide additional methods (versions of methods), some of which can be deprecated over time, but the service itself, its name and classification, never changes. This scheme resembles versioning approaches in popular programming languages, such as Java or C#, where methods on classes are added and deprecated quite often while existing classes that are widely used rarely change.
- It minimizes the impact of the service changes to consumers. Only consumers using a particular method are affected by a change, rather than all service consumers.
- It minimizes the overall amount of deployed code. Only methods with a new version get redeployed in the process of introduction of the new version. The code implementing methods that have not changed remains unchanged.

It also has the following liabilities:

- It calls for deploying each method independently with its own endpoint address(es). (Although such deployment approach is not mainstream today, it has some advantages, such as providing different service level agreements (SLA) for different methods within the same service.)
- It requires a different, slightly more complex, service invocation addressing schema. Rather then specifying a service that it needs to invoke, the service consumer, in this case, needs to explicitly specify the service, the operation, and the version of the operation that it requires.

Despite the requirement for non-standard service invocation/routing, method-based service versioning provides enhanced flexibility and better aligns service versioning with the versioning practices prevalent for programming languages. It also minimizes the amount of code that has to be redeployed to support a new version. These characteristics make method-based versioning a powerful service versioning approach.

Version Definitions

Defining what constitutes a new version of the service (method) requires analyzing the possible changes, their potential impact on the consumer's execution, and identifying the ones that will "break"

it. *Any change in the service, whether it is a change in the interface or implementation that might impact consumer execution, should lead to creation of the new version of service (method).* Although the above definition is not very precise and open for interpretation, it provides a good starting point for decision on creation of the new version.

I will further examine the major components of service definition (interface and message definitions) and implementations to determine particular situations that can lead to new version creation.

Service Interface Changes

Following the adoption of the semantic messaging model, the service method signatures never change (all changes are reflected in the semantic model changes). The service method's interface in the case of semantic messaging is:

servicemethod (XML in, XML out)

Consequently, the service method's interface never changes and should not be considered in versioning definition. Because every method is individually deployed and addressed in a method-based versioning scheme, additional methods can be introduced for the service without impacting existing service consumers. Finally, because all the interface changes are contained in the messaging model, method removal (deprecations), in this case, is equivalent to elimination of some of the enterprise functionality and should happen very rarely. From the consumer point of view, this situation requires (potentially significant) modifications, which entail either internal implementation of the required functionality or usage of a completely different service. In this case, the service method is defined as deprecated and is kept around, while each of its consumers will be able to stop using it (see service versions life cycle considerations further in the article).

Message Changes

As defined above, in the case of semantic messaging, the changes in the service interface are contained in the semantic messages changes. These messages are defined using schemas describing their content. Usage of the messaging schemas for defining

Versioning Support in XML Schemas

The simplest way of denoting versions in XML Schema is usage of an (optional) attribute at the xs:schema element - version. The content model permits Dewey notation of major.minor version numbers.

Since XML parsers are not required to validate instances using version, it is possible to implement custom representation of version, enabling the parser to include it in the validation process. Using this technique typically requires introduction of a versioning attribute as a fixed, required value for identifying a specific schema version. However, this approach for schema versioning is not very practical. There are several disadvantages:

- An XML instance will be unable to use multiple versions of a schema representation because versioning occurs at the schema's root.
- XML schema validation tools are not required to validate instances using the version attribute; the attribute is provided purely for documentation purposes and is not enforceable by XML parsers.
- Since XML parsers are not required to validate using the version attribute, additional custom processing (over and above parsing and validation) is required to ensure that the expected schema version(s) are being referenced by the instance.
- Marshaling/unmarshaling of XML documents is very rarely done using direct manipulation of the DOM tree. The prevalent approach to marshaling is generation of the classes, supporting "automatic" marshaling, using tools like WSDL2Java, Castor, EMF, SDO, XSD, XSDObjectGenerator, and so on. In this case, classes are generated in the packages in Java or namespaces in C#, based on the schema namespaces, not the schema version.

Another option for denoting schema version is usage of XML Namespaces. In this approach, a new XML Namespace is used for all major version releases. This approach is well aligned with generation of marshaling/unmarshaling code by allowing generating code in different packages (namespaces), thus enabling a single service consumer to work with several major releases of schema simultaneously.

Yet another option is to keep XML Namespace values constant and add a special element for grouping custom extensions. This approach wraps extensions to the underlying vocabulary within a special extension element. This technique is favored by several industry-standard schemas. For example, the Open Application Group's Business Object Documents (OAG BODs) include a <userarea> element defining custom information that may not be part of the base vocabulary. This approach maximizes the extensibility of the schema constructs (schemas can be both forward and backward compatible) without introduction of new namespaces. There are two disadvantages to this approach:

- It introduces significantly higher levels of complexity into the schema.
- It does not allow implementation of multiple extensions across different portions of the XML instance since all extensions must be grouped within the extension "wrapper."

The most scalable approach to versioning of schemas is as follows:

- Componentization of the overall schema in logical partitions using multiple namespaces, thus containing changes.
- Defining a new namespace (reflecting the major version information) for every major version of each schema.
- Denoting every minor version as a schema version in a major version namespace. Because minor versions are backward compatible, generated marshaling/unmarshaling code will be backward compatible as well.

potential interface changes aligns this approach with XML schema versioning techniques (see the sidebar, "XML Schema Versioning"), and thus allows for direct representation of versioning in the service messages. Changes in schemas can be broadly defined in three major categories:

Revisions represent document's schema changes with no semantic meaning. For example, a change in white space, formatting, non-normative documentation, comments, and so on. A revision of an already published version must not impact the functionality of either service implementations or consumers.

Additionally, the initial incremental development of a semantic schema, before it is published for production use, can also be treated as revisions of the same version.

Minor changes represent backward-compatible changes to the document schema. Examples of minor changes to the schema include:

- Changing the optionality of a local element or element reference from required to optional
- Adding a global element or type
- Adding optional elements to the existing type
- Changing type of a global or local element to the new type, derived from the original by adding/restricting optional elements.

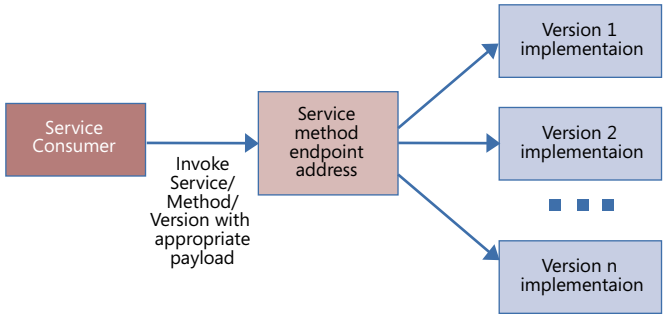
Major changes represent non-backward-compatible changes to the document schema.

Examples of major changes to the schema include:

- Changing the type of a local or global element to the new type, derived from the original by adding/restricting optional elements.
- Changing the optionality of a local element or element reference from optional to required
- Adding or removing an enumeration value
- Removing or renaming a global type or element.

Based on the above definitions, both revisions and minor changes in the messaging schema provide backward compatibility and will not "break" the service contract. As a result they will not impact either service or consumer implementation and do not require service versioning. In contrast, major changes require versioning of messaging schema and, consequently, services.

Figure 2 Implementation of versioning using covenant



Implementation Changes

A common misconception is that, because services are defined through the interface—not implementation, changes in the service implementation will not impact service consumers and do not require versioning.

In reality, adherence to the interface does not constitute replaceability of implementations. Replaceability is not defined by interface alone, but rather by the contract, which includes interface, pre- and post-conditions, and certain SLAs, which the service consumer relies on. Consequently, versioning is required when service implementation changes affect the contract on which a particular service consumer relies. Because different service consumers can rely on different parts of the service contract, every service implementation changes should be validated against all existing service consumers to ensure that no contracts will be "broken."

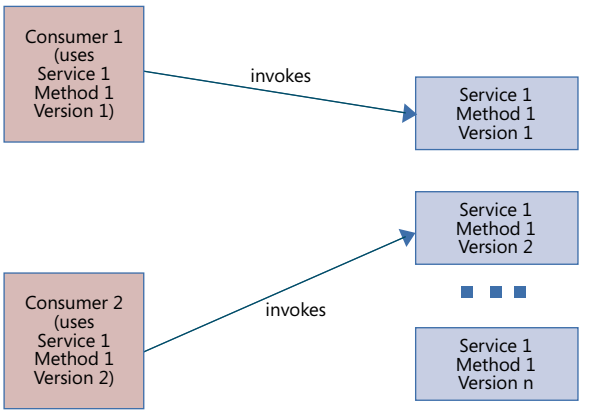
"AN IMPORTANT VERSIONING CONSIDERATION IS DEFINING THE LENGTH OF TIME FOR WHICH A VERSION OF SERVICE (METHOD) WILL BE PRESERVED. THE APPROPRIATE LIFE CYCLE FOR SERVICE (METHOD) VERSIONS VARIES SIGNIFICANTLY AND IS DEFINED BY ORGANIZATION'S ABILITY TO COPE WITH CHANGES"

Here are some of the examples of how changes in service (method) implementation can lead to the contract changes:

- New service implementation supports exactly the same interface (functionality), but changes execution timing (SLA). If a service consumer is invoking this service (method) synchronously, such change can significantly impact service consumer execution. As a result, service versioning might be required. (Interestingly enough, even redeployment of existing service, with changes to its implementation, can lead to the same results.)
- New service implementation supports the same interface, but changes incoming parameters validations (preconditions). In this case, some requests that had been processed successfully will now be rejected, requiring introduction of a new service.
- New service implementation introduces a new security implementation (precondition), which often is not reflected in service interface, but done through the configuration settings. (See Resources, *Web Services Handbook for WebSphere Application Server Version 6.1* by Wahli et al. and *Web Service Security* by Hogg et al.) In this case existing service consumers will require sending security credentials – implementation change and creation of the new version is required.

Every one of these scenarios requires analysis of existing service consumers and potentially extensive testing. As a result, a simpler way to decide whether a new version of service (method) needs to be created when the service implementation changes is to maintain a definition of service (method) contract and creating a new version when the contract changes, regardless of which consumers rely on which parts of the contract. When in doubt, it is usually simpler to create a new version.

Figure 3 Implementation of versions using directly exposed endpoint addresses



Service Version Life Cycle Considerations

One of the important versioning considerations is defining the length of time for which a version of service (method) will be preserved. Extending this period leads to the necessity of maintaining excessive amount of service versions. Shortening the period, on another hand, constrains the time for service consumers to implement required upgrades. The appropriate life cycle for service (method) versions varies significantly and is defined by organization's ability to cope with changes.

Version Deployment/Access Approaches

There are two common approaches to the deployment of service versions:

Covenant or **Version Parameter**. A covenant is an "if-then-else" agreement ("if you do this then I will do that"). In this case, there is a single endpoint address for all versions of the service (method), as shown in Figure 2.

The covenant effectively implements context-based routing, taking an incoming message and routing it (based on a version parameter, embedded in the invocation message) to the appropriate service version. The benefit of this approach is that it simplifies service addressing from the consumer point of view. The consumer, in this case, uses a single endpoint address to access all versions of a given service (method) and encodes the required method in the invocation message. An endpoint address implements a routing support, invoking a required version.

Although the covenant approach minimizes the impact of introduction of new versions on the service consumers, it introduces the complexity of packaging together multiple versions of a service method. This can lead to class name collisions, database names collisions, and so on. Furthermore, this approach effectively requires a versioning strategy not only for services themselves, but also for the components used for service implementations. Considering the tighter coupling between components, this problem can be even more complex than services versioning.

Further improvement can be achieved by replacing the local router dispatching between the service versions implementation with an external broker (**mediator**). In this case all versions can be deployed independently and it is a responsibility of mediator to dynamically resolve endpoint address of the desired service version and dispatch all messages accordingly. However, although intermediaries (mediations) are often touted by ESB publications as a cure for most routing/transformation problems, encountered in Service-Oriented Architecture, there are costs associated with them. Typically, it lowers performance. It also must support the most stringent SLA of all the services accessed through it, which could be a very strong requirement.

Multiple endpoint addresses. In this case, similar to the mediator implementation, every version of a given operation is deployed at its own endpoint address. The difference here is that every endpoint address is directly exposed to a service consumer (see Figure 3).

Multiple endpoint addresses assume that a service consumer can resolve endpoint address (typically using service registry) for a required version based on the service/method/version information. The advantage of this scheme is a complete separation of multiple method versions deployment. The drawback is a more complex addressing paradigm, requiring service registry support for resolving endpoint address.

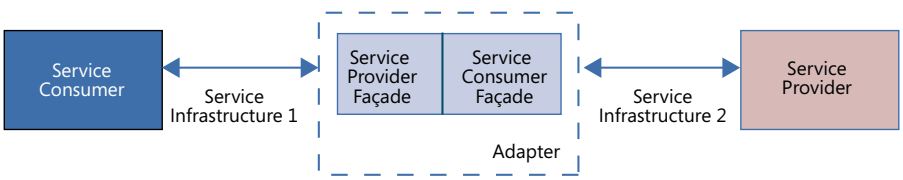
The multiple endpoint address approach typically provides better scalability (one less network hop) and lowers the coupling between multiple versions of the same operation.

Versioning of Service Infrastructure

A variant of service versioning is versioning of the service infrastructure, which can involve the following:

- Transport changes, for example, switching transport from HTTP to Java Message Service (JMS)
- Message encoding changes, for example upgrading proprietary enveloping with Simple Object Access Protocol (SOAP)
- Changes in addressing schema, for example, introduction of the Web Services Addressing for reply address specification

Figure 4 Interoperability between service infrastructures



In this case it is always desirable to implement "backward compatibility," ensuring that new infrastructure can "understand" and support messages produced by the old infrastructure and produce messages compatible with it. In reality, it might be too expensive, or even technically impossible.

Moving all of the existing service implementations and consumers to the new infrastructure is typically fairly expensive and time-consuming, requiring versioning support that provides interoperability between two different service infrastructures. The most popular

solution to this problem is a service adapter (see Figure 4).

In this case, when the service consumer supported by service infrastructure 1 invokes the service provider supported by the service infrastructure 2, the invocation goes through the adapter mediating between the service infrastructures. From the point of view of the service consumer, the adapter acts as a service provider. The adapter then invokes the service provider acting as a service consumer.

This implementation can be combined with service versioning deployment topology (*Figure 4*) to provide versioning support for both services and service consumers between different infrastructures.

Conclusion

Dealing with changes is never simple and the complexity usually grows with the size of implementation. Larger implementations typically have more interdependent moving parts, and consequently, the effects of changes are more pervasive. SOA implementations, especially enterprise-wide implementations, are no exception.

Service versioning approaches, described in this article, leads to the creation of much more loosely coupled SOA implementations. Introduction of simultaneously deployed multiple service versions allow both service consumers and providers to evolve independently, on their own development and deployment schedules. Independent versioning of service methods minimizes the impact of versioning and reduces amount of deployed code. Finally, usage of semantic messaging for service interface definitions makes service implementation more resilient to changes.

Acknowledgements

The Author thanks Jim McKune, Dmitry Tyomkin, Kiran Achen, and Luc Clement for their contributions.

Resources

"Defining SOA as an Architectural Style," Boris Lublinsky, IBM developerWorks, January 2007
http://www-128.ibm.com/developerworks/architecture/library/ar-soastyle/

"Principles of Service Design: Service Patterns and Anti-Patterns," John Evdemon, MSDN, August 2005
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/SOADesign.asp

"Best Practices for Web Services Versioning," Kyle Brown, Michael Ellis, IBM developerWorks, 2004
http://www-128.ibm.com/developerworks/webservices/library/ws-version/

"A SOA Version Covenant," Rocky Lhotka
http://www.theserverside.net/articles/showarticle.tss?id=SOAVersioningCovenant

XFire User Guide, Versioning Best Practices
http://docs.codehaus.org/display/XFIRE/Versioning

"Architecting Industry Standards for Service Orientation," Josh Lee, Microsoft 2005
http://msdn.microsoft.com/architecture/thinkahead/default.aspx?pull=/library/en-us/dnbda/html/aisforsoa.asp

"Principles of Service Design: Service Versioning," John Evdemon, Microsoft 2005
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/SOADesignVer.asp

Web Services Handbook for WebSphere Application Server Version 6.1 (Draft IBM Redbook), Ueli Wahli, Owen Burroughs, Owen Cline, Alec Go, Larry Tung
http://www.redbooks.ibm.com/redpieces/abstracts/sg247257.html?Open

Web Service Security. Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (WSE) 3.0, Jason Hogg, Don Smith, Fred Chong, Dwayne Taylor, LonnieWall, and Paul Slater (Microsoft Press, 2005)
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/wssp.asp

"Patterns: Implementing an SOA using an Enterprise Service Bus in WebSphere Application Server V6," Martin Keen, Oscar Adinolfi, Sarah Hemmings, Andrew Humphreys, Kanthi Hanumanth, Alasdair Nottingham, IBM Redbooks, 2005
http://www.redbooks.ibm.com/abstracts/sg246494.html?Open

"Supporting Policies in Service-Oriented Architecture," Boris Lublinsky, IBM developerWorks, 2004
http://www-128.ibm.com/developerworks/webservices/library/ws-support-soa/

"Toward a Pattern Language for Service-Oriented Architecture and Integration, Part 1: Build a service eco-system," Ali Arsanjani, IBM developerWorks, July 2005
http://www-128.ibm.com/developerworks/webservices/library/ws-soa-soi/

About the Author

Boris Lublinsky has over 25 years experience in software engineering and technical architecture. For the past several years, he focused on Enterprise Architecture, SOA and Process Management. Throughout his career, Dr. Lublinsky has been a frequent technical speaker and author. He has over 40 technical publications in different magazines, including Avtomatika i telemechanica, *IEEE Transactions on Automatic Control, Distributed Computing, Nuclear Instruments and Methods, Java Developer's Journal, XML Journal, Web Services Journal, JavaPro Journal, Enterprise Architect Journal and EAI Journal*. Currently Dr. Lublinsky works for the large Insurance Company where his responsibilities include developing and maintaining SOA strategy and frameworks. He can be reached at blublinsky@hotmail.com.

Microsoft®

THE ARCHITECTURE JOURNAL™

Input for Better Outcomes

098-107719