Root Document Turning the Network Into the Computer: The Emerging Network Application Platform

Version: 1.0, Jul 08 2004

AUTHOR(S):
Anne Thomas Manes
(amanes@burtongroup.com)

Jamie Lewis
(jlewis@burtongroup.com)

Additional Input:
Peter O'Kelly, James Kobielus, Chris Haddad, Dan Blum

TECHNOLOGY THREAD:
VantagePoint

Conclusion

Enterprise IT departments are being asked to deliver increasingly high levels of integration, both internally and externally, on shrinking budgets. These factors are driving both the commoditization and standardization of integration software and the emergence of the network application platform (NAP). Based on service-oriented architecture (SOA) and the Web services framework (WSF), the NAP defines a vendor-independent infrastructure that is turning the network into a practical platform for a new generation of applications. Given today's business environment, enterprises must factor the emergence of the NAP into their long-term plans, and change both their mindset and architectures for application development.

Publishing Information

Burton Group is a research and consulting firm specializing in network and applications infrastructure technologies. Burton works to catalyze change and progress in the network computing industry through interaction with leading vendors and users. Publication headquarters, marketing, and sales offices are located at:

Burton Group
7050 Union Park Center, Suite 510
Midvale, Utah USA 84047-4169
Phone: +1.801.566.2880
Fax: +1.801.566.3611
Toll free in the USA: 800.824.9924
Internet: info@burtongroup.com; www.burtongroup.com

Synopsis

The increasingly virtual nature of today's enterprise demands higher levels of application integration and interoperability, both within and between enterprises. But the proliferation of application platforms from multiple vendors has made application integration one of the toughest problems facing enterprise IT managers. While many integration middleware solutions support standards, none are based on a ubiquitous and widely accepted set of standards that lower the cost and ease the difficulty of application integration and interoperability.

But with the advent of the Web services framework (WSF) and the trend toward service-oriented architecture (SOA), such standards are starting to emerge. Market dynamics are forcing vendors to enable levels of interoperability that seemed impossible 10 years ago. And lessons learned from past attempts to solve the integration problem have yielded a more practical architecture based on loose coupling and federated services. This combination of factors creates a much higher probability of success for SOA and the WSF.

The WSF and SOA are enabling IT infrastructure that can support a truly “networked business,” providing real-time application integration that will reach beyond the confines of the enterprise. We refer to this enabling infrastructure as the network application platform (NAP). The NAP will enable a new generation of applications that can exploit the full potential of the Internet.

The NAP is an operating system (OS)- and language-independent environment that defines a model for implementing SOA. It consists of the service bus, which provides the underlying communications infrastructure; the infrastructure services model (ISM), which provides a robust set of federated infrastructure services; and service design practices (SDP), which define design principles and best practices for loosely coupled, reusable services.

The NAP is not a wholesale replacement for all existing application architectures. It is additive, giving enterprises the platform necessary to extend their business processes across the network. It will also take years for the NAP to evolve. The NAP will emerge over the next three to five years as a result of the cumulative effect of standardization and commoditization. It is not a product, nor can a single vendor provide it. It is an architectural goal state toward which enterprises should start working now.

Given its long-term ability to enable business, the NAP is a strategic imperative for most enterprises. Consequently, enterprises need to implement their own version of the NAP in accordance with their business needs.

Analysis

Today, discussions of the “networked business,” which we've encapsulated with our definition of the virtual enterprise network (VEN), can seem like a brilliant grasp of the obvious. It's clear, for example, that the piece-parts of enterprise IT—the networks, operating systems (OSs), middleware, applications, and information assets that comprise today's computing infrastructures—are now inextricably intertwined with how companies do business. To varying degrees, most enterprises are using IT to improve business process, boost productivity, and increase customer satisfaction, all while holding down costs.

At the same time, conventional wisdom holds that enterprise software strategies are no longer about installing new application silos. Today, the application software business is about leveraging existing applications, data assets, and services, by integrating them to create a more seamless whole that serves the business.

Dutifully, every vendor of import has latched onto the concept of more business-driven IT, using marketing slogans such as “the adaptive enterprise” and “on-demand computing.” Such marketing hype can quickly drain real meaning out of these concepts. But it's important to understand two basic realities when it comes to today's enterprise application strategies. First, these trends are not phantoms; they are real and are having substantial impact on application architecture over the long term. Second, enterprise computing infrastructure is not yet mature enough to make the dream of on-demand computing a reality.

Yes, service-oriented architecture (SOA) and Web services are all the rage, and these technologies have real potential to solve real problems. But in the reality of today's enterprise, the complexities of system integration, application development, deployment, and dynamic business change pose substantial challenges. In terms of everyday operation, for example, application integration and interoperability are seemingly intractable problems for any organization working to improve process while holding down costs. This is especially true when one considers the complexities of intercompany process integration.

The Cusp of Change

Simply put, application integration, interoperability, and security are the most significant obstacles to the long-term vision of a business-driven, utility-oriented computing environment. Today, enterprise IT departments are under inexorable pressure to achieve higher levels of integration, both internally and externally, with tighter budgets. In combination, these realities define both the rock and the hard place. Like the pressures that build along the fault lines between tectonic plates, the market pressures created by the convergence of these issues have reached the point at which only a significant movement can relieve them.

In other words, application architecture is on the cusp of great change. The application platform as we know it—defined as the OSs, services, protocols, interfaces, and tools that enable the development and deployment of applications—is undergoing a significant transformation as it makes the transition to truly networked architectures.

Today, application platforms are still largely defined by individual OSs, applications servers, and programming languages. Each vendor's platform provides its unique methods to support integration and interoperability, and few systems can cross the barriers between platforms. But to meet today's business needs, enterprises need an integration and interoperability infrastructure that transcends the constraints of individual implementations. The result will be establishing a vendor-independent environment that binds applications, platforms, and services from multiple vendors into a cohesive, usable framework. Only then can the network truly and finally become an effective and practical application platform that supports business process.

Just as the enterprise itself is becoming more virtual, then, the application platform is becoming more virtual as well. In the past, application silos met business needs because the application context they addressed was relatively self-contained. Today, however, organizations are going virtual in multiple dimensions. For critical systems that support business process, the application context is no longer a self-contained universe. The typical business application context is growing to serve large numbers of customers, to integrate partners and suppliers, and to enable an increasingly mobile workforce.

To meet business needs, application platforms must follow suit. Application platforms are evolving beyond the confines of the physical—a single machine, a single OS, or even groups of machines—to inhabit the more logical, and larger, environment of corporate networks, corporate extranets, and the Internet itself. At the same time, application platforms must become much more agile and flexible, capable of adapting to rapidly changing business needs.

The Advent of the Network Application Platform

We first introduced the basic concept of the network application platform (NAP) in the *Application Platform Strategies* overview, "The Advent of the Network Platform: Web Services Move into the IT Fabric," in September 2003. Although the Web services framework (WSF) that document describes is an essential ingredient for the creation of this new platform, the NAP, by necessity, consists of much more than the WSF.

In essence, the NAP provides a virtual application platform that enables applications to exploit the power of the network. It provides the infrastructure necessary to enable interoperability on a large scale, and the development environment that applications need to leverage the infrastructure. It's independent of both underlying OSs and the programming languages and tools that developers use to build applications. In short, the NAP defines a model for implementing service-oriented architecture (SOA). It consists of three core concepts:

- **The service bus:** The service bus is a language- and OS-neutral communications infrastructure that enables access to application and infrastructure services both within and across organizational boundaries.
- **The infrastructure services model (ISM):** The ISM establishes a service-oriented architecture for infrastructure services such as security and transactions. The ISM exposes infrastructure functionality as reusable services accessible through vendor-independent APIs, protocols, and frameworks. The ISM also supports a federated model for connections between systems at important operational junctures, including data, transactions, policy, security, and management. Via federation, the NAP supports a more practical model for integrating applications both within and outside enterprise boundaries.
- **Service design practices (SDP):** The SDP comprises design principles and best practices for loosely coupled, reusable services. These practices ensure flexibility, platform neutrality, and cross-platform interoperability.

Not a Zero-Sum Game

It's important to note that the NAP, along with the core principles on which it's based (such as SOA), is not an “either/or” replacement for all existing applications architectures. Individual OSs, application servers, and programming languages will remain important components of enterprise IT. And while the herd mentality of the software industry can lead one to believe that SOA and Web services are the answers for all problems, application architectures are not like stretch socks; one size does not fit all.

Decisions to use SOA and the interoperability framework the NAP establishes will turn on many factors. These include how high the requirements are to share data, business logic, and other resources; what the performance requirements are for any given application; the complexity of the data model; and the sensitivity of the security requirements. In some cases, monolithic architectures may prove to be more secure. In other cases, business needs may dictate self-contained architectures. While SOA and the NAP are much more flexible, enterprises will still rely on monolithic applications for many years to come.

The NAP, then, is additive: new infrastructure designed to meet a new set of business needs. While it won't necessarily make all existing architectures obsolete, the NAP will create a larger whole in which existing (and new) OSs, applications, and data assets can participate. The NAP will accomplish that goal by enabling an end-to-end processing model, one that standardizes the connectivity infrastructure known as “middleware” and substantially increases the flexibility and adaptability of enterprise IT systems.

Business Drivers

In the 1990's, enterprises began to realize that IT systems were not just a necessary evil, but could be used to gain a strategic advantage. Agile IT systems allow a business to adapt in real time in order to capitalize on new opportunities, such as:

- Provisioning a new supplier or partner that offers better prices, better services, or better quality materials

- Delivering new products and services within a constrained opportunistic window
- Implementing a customer retention program in response to a competitive threat
- Stepping up production or distribution of a particular product in response to a climatic, economic, or political event

By the end of the twentieth century, enterprises realized that computer networks and application systems were no longer just tools for doing business in the physical world. Today, the network is a business medium in itself. Businesses not only conduct transactions, but also create and maintain relationships with their partners, suppliers, and customers over the network. In that sense, networks, applications, and enterprise information assets aren't just becoming inextricably intertwined with the business; they are in fact *becoming* the business.

Burton Group first referred to this concept as the virtual enterprise network (VEN) in the *Identity and Privacy Strategies* overview, "The Network Services Model: New Infrastructure for New Business Models," published in 1999. In summary, Burton Group defines the VEN as a flexible IT environment that supports real-time responsiveness and enables business processes to seamlessly cross logical and physical boundaries. Today, emerging examples of the VEN can be found in every segment of business in which large companies compete.

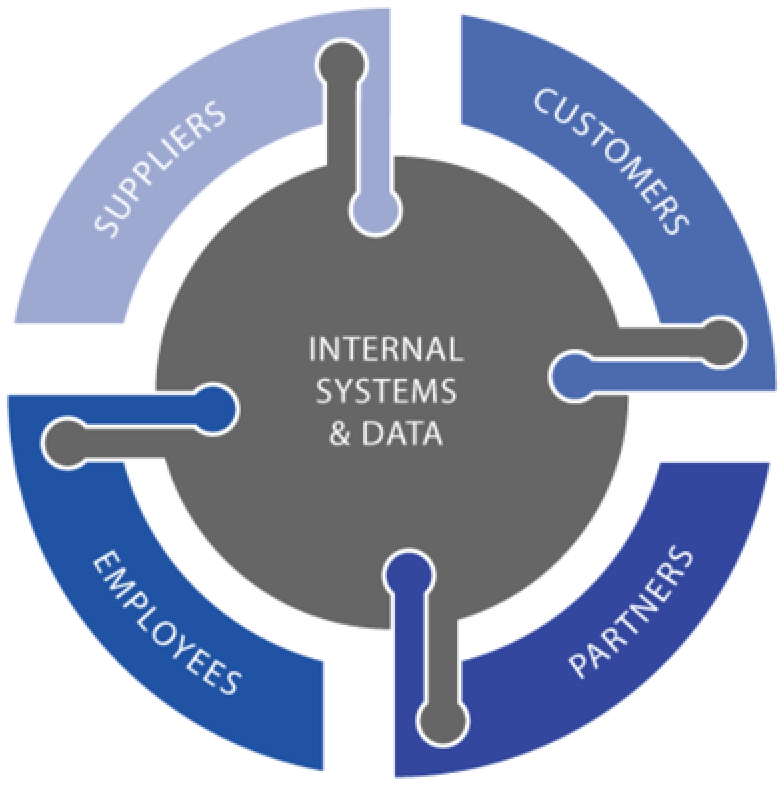


Figure 1: *The Virtual Enterprise Network*

In reality, however, IT systems are often more impediments than facilitators of change, thus making the full potential of the VEN difficult to realize. Changes to a business process can take months to implement, and the opportunity costs are staggering. For most enterprises, then, seamless, dynamic integration with customers, partners, and suppliers remains elusive. The NAP represents a clear change in application architecture—one that is designed to address many of these problems. Before discussing the NAP in detail, then, it's important to understand the business drivers behind its ongoing evolution.

Long-Term Drivers

Simply put, the VEN itself is an economic driver for a new, network-centric application platform. In spite of the difficulties involved in doing so, enterprises are starting to replicate on the network the very processes they use to run their businesses in the physical world. They're using IT to improve business process, boost productivity, and increase customer satisfaction. And they're doing so because today's business environment simply demands it.

Many organizations are using internal portals to aggregate information and functionality into systems that conform to business process. Other organizations are working to integrate their product offerings into customer processes in an effort to decrease costs and increase efficiency and service. A large manufacturer is creating a portal for its retail dealers that allows them to more easily manage inventory, order parts, and other key functions, for example. A large retailer Burton Group works with has begun requiring many of its suppliers to connect directly with its procurement and inventory management systems. Many enterprises are increasing customer satisfaction through self-service interfaces into core systems. Such efforts are increasingly common in every client Burton Group works with. Customers are demanding more from their suppliers, and suppliers are feverishly competing to meet those demands.

The Promise of the Net

Clearly, the idea of process-oriented integration architectures is not new. Integration middleware has long promised process-oriented solutions to business problems. And during the dotcom boom, IT departments were chartered with enabling real-time responsiveness. Business needs demanded that IT reduce friction and incompatibility in their internal business processes and in their interactions with partners, suppliers, vendors, and customers, often prioritizing these needs over cost control.

Today, however, business requirements and reality assessments have changed on multiple fronts. Certainly “cost is no object” innovation is a thing of the past. But just as important, the realities of today's computing infrastructure are clearer. While the dotcom boom certainly generated more than its share of harebrained schemes, many of the basic ideas for leveraging the Internet for business that came out of the dotcom boom were not without merit. Application service providers, software as a service, and any number of e-business scenarios were, at least at the conceptual level, valid ideas that will one day see real-world implementation.

But much of the Internet's potential to enable those ideas remains untapped because the infrastructure doesn't support the requisite functionality. In short, the existing Web architecture has been taken about as far as it can go. To drive further innovation, as well as the profits and productivity improvements that go with it, solutions must be found for the hard problems associated with distributed computing. Resolving problems such as process definition, integration, and automation; data interchange; interoperable transactions; and strong, federated security will make it possible to build the infrastructure that supports not just high-value transactions, but a new generation of network applications and services. Only then can we truly turn the network into the computer, creating a network platform and realize the full potential of the Internet.

Short-Term Drivers

Businesses are demanding innovation more than ever before. At the same time, however, most enterprises are working to rein in IT costs. Where innovation was a “cost is no object” imperative in the dotcom boom, today IT departments no longer have the luxury of funding expensive initiatives. In short, enterprise IT architects are being asked to innovate on a constrained budget, which means they must figure out how to use what they already have more cost-effectively.

The need to increase both the reach and range of today's integration solutions while simultaneously driving down the costs associated with implementation are the engines driving the current industry trends toward standardization, software commoditization through open source, and, ultimately, the evolution of the NAP.

In order to respond to business demands for real-time responsiveness, for example, IT departments have deployed a plethora of proprietary integration solutions, such as enterprise application integration (EAI), enterprise information integration (EII), business-to-business integration (B2Bi), and integration brokers. These solutions, most of which communicate using proprietary protocols, typically require expensive deployments of a specific vendor's software throughout the environment. These solutions also typically entail expensive professional services contracts. The integration solutions require special skills in order to implement connectors to existing application functionality and to design the workflows that tie disparate systems together. And even when deployments go exceedingly well—which isn't as often as customers would like—interoperability does not scale to the any-to-any heights that today's businesses require.

These issues are significant impediments to business agility. To address these issues, enterprises need the lower costs, ease of implementation, and scalability that only commoditization and standardization can bring. It's no coincidence that these tools and standards are emerging now, just as the business needs that characterize the VEN are reaching critical mass. Application service providers (ASPs), software as a service, and dynamic relationship management are all examples of functionality promised in the Internet boom, for example. While these are clearly desirable goals, however, they're difficult, if not impossible, to achieve based on the existing network infrastructure.

The NAP, then, is a convenient way to name the pervasive infrastructure necessary to meet the integration needs of the virtual enterprise. It describes the architecture, standards, infrastructure services, and methodologies necessary to enable not just a standardized approach to application integration, but a new generation of applications that is more capable of exploiting the full potential of the Internet.

Why Now?

Over the last 20 years, a variety of vendors, standards bodies, and very smart people have made many attempts to solve the application integration problems. Since none of those efforts resulted in mainstream standards and solutions, it's fair to ask why the NAP, the WSF, and SOA will succeed where so many before them have failed.

The Common Object Request Broker Architecture (CORBA), for example, provided a common framework that could have, in theory, bound all OSs and platforms into a standardized environment, thus solving the integration problem through homogenization. Similarly, the distributed computing environment (DCE) defined a fully distributed and interoperable environment, at least in theory. In practice, however, CORBA, DCE, and other attempts never lived up to their promise for a variety of reasons.

Neither DCE nor CORBA found the mass-market vehicle it needed to create critical mass and attract large numbers of developers, for example. Both also suffered from complexity, scalability, and interoperability problems that stymied developers. More important, while a relatively small number of large enterprises yearned for integration and standards, powerful market dynamics did not encourage vendors to standardize. The problem did not affect a large enough segment of the market to drive mainstream solutions. In fact, market dynamics encouraged vendors to create beachheads of control within key corporate accounts.

When Markets Demand It, It Will Come

As businesses worked to integrate their internal systems—and in some cases, to connect systems between companies—requirements for enterprise integration drove attempts to create middleware standards. However, these efforts still failed to solve the integration problem in a comprehensive fashion. While there are many different flavors of remote procedure call (RPC)-based middleware, for example, their effectiveness is usually limited by platform or programming language dependencies. Message-oriented middleware (MOM) and EAI offer some relief, but only when both applications use the same integration solution.

Interoperability and integration dynamics began to change much more quickly and fundamentally with the rise of, first, the World Wide Web, and second, Extensible Markup Language (XML). Suddenly all companies, not just the small number of very large companies that could afford one-off solutions, could communicate with each other. And the market demand for interoperability and integration across traditional boundaries—application, OS, platform, and organization—began to grow exponentially. Today, market dynamics are driving vendors inexorably to provide the interoperability that enterprises need.

Pragmatic and Realistic Architectures

Market dynamics weren't the only problems that prevented CORBA, DCE, and other solutions from solving the interoperability problem. From an architectural point of view, CORBA, DCE, and many other attempts to solve the problem were predicated on tightly coupled architectures. Tightly coupled systems require high degrees of symmetry between endpoints. To achieve that level of symmetry, individual developers, teams of developers, business units, and both large and small companies alike all over the globe must agree on not just which technologies to use, but on deep implementation details such as object models, tools, and approaches to building applications. It's obvious that such widespread agreement is not just impractical; it's simply unrealistic and impossible.

Loosely coupled architectures take the opposite approach. Instead of dictating how all systems implement a function, loosely coupled architectures define rules that bind autonomous domains to a common method of exchanging information regardless of which internal systems they use. Consequently, loosely coupled systems eliminate the need for interoperating parties to understand *a priori* the internal details of each other's IT systems. Heterogeneous enterprises can disagree over which technology to deploy, as well as the meaning, ownership, and schema of information. The loosely coupled architecture focuses on defining common data formats and protocols that make information and functions portable across domains, allowing systems to interoperate.

The Web services framework leverages XML to enable a more flexible and dynamic architecture that doesn't dictate symmetry between endpoints. It's independent of underlying object models, programming languages, and all application platforms. Because of its loosely coupled architecture, the Web services framework has a much better chance of succeeding where others, such as CORBA, have failed.

The combination of a practical, realistic architecture and a much larger market demand to solve the problem creates a much higher probability of success for the NAP, SOA, and the WSF. The alignment of market need and reasonable architecture has generated a high degree of cross-vendor support for the WSF, for example, thus creating a much higher likelihood that dissimilar applications can interoperate. It will take years for all the necessary standards to fall into place, find their way into products, and for those products to gain momentum in the marketplace. But the trends are clear.

The Changing Market

The emergence of the NAP has broad implications for many segments of enterprise IT, as evidenced by the following trends:

- **The march to SOA:** As the transition to SOA continues to pick up speed, enterprises will apply SOA “wrappers” to many of today's monolithic applications. Increasingly, enterprises will implement new applications by decomposing them into reusable components that can be recomposed as business needs dictate. To keep up with this change, enterprises will have to wrestle with deep cultural and architectural changes in how they build and deploy applications. They also need to learn how to recognize the applications for which SOA is the right choice, and those for which it is not the right choice.

- **The evolution of the Web services framework (WSF):** Because it establishes standards that enable new levels of integration and interoperability, the WSF is the foundation for the NAP. Vendors will fight over many components of the WSF precisely because it's strategically important. And some of those fights will get nasty. But open standardization of the WSF is a basic requirement, driven by the demands of the market. Over the next three to five years, the WSF will continue to evolve and will eventually address tough distributed-computing problems, including transactions, workflow, and security through a federated model.

- **The continuing evolution of general-purpose infrastructure:** Core infrastructure services such as security, which are today hardwired into many applications, will continue to emerge as discrete, general-purpose services that the NAP exposes in a standard fashion. The evolution of these services will create a better division of labor between the small number of systems-level programmers who can implement complex technologies, and the much larger number of business developers who need to leverage them.

- **Changing development models, changing tools:** Development will become increasingly model-driven and policy-based. Platform vendors and open source initiatives will strive to create “complete solution” tool environments that support both. While specialty tools will continue to find their niches, the market at large will be dominated by tools that focus on helping developers refactor, compose, and recompose services.

- **Redefining the platform market:** Traditional platform products, including OSs, application servers, portals, DBMSs, and integration middleware, will undergo substantial change over the next five years. Platform products such as DBMSs, application servers, and portals will converge into a new generation of platform products that will redefine markets. Managed code architectures, the commoditization of the infrastructure through open source, and richer presentation services will enable a new generation of applications, reduce costs, and change the dynamics of the enterprise software business.

A Long but Inevitable Transition

As these trends indicate, the NAP has enormous potential, meeting the need for a more flexible and adaptable IT infrastructure—one that is capable of responding to the dynamics of day-to-day business. But we're in the very early stages of a long but inevitable transition to the pervasive, network-centric platform outlined in this overview. And technology alone will not enable that full-scale infrastructure; a variety of factors must converge to meet the need for more flexible application architectures.

From the technology side, standards, frameworks, tools, and services—the basic building blocks of the platform itself—must evolve before applications can take advantage of them. From the operational side, enterprise application architects and developers must adopt a new mindset, both in terms of how they build applications and the division of development labor necessary to support the needs of the business. Increasingly, core developers will build business logic and infrastructure services as reusable components, while business developers will act more like building contractors, by assembling (and then remodeling) process-oriented systems using standardized services as building materials.

Clearly, then, we're many years away from the full realization of the concepts we define as the NAP. The evolution will be painful at times, as vendors battle over key standards and markets. But now that the tectonic plates have begun moving, there's no stopping them. While we're in the early days, then, it's clear that the move to a more flexible architecture for building truly networked applications—by whatever name it is called—has begun in earnest.

The Clear Imperative

Given these realities, today's IT managers must understand the NAP not as an off-the-shelf solution or product, but as an architectural goal state. As a term, the NAP is simply a convenient handle for the concept of a pervasive infrastructure that increases the power of existing applications by allowing them to participate in a more meaningful whole. While its definition will evolve over time, the NAP is a destination that, in order to meet the needs of the business, organizations should always be in the process of reaching. That means enterprise application architects must start laying the foundations for the NAP today. They must start creating the infrastructure, changing mindsets, and planting the seeds of change.

Yesterday's monolithic application architectures don't provide the flexibility and adaptability that businesses need. They stymie integration efforts over the long term, and, ultimately, increase costs. They also leave an organization poorly prepared for the future by making it more difficult to integrate IT with business process and respond to changing business dynamics. As acknowledged earlier, the NAP and SOA are far from panaceas, and thus aren't the solution for all problems. But enterprises can meet rapidly changing business needs by judiciously using the concept of the NAP, implementing service-oriented architectures, and leveraging general-purpose and reusable infrastructure services, thereby creating more flexible application architectures.

As it evolves, the NAP will enable enterprises to deploy new applications, services, or business models more rapidly. The NAP can reduce the high degree of friction surrounding today's integration efforts, thereby reducing costs. It will also put an organization on the path to better integration between IT and core business processes. As an increasing number of enterprises take that path, the NAP will emerge as a pervasive infrastructure, within, between, and across organizational structures. And the enterprise's efforts to instantiate, manage, and maintain business relationships and processes within the network will increase in both reach and range.

Burton Group started the Application Platforms Strategies service precisely to track, encourage, and facilitate the development of the NAP. Over the coming years, Burton Group will be covering the wide range of issues summarized in this overview in more detail, helping clients to understand the dynamics driving the NAP's evolution, the concepts and functions behind it, and how it affects their organizations.

The Details

Like many other technology terms, “platform” has been so overused that, at this point, it's easy to ignore it as just another buzzword. But it's a useful term in that it implies a foundation, which is a necessary element of any architectural metaphor. Consequently, we define the term here not to espouse what we see as a canonical definition that we want to foist on the world, but rather to make our intentions and meanings clear. Just as we define the network application platform (NAP) later in this document, an understanding of how we're using the term “platform” will make that definition all the more clear.

For the purposes of this discussion, then, the term “application platform” describes an integrated, self-contained environment for hosting and building applications. By definition, then, the platform must encompass both the runtime environment where applications execute, and the tools and services necessary to build the applications that run in that environment. A platform is defined by a set of underlying infrastructure services and functionality that developers can reuse instead of recreating as they build applications. An operating system (OS) is a platform, for example, that gives developers access to a set of general-purpose services, such as processing, memory management, storage, and user interface subsystems. Typically, the platform will make those services and functions available via application programming interfaces (APIs). The value of a platform is often determined by not just its technical architecture, but the quality and number of tools available for building applications on the platform, the market for supporting services that has grown up around the platform, and the market for developers that is created by the installed base of the platform.

Example platforms include Java, Microsoft .NET, and the open source scripting platform known as LAMP (Linux, Apache, MySQL, and PHP/PERL/Python). These platforms typically package their APIs with a rich set of class libraries, tools, and wizards, which provides a usable and consistent framework that automates the use of the underlying infrastructure. Figure 2 illustrates an application platform.

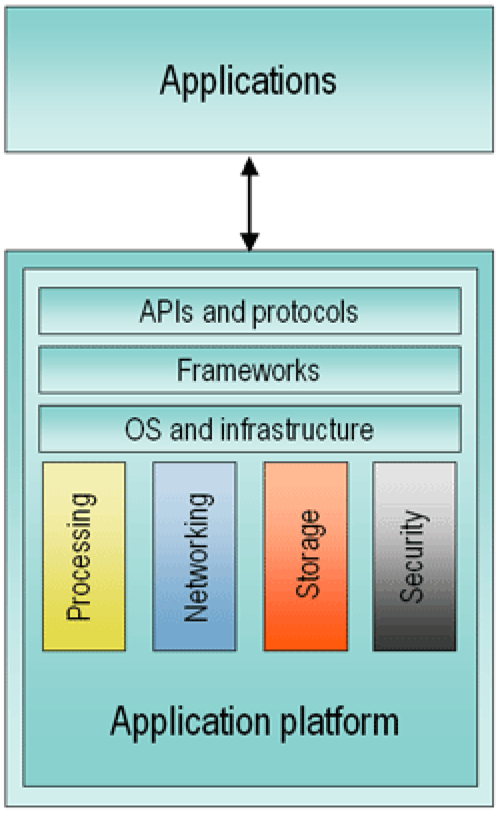


Figure 2: *An Application Platform*

Platform Evolution

Over time, application platforms have grown in terms of not just the quality and type of services they offer, but also the manner in which they provide them. In many ways, it's useful to think of the services that are provided by the platform as being “south” of the API, while the services, applications, and tools that third parties provide as being “north” of the API. Over time, the API has naturally migrated northward, and today's platforms contain far more functionality than they did in the past.

Thirty years ago, for example, the frameworks within an application platform were small and basic. For the most part they consisted of a programming language, compilers, and a set of APIs that supplied access to core OS services for processing, memory allocation, input/output (I/O) devices, and networking. These early platforms did very little to insulate the developer from the underlying OS. Developers were responsible for writing huge amounts of system-level code, or “plumbing,” in order to make the business logic work.

Most modern application platforms now provide a broad set of frameworks and infrastructure services that support simplified development of distributed application systems. In addition to a language, compiler, and APIs, a modern application platform also includes:

- Visual design and development tools
- Wizards and code generators
- Deployment and code distribution systems
- Management facilities
- Web servers
- Web services frameworks
- Portal servers with personalization features
- Collaboration systems
- Application servers
- Integration brokers
- Transaction management systems
- Messaging middleware
- Database servers
- Directories
- Security frameworks
- Many other frameworks and services

Platform Integration

Just a few short years ago, all of these tools and frameworks were packaged as separate products. In many cases, they still are. But the trend is clear: Increasingly, these tools and frameworks will be packaged within the platform, thus redefining what a platform is and providing a set of integrated building blocks. These frameworks provide layers of abstraction between the application development environment and the underlying OS, and they automate a lot of plumbing code. For the most part, developers can concentrate more on building business logic and less on making the logic work effectively on a given platform.

By embedding the infrastructure functionality, the application platform can ease application development, increase developer productivity, and increase integration and interoperability within the environment the platform defines. The frameworks provide a set of integrated building blocks that accelerate the development of applications. Frameworks also improve the quality of code because they automate complex programming, such as access control and transaction demarcation.

Applications developed using frameworks also tend to implement infrastructure policies more consistently. In many cases, for example, enterprises leave the correct and consistent implementation of complex functions such as access control and transactions to the discretion of developers. Given the complexity of these functions and the varying skills of developers, implementations are not only inconsistent, but can also contain bugs and vulnerabilities that create management nightmares. Frameworks can automate the development process by giving developers a standardized set of functions to use in specific areas. And developers who understand these complex functions can focus their efforts on making sure the framework implementation is not only consistent with corporate policy, but as bug-free as possible.

Integration Becomes the Issue

Application platforms are the locus for intense vendor competition, which benefited the customer by generating a great deal of innovation over the years. The downside of that competition, however, is that embedded application platform frameworks are typically delivered as proprietary collections of components and services that may be used only by applications developed and/or executed on the specific application platform. Windows applications only run on Windows, for example, and Java applications only run in the Java runtime environment.

Although the rise of integrated platforms and frameworks has eased the development burden in many cases, the proliferation of different platforms from different vendors has created an integration problem. As long as applications could work within the confines of a given platform, interoperability and integration weren't first-order concerns. But integration and interoperability between platforms has become a growing issue as businesses have made increasing use of IT to support their processes.

The Rise of Middleware

During the 1980's and 1990's a market segment known as “integration middleware” developed and flourished. Middleware refers to sets of tools, frameworks, and runtime systems that automate system-level functionality in an effort to minimize the amount of plumbing code that developers need to write. In the early years of this market, middleware referred to database systems and transaction processing (TP) monitors. More recently, though, most people associate middleware with products such as messaging systems and integration brokers.

In recent months, both Microsoft and Sun have stated that the middleware market is dead. On one hand, these statements are pure marketing hype; the increasingly heterogeneous nature of the world ensures that there will always be a market for integration products. On the other hand, however, it's equally clear that the *kind* of middleware necessary to facilitate integration will change dramatically. What we currently think of as middleware functionality—messaging systems and integration brokers—has begun its inevitable migration, moving “south of the API.” But rest assured—independent vendors will continue to innovate and produce new types of middleware products. And over time, as the new middleware functionality goes mainstream, it too will eventually work its way into the underlying application platform.

Network Platform Prerequisites

For many years, vendors have talked about network computing, network-enabled applications, and distributed computing. Sun is famous for boasting the tagline The Network Is the Computer, for example. In reality, however, the network is a disjointed collection of physical devices and heterogeneous application platforms that are very difficult, if not impossible, to integrate fully. Distributed computing theory has been around for decades, and the industry has made many attempts to establish large-scale distributed systems. But neither the business drivers nor the technical infrastructure existed to drive it into the mainstream, much less to the height of Internet standards.

Today, however, circumstances have changed. The business drivers that comprise the VEN are demanding a solution that increases the reach and range of integration well beyond the confines of the corporation. Consequently, it is no longer an academic exercise to consider what it will take to turn the network into a platform—an integrated environment that provides a common set of general-purpose services along with the protocols and interfaces that make those services available. Such an integrated network-centric platform will enable a new generation of applications.

This platform must meet the need for large-scale integration, and it must thus yield tremendous flexibility and power. While it's tempting to dive into the technical complexity of distributed computing theory and systems integration minutiae to describe that power and flexibility, it's wise to first consider the underlying design principles that can ensure large-scale success. These principles have, through the test of time and the failure of alternative approaches, proven themselves to be the only practical approach to creating the NAP.

Given the requirement for real-time responsiveness and cross-boundary communication, the VEN establishes the following basic set of platform requirements:

- Service-oriented architecture (SOA)
- An OS- and language-independent service bus
- Policy-driven management and control
- Federation of infrastructure services

Service-Oriented Architecture

The inertia of most IT systems stems from their application architecture. Most applications are designed as monolithic and autonomous systems. Each application implements a complete business process, and the specifics of the process are hard-coded in the application. Modifications of the business process often require a corresponding modification to the application code.

In many cases, two or more applications need to perform the same business tasks, yet each application supplies its own duplicate implementation of these tasks. If the business rules associated with a specific task change, then every application that implements that task must be changed.

In contrast, SOA defines application functionality as a set of shared, reusable services. Each application service implements a discrete task, and any application that needs to perform that task uses the shared service to do so. An application, therefore, is created by assembling the appropriate services it needs in order to accomplish its business process. Figure 3 illustrates the concept of shared, reusable services.

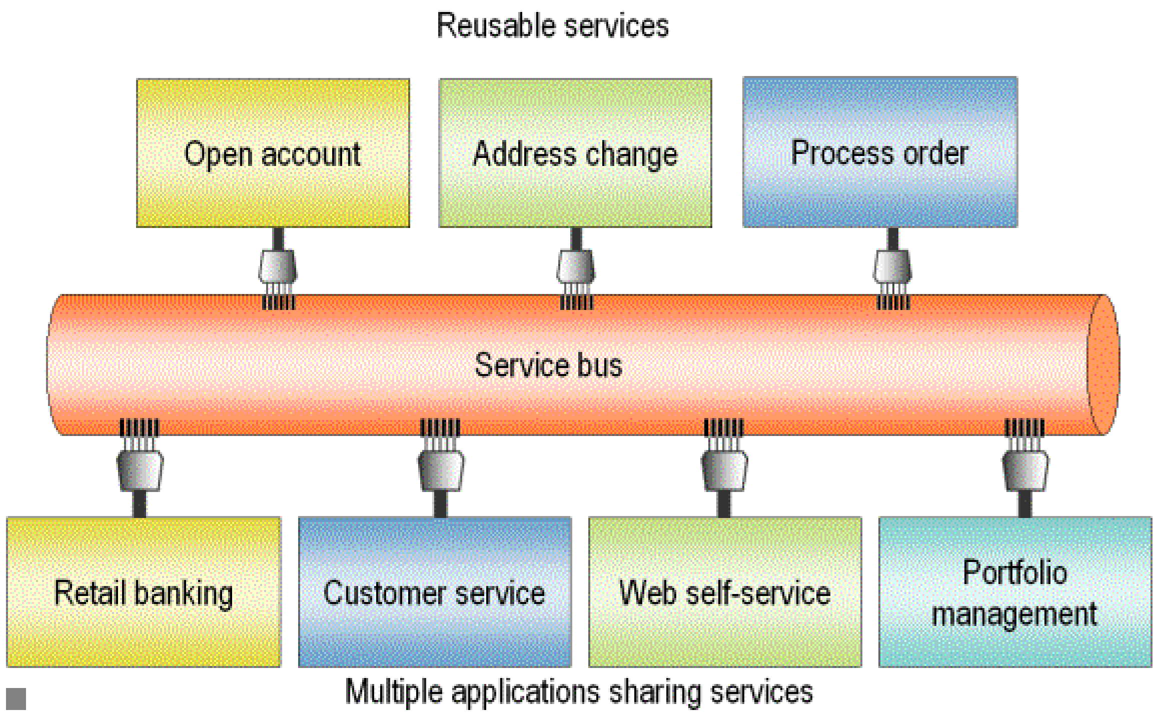


Figure 3: *Shared, Reusable Services*

In this architecture, when the business rules associated with a specific task change, developers must modify only the one service that implements the task. In theory, all applications that use the service will then automatically adopt the new business rules. And when a new opportunity appears, developers can rapidly implement a new business process by assembling a new application from the available services. It will be many years before enterprises can realize the full benefit of SOA. But to fully understand the impact of SOA, it's important to see it as an inevitable outcome of the ongoing evolution of enterprise application architectures.

Evolving Application Architecture

Application architectures have been on a continuous track of evolution since the introduction of distributed computing. In particular, a basic evolving design principle has focused on factoring monolithic application functionality into multiple physical or logical tiers. The clean separation of presentation, business, and data access logic improves the flexibility and maintainability of application code.

In two-tier client/server architecture, for example, an application consists of two physical tiers: a monolithic application client and a database management system (DBMS) server. Note that, from a logical perspective, the DBMS is not actually part of the application. The DBMS manages data access and persistence, but it typically does not implement any application logic. (One major exception to this rule occurs if the application uses stored procedures. A stored procedure is a shared, reusable data access service—a piece of application logic—that executes in the DBMS. A client/server application using stored procedures qualifies as three-tier architecture, which is discussed next.) Enterprises appreciated the productivity features afforded by client/server development tools, but soon found themselves buried under client code that was hard to maintain or reuse. Within the two-tier client/server architecture, the presentation, business, and data access logic is tightly coupled within the monolithic client.

In the 1990's, many companies adopted the three-tier client/server architecture. Here an application is divided into three physical components: a presentation tier that runs on a client device, a server-based application that implements business and data access logic, and a backend database. In this architecture, there's clean separation of presentation logic from business logic, but in most circumstances, the business and data access logic remain tightly bound. Enterprises found the server-based application logic to be easier to maintain, but the architecture doesn't enable high degrees of reusability because the business tier is still fairly monolithic.

By the mid-1990's, the client/server wave peaked, just as the Web application wave began. By the late 1990's, in response to the growing adoption of Web-based applications, architects began thinking of application tiering from a more logical perspective. The industry began using terms such as five-tier and n-tier to describe this perspective. For example, vendors such as Oracle and Macromedia promote the five-tier client/server architecture, as shown in Figure 4.

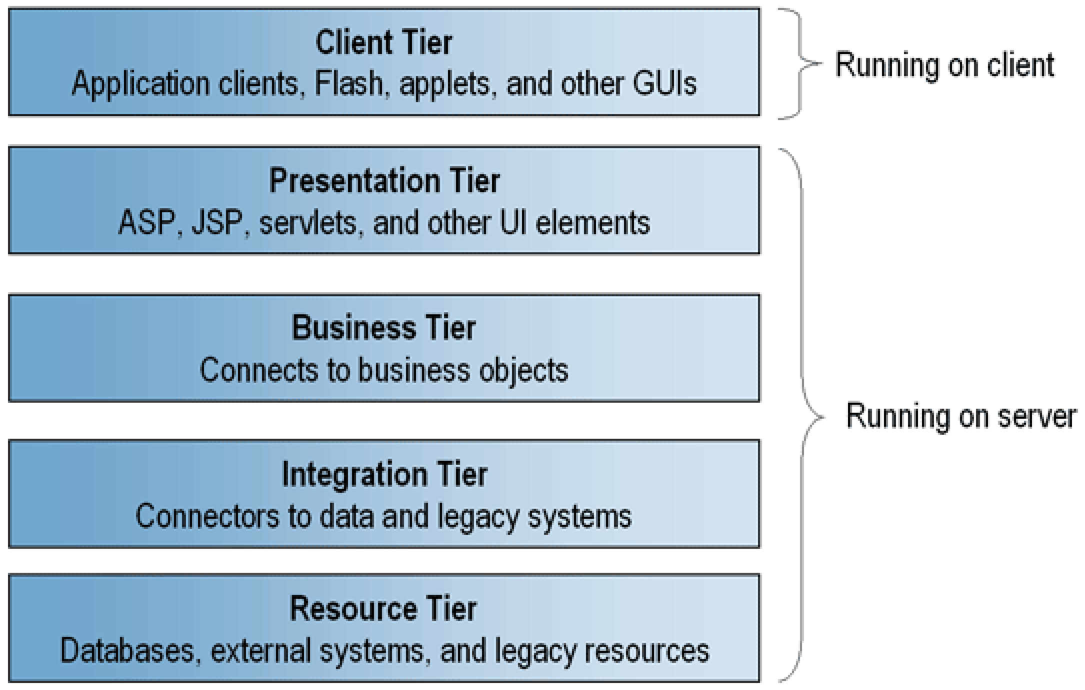

Client Tier
Application clients, Flash, applets, and other GUIs
Running on client
Presentation Tier
ASP, JSP, servlets, and other UI elements
Business Tier
Connects to business objects
Integration Tier
Connectors to data and legacy systems
Resource Tier
Databases, external systems, and legacy resources
Running on server

Figure 4: *The Five-Tier Client/Server Architecture*

Five Tiers, Tightly Coupled

The five-tier client/server architecture focuses on a clean separation of concerns within the application. It recognizes the need to separate user interface (UI) display processing (client tier) from UI generation and control logic (presentation tier), and the need to separate business logic (business tier) from data access logic (integration tier). The tiers in this architecture are logical and don't necessarily imply any kind of physical deployment topology. The various tiers may be deployed all on one system or scattered across any number of systems. The tiers can run on desktops, in application servers, in integration brokers, or in the DBMS, depending on what's optimal for the application. The logical tiers include:

- **Client:** The client tier renders the application's UI and performs interactions with the user. In many circumstances, the client tier simply displays content that is dynamically generated at runtime by the presentation tier. The client tier typically runs on some type of client device, such as a mobile handset, desktop, or user workstation. It may run within a browser or as a desktop application, using technologies such as Hypertext Markup Language (HTML), Flash, Java, and .NET.

- **Presentation:** The presentation tier generates and personalizes the UI and controls the flow of the application. It responds to events generated by the UI and invokes the appropriate functions in the business tier. The presentation tier typically runs in an application or presentation server. It may be implemented using technologies such as Active Server Pages (ASP), JavaServer Pages (JSP), servlets, portlets, a rich interactive application (RIA) presentation server, or similar user interface system.

- **Business:** The business tier manipulates business objects and implements business algorithms. It relies on the integration tier to map business objects to various persistent data resources, such as databases, message queues, legacy applications, documents, and files. The business tier often runs in the same application server as the presentation tier, although the two tiers may be segregated to support better scalability. The business tier may be implemented using technologies such as ASP, Microsoft Transaction Server (MTS), servlets, Enterprise JavaBeans (EJB) session beans, other container-managed objects, and simple classes.

- **Integration:** The integration tier provides access to data resources, and it manages data query, aggregation, and persistence operations. The integration tier often runs within the same application server as the business tier, although it may run as an external resource adapter or within the target data resource. It may be implemented using technologies such as Active Data Objects (ADO), Microsoft Data Access Components (MDAC), Java Data Objects (JDO), Java Database Connectivity (JDBC), J2EE Connector Architecture (J2CA), EJB entity beans, various object/relational-mapping frameworks, stored procedures, Java Message Service (JMS), and various legacy application adapters.

- **Resources:** The resource tier consists of databases, message queues, legacy applications, documents, files, and other data resources. These data resources may be scattered throughout the enterprise.

The five-tier architecture affords more flexibility in terms of component deployment topologies than previous application architectures, but the design still imposes limitations in terms of reusability and cross-domain interoperability. These limitations stem from a design mindset focused on building single-purpose applications using tightly coupled connections between each tier.

Most developers find it much easier to implement tightly coupled connections than loosely coupled connections, for example, especially when dealing with transactions. That mindset, in turn, stems from the fact that the tools, combined with a desire to deliver interactive, real-time data to end users, make it difficult to do otherwise. Tightly coupled connections permit the presentation tier to initiate a transaction and propagate it through the three back-end tiers. In other words, the presentation tier can hold and manage locks in the resource tier.

Unfortunately, the tight coupling between the tiers results in the equivalent of a monolithic application. Each tier tends to be highly dependent on the other tiers within the application. And in many cases, the only supported interface into the application is through the presentation tier. The intermediate tiers typically aren't accessible to other applications, and only the resource tier is available to other applications. (A database is a reusable service, after all.)

On the other hand, based on current prevalent design practices, the business and integration tiers—the components that implement most of the application's business rules—typically are not designed for reuse. They are designed to support one specific application. One exception to this trend occurs when the integration tier has been implemented using stored procedures. Developers rarely define separate stored procedures for each individual application. In other words, a stored procedure is *designed* to support reuse. To maximize flexibility and reuse, all application logic—not just data access logic—should be designed with reuse in mind.

SOA Design Principles

SOA is a style of application design that focuses on reuse. It enables flexible, adaptable, distributed-computing environments. SOA defines a set of principles and practices for designing application functionality as shared, general-purpose services. Each service implements a particular business task and operates relatively autonomously. Developers can compose and recompose these services into orchestrated applications that implement a variety of business processes. In combination with the proper infrastructure, SOA can allow IT systems to rapidly respond to changing business conditions. (Please see the *Application Platform Strategies* overview, "Service-Oriented Architectures: Developing the Enterprise Roadmap," for a thorough analysis of SOA.)

The real distinction between client/server architectures and SOA is in terms of the design mindset. In client/server architectures, the focus is on building a specific application and factoring that application into logical tiers. In most circumstances, the developer fully expects to implement the entire application. In SOA, the focus is on building reusable services and then assembling those services to implement a business process. A given service may be used in any number of applications. In SOA, the developer expects to reuse as many existing services as possible and to implement only as much code as necessary to orchestrate the business process.

Refactoring and abstraction are constant themes in SOA. Rather than developing a monolithic application that executes a complete business process, a developer factors the application into discrete, reusable business tasks (services)—each of which may be used in other business processes (applications).

The notion of general-purpose services that can be reused by multiple applications, as opposed to recreated for every application instance, is the essence of SOA. But it imposes a number of challenges. A given service implementation may require different operational semantics in terms of security, reliability, or transactions, for example, depending on the application context in which it's used. In traditional application architectures, developers typically implement these operational semantics *within the application code*, which reduces, if not eliminates, the reusability of the service.

SOA solves that problem by externalizing operational semantics. Instead of baking security semantics (such as identity and entitlement management) into a service, for example, developers rely on an external, general-purpose security framework that enforces authentication and authorization rules based on declarative policies that specify the required security semantics for a particular service within the context of a specific application. Likewise, instead of building their own transaction semantics into a service, developers rely on a general-purpose transaction framework, again governed by declarative properties. Consequently, the ability to reuse the service increases exponentially. As new use cases arise—which may require different security and transaction semantics—external transaction and security frameworks can easily accommodate the new use cases, without requiring changes to the service code.

In this way, SOA changes the division of labor amongst developers by separating concerns. It also reduces development effort and leads to more robust applications. Infrastructure programming requires different skills from business logic development. Business developers should not need to be experts in security, reliability, and transactions. Given the complexity of these technologies, the likelihood that a business developer will make a mistake is frighteningly high. Likewise, security experts should not need to be experts in inventory control or payroll processing. By separating responsibility for business development from infrastructure development, each class of developer can focus on what it does best. Business-oriented developers can leverage the work of more skilled systems-level programmers who understand how to implement complex infrastructure services.

Tiers vs. Peers

In SOA, connections between the services that implement different pieces of application functionality are loosely coupled—meaning a given service doesn't have a strong dependency on another given service in the environment. Typically, service interactions are stateless and nontransactional. (Note that developers must use different development techniques, such as reliable, asynchronous message passing, to manage state and transactions in a loosely coupled architecture.) In SOA, then, applications don't really consist of tiers—at least not quite in the same sense as client/server application architectures define them. Applications are made up of a collection of loosely coupled services. Each service operates as a full peer to other services, and services are connected by a communications bus.

Certainly, it's still an excellent design practice to separate presentation, business, and data access logic, and often these parts of an application will be implemented as separate services. But an SOA application may not require such an ordered tiering of functionality, and the mapping of tiers to services may not be one-to-one. If an SOA application comprises a number of related business tasks, the business tier may be factored into multiple services. Likewise, if the application uses multiple data resources, the integration tier may be factored into multiple data access services. In some cases, a single service may encapsulate multiple tiers. Also, an application may not necessarily have a presentation component. An application may be an automated process that gets invoked in response to a business event. This type of application consists of a controller component that invokes the appropriate set of services to accomplish the desired business process. Typically, this type of application uses an integration broker to manage the intricacies of the process flow.

A service may complete its entire task on its own, or it may call other services to perform various subtasks. A service that calls other services becomes the controller of its subtask. Hence, applications may not conform to a simple tiered architecture. SOA supports complex interactions among the services. An application may comprise a matrix of services working together to execute a business process.

In reality, then, SOA is a design style. It's more about a mindset than it is about technology. First and foremost, a developer must adopt an attitude that it is better to reuse an existing service than it is to build functionality from scratch. In those situations where no preexisting services are available, the developer should implement the service is such a way that other developers may reuse it in the future. Developers must constantly think about reusability and interoperability.

The Tipping Point for SOA?

SOA is currently a popular topic of discussion given the advent of Web services, but it's not a new phenomenon. SOA design principles were first defined and articulated more than 20 years ago. Given the exponential increase of interest in and coverage of SOA, however, it's clear that quite a few people have caught SOA fever. So why has SOA suddenly captured so much attention?

In many ways, the sudden reduction in IT budgets—which is accompanied by a substantial increase in the demand to integrate IT assets—has brought the industry to the tipping point in the adoption of SOA.

A tipping point marks the moment at which a trend shifts from linear to exponential growth. The term comes from epidemiology and indicates the point at which an infection becomes an epidemic. The tipping point concept observes that systems tend to stay at equilibrium—small changes have little or no effect—until the situation reaches critical mass. Then, one more small change “tips” the system and precipitates a large effect.

For the last 40 years, most enterprise IT departments have developed and maintained monolithic application systems based on centralized or client/server architectures. In their day, vertically integrated systems met the needs of the business and, given the business context in place at the time, were generally wise investments. Today, however, these systems are impediments to enabling the VEN. They are resistant to change and stymie innovation.

As stated earlier, the needs of the VEN dictate interoperability and integration on an exponentially larger scale than past architectures can even consider. The scope of the problem pushed the industry to the brink of change. Both the global adoption of the Internet and the development of practical, widely adopted standards (in the form of the Web services framework) were, of course, important precursors that brought the industry to the tipping point. The overall reduction in IT budgets, which generated a demand for cost-effectiveness and commoditization, was the additional, but small change necessary to tip the system and precipitate a large effect: an inevitable and inexorable transition from traditional application architectures to SOAs.

SOA: Security Risks and Rewards

While that transition is inevitable, however, it's important to note that SOA is not a panacea. Like any other architecture it involves tradeoffs, and security is one of the significant factors that will affect when it's appropriate to use SOA.

The Web services architecture eschews the complexity of distributed objects in favor of a document-oriented architecture. In some ways, then, SOA and Web services can help improve security. In theory, loosely coupled environments that rely on asynchronous message passing can be easier to secure than tightly coupled, object-oriented environments that rely on remote procedure calls. Because SOA relies on a policy-based model, service contracts and schema can be validated by firewalls and other devices. Endpoints can use explicit policies regarding what kind of data, transactions, and operations are allowable, while implicitly denying all else. SOA will also make it easier for distributed applications to work together securely across perimeters.

Loosely coupled systems also increase the ability to create enclaves for improved network security. An enclave, or zone, is a logical or physical subspace of an enterprise network. By segregating mission-critical applications and services within an enclave, enterprises can focus on strong controls and monitoring of all inflows and outflows to and from those applications. While attractive, however, enclaves can be hard to enable when application tiers are tightly coupled. Loose coupling can make that segregation and the creation of secure channels more feasible.

The industry has also made progress on a range of Web services security and identity management (IdM) standards, including Extensible Markup Language (XML) cryptography, WS-Security, and the Security Assertion Markup Language (SAML). Products supporting these standards bring basic cryptography to XML, wrap Simple Object Access Protocol (SOAP) messages in security services, and provide XML-based methods for passing user identity tokens between Web services or linking identities in different contexts today. (See the *Identity and Privacy Strategies* overview, "Toward Federated Identity Management: The Journey Continues," for more details on these standards.)

New Risks

But in the short to medium term, Web services and XML-based integration tools actually increase the heterogeneity of the enterprise as they wrapper legacy systems, and as they themselves follow multiple, sometimes incomplete, sometimes conflicting standards or prestandard specifications. Until the standards settle down and SOA digs deeper into the enterprise application and infrastructure fabrics, SOA deployment may actually increase vulnerabilities or at least create new vulnerabilities even as it helps eliminate other old vulnerabilities in the enterprise.

And while Web services and SOA may streamline parts of the security infrastructure, Web services security protocols will create complex cryptographic interdependencies and risk aggregations between security components and between security domains. In fully service-oriented designs, interdependencies between components can raise surety, and accreditation issues, for example. How enterprises validate the distributed components within an SOA, especially when some of those components reside in a partner or supplier's organization, will be a significant issue. And as SOA encourages reuse of services, it will aggregate risk. When a single function is used by many applications, the compromise of that single function can compromise many applications.

The ability to create enclaves is also accompanied by the need for deep inspection of XML content and SOAP headers, which is a significant issue. New security products, including appliances that operate at domain boundaries and standards that create containment and policy enforcement on the network, are crucial for the long-term success of SOA.

While enterprises must clearly manage these risks, however, it's clear that many security and regulatory concerns will drive reuse and the implementation of SOA. By encouraging the reuse of security services, for example, enterprises can address and manage security and regulatory compliance issues in one place, and many applications can inherit the benefit of that work. And the ongoing quest for reuse and the separation of concerns in application development, which began with client/server computing, will lead inevitably to widespread adoption of SOA.

The Service Bus

A communications infrastructure that provides access to services is at the heart of the SOA-based environment. This communications infrastructure, or service bus, must provide standard mechanisms to advertise and discover services and to establish connections with those services.

As is the case with SOA, the service bus is not a new concept. Enterprises could build a simple SOA service bus on any middleware framework that supports the necessary register, find, and bind mechanisms. The Common Object Request Broker Architecture (CORBA), Distributed Computing Environment (DCE), Java, and Microsoft Distributed Component Object Model (DCOM) all meet these basic requirements and support service bus functionality. But these middleware frameworks all come with dependencies that have prevented them from achieving the status of a ubiquitous standard. Frameworks such as DCOM are OS dependent, for example, while Java is language dependent. And all of them require a homogenous environment, end-to-end, to enable interoperability.

A number of vendors supply gateway technologies that enable interoperability between disparate middleware frameworks. Some refer to this type of product as an enterprise services bus, or ESB. While ESBs are useful products within a given context, they typically use proprietary message-oriented middleware (MOM) as their core communications infrastructure. These proprietary protocols impose a requirement for product symmetry that is contrary to the requirements of the NAP service bus. It's the standardization of the bus and its connectivity, reliability, and security features that is an essential requirement for large-scale interoperability.

In short, the needs established by the VEN make it clear that the NAP must innately support heterogeneous systems and nonproprietary protocols. It requires integration with application systems that are not under the control or jurisdiction of the IT department. Consequently, the NAP must support a communications infrastructure that is independent of vendor, OS, and programming language—one capable of connecting heterogeneous systems. In other words, the NAP must support a standardized service bus.

Policy-Driven Management and Control

In a loosely coupled environment where services act with relative autonomy and support reuse, the rules that specify the requirements for working with a particular service implementation must be widely available. Because they may change depending on the application context, the policies associated with a service cannot be hard coded into the service. Instead they must be specified using a declarative mechanism, and they must be implemented at runtime using some type of runtime framework. To meet the needs of the VEN, then, the NAP must support dynamic enforcement of operational semantics based on declarative policies.

Declarative policy-driven management and control deliver a number of additional benefits, including centralized policy control and improved consistency in policy enforcement. Centralized control ensures that security, management, and transaction policies are consistently applied to all applications.

Policy-Driven Development

Just as important, or perhaps more so, policy becomes the foundation for application development in the SOA. Because the SOA makes general-purpose infrastructure services shareable, developers can declare, through policy, what services must run, and how they run, in conjunction with their applications. Thus, services must be able to publish policies related to their use, and those policies must be accessible to both applications and other services.

That makes network-based containment and interception models an important requirement of the NAP in general, and the service bus in particular. Containers (such as a virtual machine) enable policy-based application management. Developers can specify declarative policies (such as security requirements) and at runtime the container automatically enforces those policies on behalf of the application, invoking any necessary services. By moving such containment and interception models off application servers and into the network fabric, the NAP can support dynamic policy enforcement, governing operational semantics based on application-specific policies across distributed systems. Developers, then, can use declarative programming techniques to define the policies that determine what platform services their applications invoke, and how they do so. (We discuss these containment and interception models in more detail later in this document.)

Federation of Infrastructure Services

Given the need to connect heterogeneous system, the service bus must extend application interoperability and integration across the logical and physical boundaries between domains. (We define domains here as areas of control that manage infrastructure issues such as identity, transactions, provisioning, and the like.) Some of these boundaries are imposed by infrastructure systems, such as firewalls, security systems, transaction managers, and management frameworks. At the same time, application integration must transcend the differences in operational semantics within different domains, including object models, OSs, programming languages, interfaces, and other service- and application-specific implementation details.

These implementation differences are, in fact, one of the most significant barriers to cross-platform integration. Most applications use platform-specific infrastructure functionality to implement advanced operational semantics, such as security, reliability, and transactions. Most application platforms supply comprehensive APIs and frameworks that simplify or automate the use of the infrastructure. The frameworks enable tight integration within that platform, but offer little if any means to integrate or interoperate with other application platforms.

Tightly Coupled Systems Don't Scale

Most previous efforts to solve these problems have attempted to dictate a homogeneous approach. Efforts such as CORBA and DCE attempted to dictate not only the overall architecture, services, and tools that developers should use, but also the implementation details regarding how they should be used.

These tightly coupled approaches, then, attempted to dictate high degrees of symmetry between end systems. But as history has taught us, such approaches are just impractical. They're diametrically opposed to the ways in which companies do business. They're in direct opposition to the paths that innovation, business development, and system implementation have always taken in enterprise environments.

Simply put, individual domains—be they discrete business units, ad hoc working groups, or entire divisions of a much larger company—must be free to implement systems as they see fit. Given accountability for meeting a business objective, a domain must be given the authority to implement the systems necessary to meet the objective. At the same time, the rapid development and arrival of new technologies makes the world more heterogeneous, not less so. The trick to scalable, heterogeneous interoperability is to find a way that allows individual domains of development to evolve in a fashion that meets the objectives of the domain, while at the same time enabling meaningful connection with the outside world, starting with other internal domains, extending to the enterprise as a whole, and, ultimately, working with the world in general across the Internet on an as-needed basis.

Defining Federation

Enabling meaningful connections across domains is precisely the goal of so-called federated systems. Federation enables a more loosely coupled, decentralized architecture for integration between discrete systems. Federation defines interoperability protocols that allow domains to exchange crucial information in a just-in-time fashion. It defines data formats and exchange protocols that make crucial information portable across different application and service environments.

Federation is the agreements, standards, and technologies that bind autonomous domains to a common method of exchanging information regarding operational semantics while allowing those domains to implement those functions internally as they see fit. These standards and agreements enable interoperability by making the necessary information portable.

Federation transcends the problems associated with tightly coupled systems by:

- Allowing heterogeneous enterprises to disagree over which technology to deploy, as well as the very meaning, ownership, and schema of information
- Eliminating the need for federating parties to understand a priori the internal details of each other's IT systems
- Defining rules that bind autonomous domains to a common method of exchanging well-understood information regardless of which internal systems they use

Instead of dictating how all systems will implement a given function, federation focuses on dictating how systems exchange information so that they can properly coordinate the function, regardless of either system's internal architecture. Typically based on message-passing protocols, federation standards simply create an I/O mechanism that allows any given system to output information in a given format and to then assume that any other communicating system can consume that format. The first system needs to have no further knowledge of how the connecting system works.

For example, federation protocols related to security define standards for exchanging user authentication and authorization information. And federation protocols related to transactions define standards for exchanging state information, allowing multiple transaction managers to cooperate in the management of a transaction that crosses organizational boundaries.

Making Federation Work

Given the failure of tightly coupled systems to enable large-scale integration, the NAP must support the federation of infrastructure systems at critical junctures. That's precisely why the Web services framework and SOA are both so closely associated with the term “loosely coupled”; the industry has learned through hard experience that federated models are the only practical answer to the need for large-scale integration. The increasing ubiquity of XML, along with standards like SOAP for moving XML across networks, have made federation protocols a much more practical goal, and thus the NAP relies heavily on these emerging technologies.

It's equally clear, however, that federation is a more challenging task than simple integration. Federation hinges on the notion of bridging domains. In order to achieve federation, then, separate domains must establish trust relationships, common protocols, and governance structures that permit them to coordinate their activities. Therefore federation requires some type of cross-domain interoperability framework within which autonomous entities honor each other's decisions and trust each other's assertions. Federation also requires much more than technology. Legal frameworks, administrative structures (such as dispute resolution), and management systems are all necessary components. To ensure the integrity of the environment, then, the NAP must support federation in a way that does not make the environment too complex to manage.

Defining the NAP

Burton Group introduced the concept of a network-based application platform in the *Application Platform Strategies* overview, “The Advent of the Network Platform: Web Services Move into the IT Fabric.” That report describes it as a platform that supports a common set of APIs and a consistent application framework that are independent of any particular programming language or OS.

The NAP is a virtual application platform that enables integration across logical and physical boundaries. The NAP is a virtual platform because it does not provide a complete environment for developing and deploying applications. It does not replace existing application platforms. Instead, it augments these platforms by enabling cross-platform integration and federation.

In keeping with the prerequisites that the VEN defines, the NAP assumes and enables a SOA. (See the *Application Platform Strategies* overview, “Service-Oriented Architectures: Developing the Enterprise Roadmap.”)

The WSF Enables SOA

To facilitate SOA, the NAP relies on the Web services framework (WSF). The WSF is a middleware fabric that leverages a set of open standards based on the Extensible Markup Language (XML). These standards include Web Services Description Language (WSDL), Universal Description, Discovery and Integration (UDDI), and SOAP. Using these standards, the WSF provides an open, vendor-independent, language-neutral middleware framework for application integration.

Unlike any previous middleware framework, the WSF has gained universal endorsement from virtually every software vendor in the industry. Vendors are adding WSF support to their platforms, tools, applications, and infrastructure products. The WSF also works with traditional development models and technologies. For example, Microsoft's .NET technology is tightly integrated with the WSF, and Visual Studio .NET (and the yet-to-be released Visual Studio 2005) supports transparent integration with Microsoft Component Object Model (COM). The J2EE and Java tools vendors supply frameworks and services that enable transparent integration of the WSF with platform-specific frameworks such as Java Remote Method Invocation (RMI), EJB, and JMS. Using various encapsulation strategies, such as Web services-enabled integration brokers and legacy resource connectors, the WSF also enables transparent integration with legacy environments, including MOM, CORBA, DCE, BEA Tuxedo, and IBM Customer Information Control System (CICS).

The WSF takes the Web to the next level, supplying the technology necessary to support real-time responsiveness and cross-domain integration. And perhaps more important, the WSF supplies a foundation for implementing SOA, which is the real key to business agility. Through standardization and commoditization, the WSF will, over the long term, enable the large-scale and economical integration functionality that the virtual enterprise demands. (See the *Application Platform Strategies* overview, "The Advent of the Network Platform: Web Services Move into the IT Fabric," and the Reference Architecture Template, "Web Services Framework Standards.")

NAP Core Concepts

With both SOA and WSF providing foundational elements, the NAP consists of three core concepts:

- **The service bus:** The service bus is a language- and platform-neutral communications infrastructure that enables access to application and infrastructure services both within and between organizations.
- **The infrastructure services model (ISM):** The ISM (formerly known as the network services model) defines a set general-purpose infrastructure services that, when made available via the service bus, enables a new generation of applications.
- **The service design practices (SDP):** The SDP defines a set of design principles and best practices that developers should follow when developing service-oriented applications to ensure flexibility, platform neutrality, and cross-platform interoperability.

Figure 5 illustrates these three concepts and the NAP. The following sections examine these three concepts in more detail.

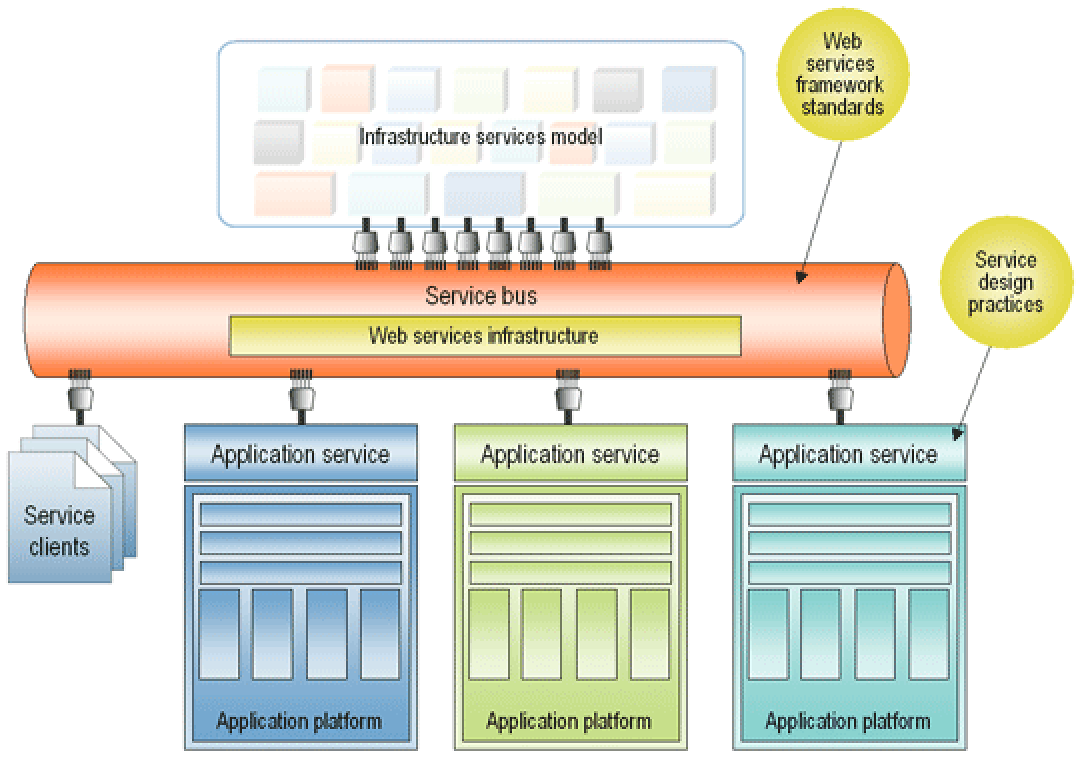


Figure 5: *The Network Application Platform*

The Service Bus

As we said earlier, the concept of a service bus as an integration mechanism is not new. What is new is the degree to which the WSF has initiated the creation of a standardized service bus. By converging on a common encoding format for all types of data (XML) and a common protocol for communicating that data over the network (SOAP), the WSF has created the foundation for a standardized communications infrastructure that can link heterogeneous platforms and systems. This service bus enables a loosely coupled application model by providing a common means for communicating and federating data, transactions, security, and other key operational semantics between systems. Therefore, the emergence of this service bus has profound implications for integration, and serves as the foundation for SOA and the NAP itself.

As a core component of the NAP, the service bus has the following characteristics:

- The bus topology imposes no single point of control or coordination and makes all services equal peers in the environment.
- Any service client, regardless location or underlying application platform, can plug into the bus.
- Any application service, regardless of location or underlying application platform, can plug into the bus.
- The installation of any particular product is not required in order to connect to the bus.
- The bus supports integration with legacy environments.
- The bus supports communications both within and across corporate boundaries.
- The bus supports multiple communication models, such as one-way, request/response, and multipoint message exchange patterns; static and dynamic binding; synchronous and asynchronous connections; stateless and stateful conversations; transacted and nontransacted sessions; and reliable and nonreliable messaging.
- The bus enables cross-platform utilization and federation of infrastructure services, such as security, reliability, and transactions.
- The bus provides managed runtime facilities that implement and enforce advanced operational semantics defined using declarative policies.

As these characteristics indicate, the service bus cannot be implemented using a single product. While enterprises may well install products that help manage and secure it, the service bus is, in reality, a logical construct created through the intersection of many products. As core infrastructure services, enterprise applications, and other products evolve to support the WSF, they will support native connectivity to the bus via Web services standards. Enterprise application architects will create the service bus the first time they connect two dissimilar systems using standard formats and protocols.

Because it satisfies the prerequisite for OS- and language-independent services, then, the service bus also enables SOA in general, and policy-driven management and control in particular. To understand how the service bus meets these core requirements, it's necessary to understand how the WSF supports policy-driven management, and the basics of the SOAP processing model.

The WSF: Policy-Driven

The mix-and-match nature of an SOA offers tremendous flexibility and power, but as discussed earlier, it also introduces new technical challenges. Many different applications might use an account management service, for example, that supports a function such as “open an account.” But each application may require slightly different operational semantics—such as security, reliability, transactions, or auditing—when invoking the account management service.

The WSF inherently supports declarative, policy-driven management and control via the SOAP processing model and the emerging WS-Policy specification. Developed by BEA, IBM, Microsoft, and SAP, WS-Policy defines a general-purpose framework for describing and exchanging information about the rules and policies associated with using a Web service. These policies describe the capabilities, requirements, and general characteristics of the service. For example, policies can define information such as required authentication rules, supported transport protocols, acceptable payment methods, time constraints for response messages, and message-routing rules. A WS-Policy expression is a form of Web service metadata. Policy expressions can be associated with elements in a service's WSDL definition, and they can be associated with a particular Web service end point using WS-PolicyAttachment. A WS-Policy expression can also be registered in UDDI and associated with a service or specification.

WS-Policy expressions specify the requirements applications must meet in order to use a service. At runtime, the SOAP processing model uses a containment and interception mechanism to enforce these policies.

The SOAP Processing Model

The SOAP processing model works like a virtual application container model with a built-in interception framework. It is similar to the way that application server container models, such as ASP.NET and EJB, work. For example, the EJB container model allows developers to define advanced operational semantics, such as state management, transactions, and security, via declarative policies, without having to hard code the implementation of these operational semantics into their applications. The developer simply specifies these policies in the EJB deployment descriptor. At runtime, the EJB container automatically enforces the state, transaction, and security policies on behalf of the application, invoking the necessary services, which are provided by the application server. These services are general-purpose in nature and available to all applications running on the server.

The EJB container's interception model enables this type of dynamic and declarative invocation model. When a client application invokes an EJB, the request is first routed to the EJB container. For each invocation, the container determines which policies apply to the request, enforces the policies, and then forwards the request to the EJB object.

The SOAP processing model supports this same type of interception model, with one very important difference: It doesn't require a centralized application server to enable the containment and interception model. The SOAP containment and interception model operates throughout the service bus, effectively turning the bus—and by extension the NAP itself—into a container for services and the applications that need them. Thus, the SOAP processing model is the foundation for truly turning the network into a platform because it moves the functionality of a containment and interception model out of a single instance of an application server and into the very fabric of the network.

Basic Functions

In the SOAP processing model, two SOAP nodes—a sender and a receiver—communicate by exchanging SOAP messages. A SOAP message may be transferred directly from the original sender to the ultimate receiver, or it may be routed through any number of SOAP intermediaries, as shown in Figure 6. A SOAP intermediary is a SOAP node that acts as both a sender and a receiver. From a NAP perspective, a SOAP intermediary is a service on the service bus (typically within the ISM).

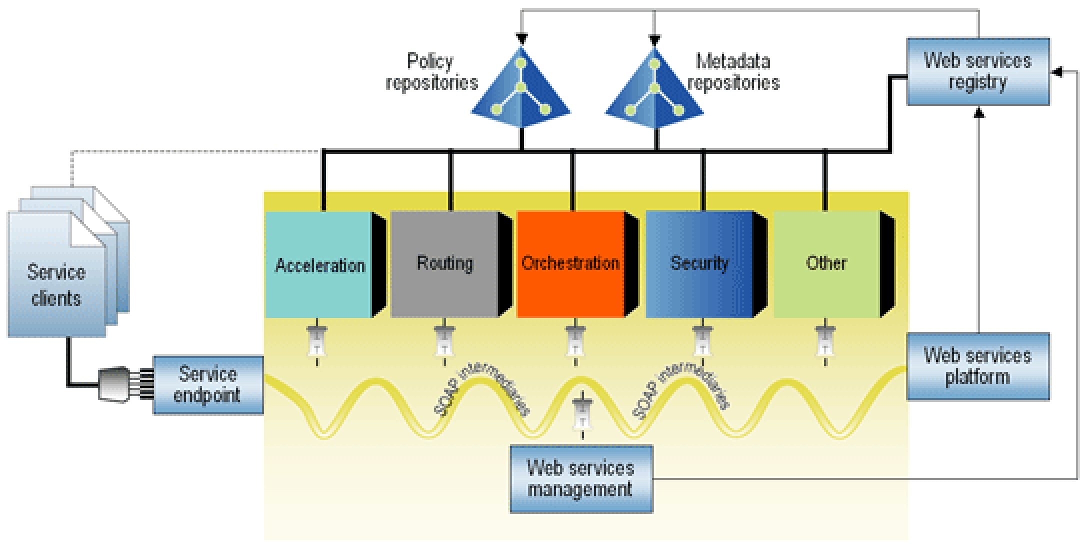


Figure 6: *SOAP Intermediaries*

A service client doesn't need to be aware that its request is being routed through SOAP intermediaries. The service client simply sends its request to the specified service endpoint, which typically represents the first SOAP intermediary in the chain.

A SOAP intermediary can be configured to perform a variety of functions, such as acceleration, routing, orchestration, and security. A SOAP intermediary performs its actions based on directions configured through declarative policies specified in WS-Policy expressions. A WS-Policy expression may contain any number of policies, indicating the processing rules that apply to the service. Types of policies include:

- Authentication policies, such as whether or not authentication is required, and if required, what type of authentication mechanisms or tokens are accepted
- Encryption policies
- Nonrepudiation policies
- Role-based authorization policies
- Auditing policies
- Compression policies
- Routing policies based on network load or message content
- Reliable message delivery policies
- Transaction management and recovery policies

When a SOAP intermediary receives a SOAP message, it determines which policies apply to the message and which policies it is responsible for enforcing. It enforces the policies, invoking any services necessary for doing so, transparently to the application. It then forwards the message to the next SOAP node in the chain. This processing model permits a SOAP message to be routed through a series of SOAP intermediaries, where each intermediary enforces a different set of policies. The *Application Platform Strategies* service will be publishing additional research on policy-driven management and control services in the future.

Implementing Policies Within a Web Services Platform

Most Web services platforms (WSPs) support a set of interception points that developers can use to insert policy control functionality during SOAP message processing. The Java APIs for XML-based RPC (JAX-RPC) API defines a standard interceptor programming model for Java Web services called JAX-RPC handlers. (For more information on how Java supports Web services, see the *Application Platform Strategies* overview, "Java Standards for Web Services: Bringing Synergy to a Fractured Environment.")

A SOAP node determines how to process a specific SOAP request based on configuration information. When deploying a Web service, a developer or administrator typically creates or generates a deployment descriptor for the service. A deployment descriptor provides the means to declaratively assign and enforce policy requirements on a service.

Some WSPs also provide policy administration mechanisms via an administration console. Using the console, administrators can use a point-and-click interface to assign policies governing security, reliability, logging, version control, and other functions to services.

Implementing Intermediaries Using Web Services Management

Developers and administrators typically implement intermediaries using a Web services management (WSM) product. WSM products can perform a variety of functions, such as implementing and enforcing security, ensuring reliable message delivery, content-based message routing, and message transformation. (Please see the *Application Platform Strategies* report, "Web Services Management: Gaining Control of Distributed Services," for more information on WSM.)

The Infrastructure Services Model

Given the nature of truly networked applications, enterprise network infrastructure must provide a robust set of services. SOA is predicated on the notion that instead of building infrastructure into an application, developers can simply rely on generally available infrastructure services within the SOA environment.

In 1991, Burton Group proposed the network services model as a way of describing the core infrastructure services necessary to enable a distributed system environment. In one sense, then, the network services model was a rudimentary description of the foundation for SOA. We have periodically updated the network services model to maintain equilibrium with technology evolution. The most recent updates to the model were presented in the *Identity and Privacy Strategies* overview, "The Network Services Model: New Infrastructure for New Business Models," and the *Application Platform Strategies* overview, "The Advent of the Network Platform: Web Services Move into the IT Fabric." The network services model is illustrated in Figure 7.

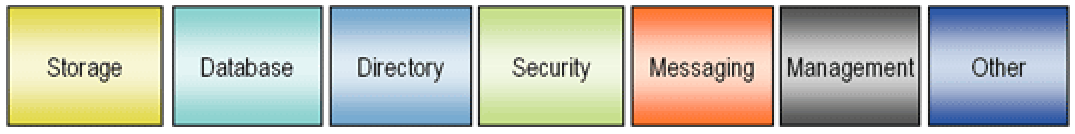


Figure 7: *The Network Services Model*

While it's a useful tool for examining and describing requisite services, however, the network services model is only one of the components necessary to create the NAP. At recent Catalyst Conferences, Burton Group has said that the availability of the core network services over the bus is a basic requirement of any network-centric application platform. As Figure 8 illustrates, then, the combination of the network services model and the service bus is essential to the enablement of the NAP.

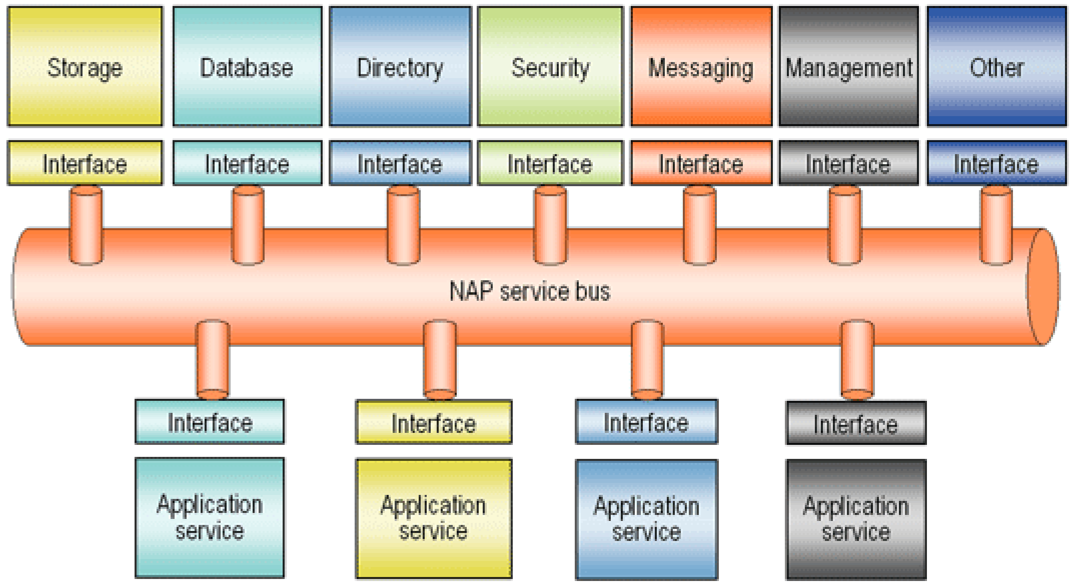


Figure 8: *Infrastructure Services Exposed on the NAP Service Bus*

Raising the Level of Abstraction

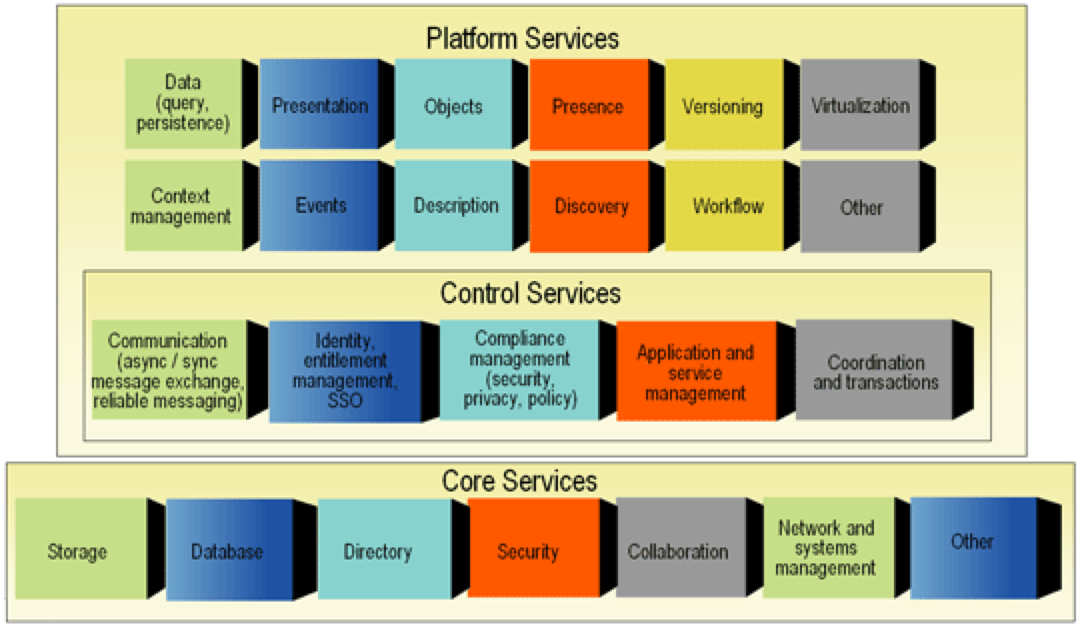
While most people agree that the core services the network services model defines are critical for enterprise-class applications, it's obvious that modern application systems require a much larger number of services to operate. It's equally clear that the core services defined in the Burton Group network services model are low-level and complex. In order to properly enable developers, the NAP must provide higher-level abstractions that allow developers who aren't systems programmers to leverage those services with high-level tools. The goals of composability, reusability, and flexibility simply aren't achievable if the complexity of programming to essential services arbitrarily constrains the flexibility of the network platform.

This evolution of the frameworks that expose infrastructure services within the NAP is directly analogous to the evolution of the frameworks that expose services within a stand-alone OS. Twenty years ago, developers had to write their own windowing frameworks, create system dialogues, and perform their own memory management functions. Over time, OSs evolved in sophistication, offering higher levels of abstraction for these and many other low-level system services. Developers can now simply call frameworks that provide high-level abstractions of these services, define input parameters, and then rely on those frameworks to interact with low-level OS functions, such as user interface and memory management systems, on their behalf.

Similarly, the NAP must provide appropriate—and higher—levels of abstraction, making it easier to quickly and easily leverage those services in a large number of applications. In short, applications rarely work directly with core networking services. Instead they work with higher-level frameworks that more closely match the business requirements of the application. For example, rather than interfacing with a directory using Lightweight Directory Access Protocol (LDAP), an application is more likely to use a provisioning service to add a new user. As an added benefit, the provisioning service ensures that the new user is provisioned properly according to corporate policies.

The Infrastructure Services Model

A fully functional NAP, then, requires an expansion of scope, beyond what we once defined as the network services model, accommodating the need for additional services and frameworks that provide higher-level abstractions of those services. As Figure 9 illustrates, the infrastructure services model (ISM) is the successor to the network services model and accommodates these realities.



Platform Services
Data (query, persistence)
Presentation
Objects
Presence
Versioning
Virtualization
Context management
Events
Description
Discovery
Workflow
Other
Control Services
Communication (async / sync message exchange, reliable messaging)
Identity, entitlement management, SSO
Compliance management (security, privacy, policy)
Application and service management
Coordination and transactions
Core Services
Storage
Database
Directory
Security
Collaboration
Network and systems management
Other

Figure 9: *The Infrastructure Services Model*

The ISM defines the NAP infrastructure in terms of the services that constitute it. Those services, such as discovery, communications, security, transactions, and reliability, supply the essential infrastructure functionality that organizations need in order to build and deploy SOA-based applications and to enable composition and recomposition of services within the NAP. The ISM relies on the WSF to enable language- and platform-neutral access to infrastructure functionality that would otherwise be trapped within a platform-specific environment.

Creating enterprise infrastructure will require a wide variety of services and applications. The ISM is not an attempt to define all possible or necessary services the infrastructure requires. Nor is it an attempt to create a canonical representation of the relationships between services. Rather, the ISM is a conceptual framework that illustrates how the next generation of network infrastructure services must evolve, and be exposed, to enable a new generation of network applications.

Standards, Interoperability, and Federation

The ISM adopts a platform-independent, service-oriented view of infrastructure functionality. The ISM defines a model that exposes infrastructure functionality as reusable services accessible through vendor-independent APIs, protocols, and frameworks. In other words, these infrastructure services can be accessed just like any other service on the service bus, as shown in Figure 9. By raising infrastructure functionality to a vendor-neutral level, the ISM enables cross-platform integration and federation. The ISM is the key enabling feature of the NAP. The ISM turns the network into a workable application platform.

Standards, then, are an obvious and important part of the evolution of the ISM. The levels of interoperability the VEN requires are possible only with solid, widely acceptable protocol standards for core service functions. Simply put, standard networking protocols are the key ingredients that enable a shared services model. Any application, regardless of its language or platform can access these services using these protocols.

Federation is an equally important component of the ISM. As Figure 9 illustrates, however, federation isn't called out as a separate service. Instead, the ISM assumes that federation is a key capability of most, if not all, of the services within the ISM. Security services such as identity management (IdM), for example, must be able to federate identity with other IdM services across the network. Likewise, transaction managers must be able to federate transactions across transaction domains, cooperating in an end-to-end process.

In that light then, federation is so important to the ISM that it's an assumption that each service within the model must be capable of federation with its peers on the network. It will, however, take years for these federation capabilities to fully evolve. And federation for all services won't evolve in lockstep. Federated identity and authentication services have already begun to evolve, for example, but in their most basic form: Web-based single sign-on for browsers. The trust fabrics and strong security mechanisms for more robust federation, including strong authentication for rich clients, will evolve over the next three to five years. Other federated models will evolve over time, with transactions being the next most likely candidate, but it will take many years for the service federation fabric to evolve.

As Figure 9 illustrates, the ISM groups infrastructure services into two categories: foundation (or core) services and platform services. The platform services also include a subcategory: control services.

Core Services

The core services form the foundation for the ISM. These low-level services provide systems-level functionality that other services, abstraction frameworks, and, ultimately, applications rely on to operate. These services include:

- Storage services, which facilitate both the storage and efficient distribution of data assets. Storage area networking technologies allow organizations to consolidate data onto centrally administered and consolidated storage devices, for example, making them available via Internet Protocol (IP) or Fibre Channel (FC) infrastructure. Organizations can partition, aggregate, virtualize, and allocate shared storage. Likewise, next-generation file systems are moving beyond physical machines toward fully distributed architectures, changing the nature of file systems and how they're used.

- Database services, which have traditionally been a core component of IT systems, continue to evolve. In fact, database systems are becoming more important as they begin to manage XML documents, objects, and other content in addition to structured relational data. In recent years, DBMSs were made less relevant by application servers. But now, in addition to providing a native data store for XML, most DBMSs support Web services.

- Directory services, which provide a simple way of naming, finding, accessing, and protecting resources. Primarily, directories provide a general-purpose identity repository, one containing organizational units and groups. In turn, these organization units and groups contain entries for people as well as their attributes including roles and other information. Other services leverage the information in the directory to perform a variety of functions including authentication, authorization, and personalization.
- Security services, which provide the basic functions of authentication, access control, data confidentiality, data integrity, nonrepudiation, content-filtering, code authentication, and intrusion detection. As such security services enforce the confidentiality and integrity of messages, authenticate users, assert entitlements, manage keys, and authorize access to system resources.
- Collaboration services, which enable communication in both synchronous and asynchronous forms. Collaboration is undergoing a revolution as voice over IP (VoIP), instant messaging, community and social software, syndication formats, and other services and tools emerge. Collaboration will continue to evolve, and collaboration services will become an essential ingredient of any application platform.
- Network and systems management services, which provide the facilities to manage the underlying network transport services (routers, switches, and other network infrastructure) and systems (OSs and applications) that constitute the network.

Control Services

As a subset of the overall set of platform services, the control services provide a higher-level set of functions and frameworks that, at least in some cases, provide higher-level abstractions of the core services. In short, the control services are responsible for routing and managing traffic, enforcing policy, and driving execution flow across services participating in a composite business process. The control services include:

- Communication services, which support addressing, routing, dynamic location, and flexible binding schemes. They support synchronous and asynchronous communications, notifications, and publish-subscribe message interactions.
- Identity and entitlement management services, which rely on underlying directory, security, and database services to provision (and de-provision) users; define, propagate, and enforce access control policies; and support single sign-on through standardized authentication mechanisms.
- Compliance management, which supports both the enforcement and audit of policy across many different functional areas including security and privacy, allowable use, and other functions.
- Application and service management services, which support the control, instrumentation, and provisioning of Web services. They also manage service lifecycle.
- Coordination and transaction services, which support context propagation, transaction management, and reliable message delivery.

Platform Services

In addition to the control services, the platform services include infrastructure-level services that support distributed application processing. (Readers should note that while the diagram in Figure 9 shows two levels of services, those levels should not be construed to imply a hierarchical relationship. Rather, the two levels conform to the arbitrary constraints on presentation represented by document formats.)

- Data services supply functions to perform data queries, definition, data modeling, metadata management, and validation.
- Presentation services are responsible for rendering information according to a specified template and aggregating data from disparate sources.
- Object services typically perform lifecycle, caching, session clustering, and reference counting.
- Presence services are used to register and discover transient service instances that can participate in a business process.
- Versioning services enable service consumers utilizing an older service interface to transparently link to a new version of the service.
- Virtualization services provide a container, virtual machine, or emulator that permits applications to operate in an environment for which they were not originally designed. Virtualization services also provide a layer of abstraction between application code and its hosting environment, thereby enabling dynamic reallocation of resources to support performance, scalability, and quality of service requirements.
- Context management services manage, control, and propagate application state.
- Event-handling services manage and route events.
- Description services provide standard frameworks to represent, manage, and distribute service metadata.
- Discovery services support mechanisms to locate services and the metadata associated with those services. For example, UDDI is a discovery service.
- Workflow services provide standard frameworks for defining and managing workflows that cross boundaries between systems and organizations.

ISM Developments

Clearly, the ISM defines a goal state—a work in progress. Standards have yet to evolve at many points in the ISM. Although the goal is still many years away, the industry is making clear progress toward the ISM nonetheless, starting with some of the most elemental components of the model. And the progress in these crucial areas demonstrates not only the value of the ISM, but how it will provide increasing levels of abstraction to support developers.

For example, great headway has been made in security, in the areas of authentication, authorization, and identity federation in particular. Low-level security functions such as strong authentication (such as PKI) and authorization have made progress in fits and starts. The complexity of these low-level services has stymied adoption. But with the emergence of higher-level, abstracted control services, adoption has become more widespread, creating stronger potential for the future.

For example, SAML defines standard, platform-independent protocols for accessing a SAML-compliant single sign-on service (SSO). Any application, regardless of its language or platform can access a SAML SSO service using these protocols. The Organization for the Advancement of Structured Information Standards (OASIS) Security Services Technical Committee (SSTC), which manages the SAML specification, is working to expand the scope of the SAML standard, including adding support for federation based on submissions from the Liberty Alliance.

Meanwhile, IBM and Microsoft continue to develop the plans laid out in the Web services security roadmap. The WS-Trust specification defines standard, protocols for requesting and issuing security tokens, and for managing trust relationships. In some ways WS-Trust (along with WS-Federation) competes with SAML, although the two specifications are also quite compatible and complementary. WS-Trust defines a framework that supports multiple authentication and authorization mechanisms and token formats. Users will be able to use WS-Trust to access a SAML SSO service. WS-Trust can also be used to access other security authorities, such as public key infrastructure (PKI) certificate authorities and Kerberos servers. WS-Federation builds on WS-Trust and defines protocols and mechanisms that will support federation of security information across trust realms.

For more information on SAML, Liberty Alliance, WS-Trust, and WS-Federation, please see the *Directory and Service Strategies* report, "Toward Federated Identity Management: The Journey Continues."

Reliability

Reliability is another example of where the industry is making progress in filling out the ISM. Traditionally, reliable messaging systems have used proprietary protocols to communicate. Different vendor products cannot interoperate, much less federate their services. The Web services reliability efforts are defining standards that will support reliable messaging over standard, interoperable protocols. Any application, regardless of language or OS, will be able to use reliable messaging services to queue messages, dequeue messages, subscribe to topics, and send notifications using standard protocols. Specifications of interest in this area include:

WS-Reliability: an OASIS effort that supports reliable messaging

WS-ReliableMessaging: a specification from BEA, IBM, Microsoft, and TIBCO that supports reliable messaging

WS-Eventing: a specification from BEA, Microsoft, and TIBCO that supports a lightweight event-management mechanism

WS-Notification: an OASIS effort that supports a publish-and-subscribe mechanism

Web Services Resource Framework: an OASIS effort that supports stateful resource management

Transactions

Likewise, there's discernible progress in the area of transactions, albeit somewhat slower. Traditional transaction management implements the X/Open Distributed Transaction Processing (DTP) model, a set of transaction-processing standards maintained by The Open Group. This model includes the XA Specification, which defines protocols to support tightly coupled, synchronous, distributed transactions using a two-phase commit (2PC) mechanism. These standards are used by most database, message-queuing, and other resource management systems. These standards, which predate the launch of the World Wide Web, are rather complex and define language bindings for the C programming language. Applications rarely interface directly with DTP. Instead applications often rely on the database to manage transactions. In other circumstances, applications might rely on an application server framework, such as an MTS or EJB container, to manage transactions.

And while the DTP model will be with us for many years, it doesn't fully address the requirements for transaction services in a loosely coupled, service-oriented environment in which business transactions may take minutes or days to complete. The NAP requires a versatile transaction-coordination framework, which supports more options and flexibility than DTP has to offer. It requires a loosely coupled transaction-coordination framework that supports asynchronous communications and long-term transactions. Rather than relying on 2PC mechanisms, loosely coupled transactions must rely on reliable messaging, state synchronization, compensating transactions, and other loosely coupled, coordinating frameworks. The NAP must also provide language-independent APIs for accessing these frameworks via standard protocols.

Specifications of interest in this area include:

- WS-Composite Application Framework: an OASIS effort
- WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity: specifications by BEA, IBM, and Microsoft

Policy Will Become the New Programming Model

As the previous section of this overview illustrates, progress toward the ISM is being made in the areas of security, reliability, and transactions. The developments in these areas also demonstrate how the ISM will change the way developers build applications.

The ISM exposes infrastructures service on the service bus via standard protocols. Consequently, any application, regardless of its language or OS, can access and utilize the service. As discussed in this document's “The WSF: Policy-Driven” section, this basic architecture will permit developers to use declarative policies to define the semantics of how the application should use the service. Effectively, policy will become the new programming model. These concepts will facilitate the dynamic composition and recomposition of services (including infrastructure services) required by the VEN.

NAP Service Design Practices

The NAP will not emerge instantaneously in its full glory in an organization. It is not a product you can buy. It is an architecture that enterprises must design and implement with deliberate intent over time. The key, then, is to begin using implementation techniques that are available today, to start down the path toward the NAP. Only by taking small steps toward the goal will enterprises ever reach it.

The place to start, then, is to focus on flexibility and reusability in application design. At the big-picture level, development teams should start using SOA design principles and best practices on all new application development projects. The first step in this process is, of course, training developers on SOA and best practices for developing reusable services. For more information on how to factor monolithic applications into discrete application services through the use of service design practices, see the Application Platform Strategies Reference Architecture Technical Position, “Application Factoring.”

The service design practices (SDP) define a set of basic design principles and best practices for loosely coupled, reusable services. Organizations should follow the SDP as a foundation for defining precise corporate design and development policies.

In summary, the SDP maximize service sharing, reuse, and interoperability. The primary goals of the SDP are to ensure:

- Clean separation of service interface from implementation
- Clean separation of business logic from infrastructure functionality

Separation of Interface from Implementation

The first goal, which is often referred to as loose coupling, is an essential aspect of flexibility and interoperability. Service consumers interact with a service through its interface. The service interface should hide the implementation details of the service, such as programming language, OS platform, and internal object model. The degree of abstraction between the service interface and the implementation directly affects how brittle (or flexible) application connections will be. The higher the abstraction, the more flexible the connection is. When there is clean separation of interface from implementation, a developer can modify the implementation without impacting the consumers that use the service.

To increase the abstraction layer between interface and implementation, developers should follow three basic rules:

- Services, not components
- Chunky, not chatty
- Schemas, not classes

Services, Not Components

The first rule addresses service granularity. A service should implement a relatively autonomous, coarse-grained function. It should not be designed as a fine-grained component with interdependencies on other components. Coarse granularity reduces the complexity of the application topology and simplifies application maintenance. As a rule of thumb, a service should implement a discrete task within a business process. Note that developers may use fine-grained components and tightly coupled connections within a service.

Chunky, Not Chatty

The second rule addresses frequency of interaction. A developer should design a service so as to reduce the number of network interactions required to complete a discrete task. Figure 10 demonstrates the difference between “chatty” and “chunky” interfaces for an order-entry service. The chatty application uses an object-oriented design, in which the client interacts with the service as if it were a remote object. It requires separate interactions to create an order, to add each item to the order, and to submit the order. The chunky application uses a service-oriented design, in which the client application builds the order locally and requires only one interaction to submit the order. Chunky interface design reduces the likelihood that external programmatic interfaces will need to change as internal service implementations change.

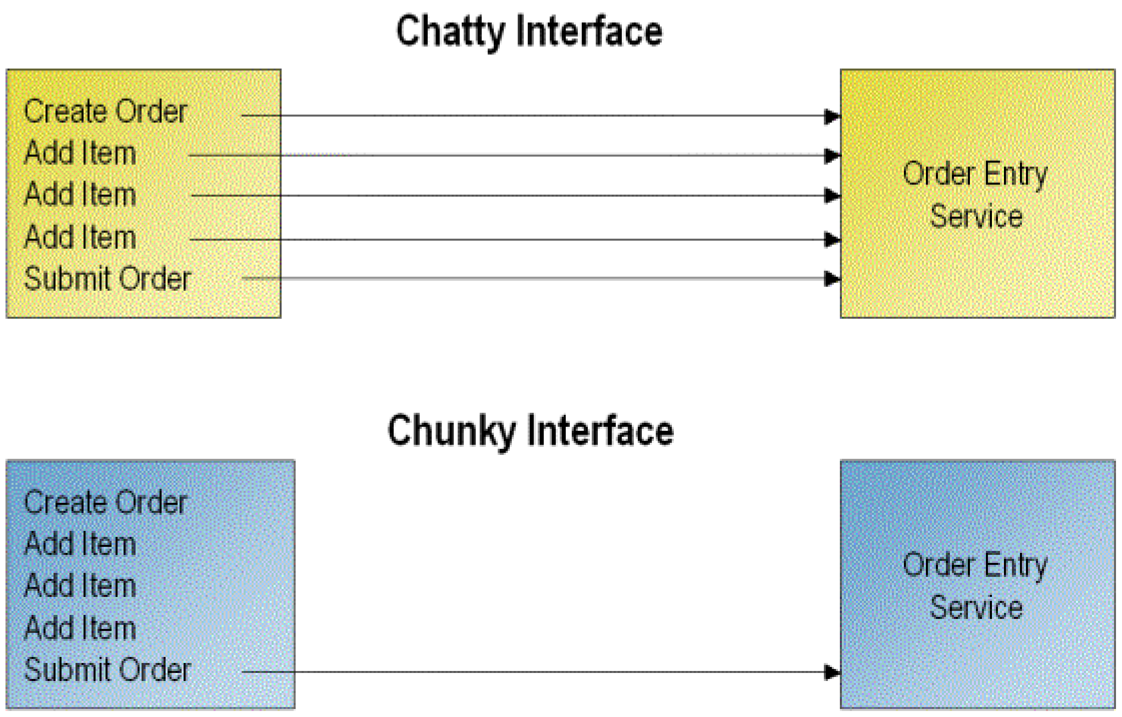


Figure 10: *Chatty vs. Chunky Interfaces*

Schemas, Not Classes

The third rule addresses message formats. A service should exchange well-structured messages rather than objects or classes. Exposure of the internal object model reduces the interoperability characteristics of the service. In particular, a service should not expose language-specific data types, such as Java Hashmaps or .NET Datatsets through its interface. The service's input and output messages should be defined using a language-independent data definition schema such as XML Schema. The message format should consist only of simple types, structures of simple types, and arrays of structures and simple types.

See the *Application Platform Strategies* overview, "Service-Oriented Architectures: Developing the Enterprise Roadmap," for further discussion of loose-coupling best practices.

Separation of Business Logic from Infrastructure Functionality

For maximized reuse and flexibility, an application's infrastructure functionality must be defined separately from the business logic. When possible, infrastructure functionality should be specified using declarative policies and be automatically implemented by the runtime environment.

As a general rule, developers should develop services using a managed code environment, such as Java and .NET. The managed code runtime system (also called a virtual machine) implements quite a bit of infrastructure functionality, including storage allocation, object lifecycle, session management, thread management, resource pooling, and code security, on behalf of the applications.

Developers should also be sure to use intermediaries and interceptors to implement advanced operational semantics, such as security, reliability, and transactions.

The NAP: Trends, Inflection Points, and Discontinuities

Because the NAP, by definition, is a multivendor environment, no vendor can ever deliver a single product that implements the NAP in its entirety. The NAP will emerge through the cumulative effects of standardization and commoditization. As vendors include support for emerging WSF standards in their products, the NAP will be a natural byproduct of those efforts. Enterprises will need to design their own NAP, based on available products and solutions. And yes, products will emerge for managing the connections between systems, and easing integration via the NAP. But to succeed, the NAP must be as pervasive as basic Internet transport protocols (such as TCP/IP) are today. The components of the NAP must simply be part of a pervasive infrastructure.

Still, the emergence of the NAP, and the new levels of connectivity it portends, will have a significant impact on the platform, integration, and middleware markets as we know them. Standardization and commoditization will have a huge impact on software vendor business models, for example. And the NAP will introduce discontinuities—points at which competitive dynamics shift as the market shifts, undermining entrenched leaders that are unprepared for that shift and empowering both existing and new companies that are driving the change. Here, we outline the trends that are leading to the NAP's evolution in key market segments, and the impact some of those trends will have.

Platform Vendor Trends

During the past few years, application platforms have grown from being a basic application development environment to a comprehensive, integrated solution. One of the platform vendors' core goals is to provide a complete solution, giving them more account control. A modern application platform, such as IBM WebSphere, Microsoft .NET, or Oracle 10g, is an integrated product suite that includes tools, portal, presentation server, application server, integration broker, resource connectors, database server, directory, security framework, and management facilities.

The platform aggregation trend will continue unabated. The boundaries among OSs and specialized servers for applications, database management, enterprise messaging, portal/presentation, collaboration, and other needs are rapidly blurring. As more services sink into platforms (Java 2 Enterprise Edition [J2EE] environments, virtual machines, OSs, and other environments), and as SOA streamlines and simplifies system integration, the traditional product categories for DBMSs, application servers, integration brokers, TP monitors, and MOM, among others, are converging and consolidating.

Middleware, as we know it, will be subsumed by the base platform. IdM, digital rights management and channel-centric communication tools (instant messaging, Really Simple Syndication [RSS], Short Message Service [SMS], etc.), will be absorbed into the core platforms over the next three to five years. MOM, advanced file management, and collaborative workspaces will also follow this same pattern. And this is the natural order of things. As commoditization puts price pressure on the basic OSs, frameworks, and toolsets that defined the application platforms of the past, vendors will naturally look to maintain a high value proposition by including new functionality. Given Microsoft's propensity to put more and more functionality into the Windows Server box, for example, one could reasonably argue that OS kernels became zero-cost commodities a long time ago.

Burton Group's research agenda over the next few months includes coverage of many of these application platform aspects. For more information on the future of portals, see the *Application Platform Strategies* overview, "New Portal Standards and Future Developments." And for more information on the morphing integration broker market, see the *Application Platform Strategies* report, "Integration Brokers: Providing the Middleware Fabric for Complex Integration."

Subsidizing Infrastructure Deployment

Today, the APIs and frameworks within an application platform provide access to the platform-specific infrastructure services embedded within the platform. In time, the platform vendors will enable transparent integration and federation of the platform-specific infrastructure services with ISM services on the service bus. This integration will virtualize the infrastructure and permit applications to invoke ISM services using platform-native APIs.

As platforms subsume core infrastructure and middleware functions, these functions will become more widely used because they will be more widely available. Again, this is the natural order of things. Platform vendors, in essence, subsidize infrastructure development and deployment by including infrastructure functionality “for free” in their platform products. We put “free” in quotes because that infrastructure isn't free, of course. But the benefit to the market at large is clear.

Mainstream commercial developers generally don't use infrastructure services until they become available in a large segment of the market. Only when a platform with respectable market share implements a given infrastructure service in a consistent fashion can commercial developers assume that the infrastructure service is present in a potential customer's environment. Customers, however, don't want to buy and deploy the infrastructure until applications that warrant it arrive. Using platform products to push infrastructure out into the market is a proven method of breaking this Catch-22.

Customers can expect Microsoft and the J2EE vendors—especially BEA, IBM, and Oracle—to continue their efforts to provide tidy, integrated systems that profess to provide a complete application platform for enterprise applications. For many customers, integrated solutions have strong appeal—they certainly make application development easier—but enterprises must be aware that no single vendor can truly supply a complete application platform for enterprise applications. Besides, the vendor story line regarding “complete” solutions tends to lose credibility when viewed in light of reality. Most enterprises will use .NET and J2EE platforms, as well as LAMP, the open source scripting platform based on Linux, Apache, MySQL, and PHP/PERL/Python.

To address the need for integration, then, platform vendors will continue to implement support for the WSF in their products, leading to higher consistency and interoperability among the platforms. Support for the WSF technologies defined in the WS-I Basic Profile is now so pervasive that developers can assume that products are readily available if not already deployed on systems. The vendors will also implement support for the emerging Web services standards initiatives for secure, reliable, transacted messaging (known as WS-*) as they mature.

Reordering of the Server Stack

But support for SOA and the WSF will create an interesting counterbalance in the platform market. In short, the NAP promises to create a plug-and-play environment comprised of loosely coupled, specialized services and frameworks. The NAP permits decoupling of application and infrastructure functionality and supports best-of-breed technologies. As developers begin the process of factoring out application functionality into loosely coupled reusable services, applications will become less dependent on the capabilities of a specific application platform. Applications will be able to consume application and infrastructure services without being concerned about implementation details. While leading platform vendors will continue to develop integrated solutions, then, SOA and the NAP will effect a reordering of the traditional server stack.

Application architectures of the past require vertical integration with one platform at every layer of application architecture. But because SOA and NAP decouple these functions (by necessity, to enable large-scale integration) developers will be freed from the constraint of dependence on a single platform at every layer in the architecture. A single application might use a Flash client that has been distributed to a desktop via a portal running on LAMP. This portal accesses a remote portlet running in a presentation server running on Windows, which renders the client interface dynamically. The portlet aggregates information from various application and data access services scattered on Windows, Linux, Unix, and mainframe systems. The portlet relies on an integration broker running on Unix to coordinate the service invocation and data aggregation process.

In other words, the choices of a fully integrated solution versus best-of-breed will always be available. Customers will be able to choose between these approaches based on their needs. Some may prefer the unified support and service, in addition to a single source for problem resolution that fully integrated solutions enable. Others may prefer the freedom from vendor dependencies that the best-of-breed approach enables. Both have tradeoffs, and a thorough examination of those tradeoffs is beyond the scope of this document. Burton Group will be covering that topic in other *Application Platform Strategies* documents.

Open Source and Software Commoditization

Open source software (OSS) is perhaps the most disruptive trend in software development to emerge since the PC itself. Any platform discussion today must include LAMP and other open source frameworks. Given the increasing popularity of its components, and the open source value proposition, LAMP is clearly a credible platform alternative.

Given that it's a “build your own” application platform, LAMP lies somewhere in between a fully integrated and a best-of-breed offering. On one hand, it's a related set of technologies designed to enable a fully functional Web development environment. On the other hand, however, LAMP isn't quite as tidy and integrated as .NET or J2EE, and it doesn't profess to provide a complete application platform.

LAMP is a collection of loosely coupled frameworks and technologies, and it's up to the developers to make sure that the pieces work together properly. Many people are initially attracted to open source software in general, and LAMP in particular, because it's free. But open source isn't as free as many may initially perceive it to be. Software typically comprises less than five percent of the total cost of ownership of most projects. And regardless of the type of license used, software requires training, service, support, upgrades, and updates. In fact, open source software may often require more care and feeding than commercial products.

The real advantages of open source lay in many of the freedoms it allows. Open source projects are not bound by vendor initiatives and agendas, such as the endless upgrade cycle that drives vendor revenue. Open source software doesn't come with usage restrictions such as a maximum number of CPUs, network connections, or data size, nor do open source licenses restrict usage to just internal or external users. Open source software also gives enterprises the freedom to inspect and verify the code and to modify, customize, or repair it. In the end, the real advantage is that open source provides these freedoms while enabling a “good enough” solution to many specific needs.

It's also clear that OSS is both a cause and an effect of the NAP's emergence. Linux's emergence can easily be seen as the final step in the complete commoditization of the OS, for example. As the focus shifts to application frameworks that tie multiple OSs and applications into a larger, more meaningful whole, the value of the OS itself naturally diminishes. The value shifts to platform services that both reside above the OS, and the frameworks that bind many OSs into a more meaningful whole.

At the same time, open source efforts such as Ximian (now owned by Novell), Apache Axis, and JBoss are working to provide open source implementations of key components of the NAP, over and above the basic foundation that LAMP provides. It's fair to assume that, over time, open source initiatives will provide credible alternatives to commercial implementations of key functions of the NAP. And by doing so, open source software will continue to cause substantial changes in the software business itself. Like any substantial issue, open source has both advantages and disadvantages, which are discussed in detail in the *Application Platform Strategies* overviews, “Open Source Software: Risks and Rewards” and “Linux: "Good Enough" for Specific Enterprise Roles.”

The Move to Managed Code Architectures

One of the most significant platform trends is an increased focus on comprehensive security and manageability frameworks, also known as “managed code.” While perhaps not the most exhilarating of trends for application developers, this trend is a top business priority and impacts every facet of the application platform stack. The need for strong platform security has been exacerbated by the blurring of the boundary between enterprise systems and the Internet. Security and management used to be left to manual processes and developer discretion; now they are increasingly framework-based, policy-driven, and automated (mandatory and automatically enforced).

The drive for improved application security and manageability, along with the ongoing need to improve developer productivity and application flexibility, has produced a major shift toward virtualization. One form of virtualization is the transition to a new application platform runtime based on virtual machine architectures and expansive, integrated class frameworks. The leading examples are the Java platform and Microsoft .NET.

The need to support existing applications without compromising platform-level security and management frameworks is also driving another form of virtualization, in which older OSs are run in a virtual sandbox atop current-generation platforms. While this type of virtualization is not yet pervasive, it will expand rapidly, and in doing so will significantly alter assumptions and decision criteria about what it means for a vendor to be usefully “platform-independent.”

It's also interesting to note that the rise of managed code and application virtualization are additional drivers of the commoditization of the underlying OS. Managed code applications rarely interact directly with the OS, and when using sandbox virtualization, even older applications don't need to be aware of the underlying OS. However, managed code does not necessarily alleviate all platform dependencies. Customer requirements for and vendor commitments to industry standards are driving rapid platform evolution, and have produced dramatic improvements in the scope and simplicity of cross-platform interoperability. There has been consolidation in the platform, tool, and application markets for more tightly coupled applications, and some standardization progress; however, the market reality today is that tightly coupled application optimization entails commitment to a specific vendor platform, and this holds true for .NET, Java, and LAMP.

The Database Is Dead, Long Live the Database

Somewhat surprisingly, given the maturity of the technology, database systems are moving back into the limelight. DBMSs have moved beyond relational—they manage XML documents, objects, and other content in addition to structured relational data. In recent years, DBMSs were made less relevant by application servers. In tiered application architectures, DBMSs always get buried in the back end, hidden behind a resource connector in the middle tier.

But now, in addition to providing a native data store for XML, most DBMSs provide support for inbound and outbound Web services. This evolution makes it difficult to distinguish a database system from an application server or an integration broker. In an SOA, the database doesn't have to be hidden behind a resource connector. A database can expose a service, such as a stored procedure or an XQuery function, directly on the service bus. A database can also act as a coordinator of an orchestrated business process. Developers program these functions using standard development languages, such as Java or C#, providing a unified programming model for triggers, stored procedures, constraints, events, and processes.

For more information on DBMSs and their role in distributed systems, see the *Application Strategies* overview, “Data, Documents, and Database Management Systems in Context: Toward a Renaissance of Data Management.”

The Rise of Rich Clients

Client-side platform evolution has been relentless during recent years, with browser-based applications now the norm for both Internet and intranet applications. Applications and tools designed to exploit the full power of smart client devices still exist, of course; most enterprise employees today divide their time between browser-based applications and “richer” applications such as Microsoft Office or IBM Lotus Notes.

Now that the Web application wave is pervasive, a growing emphasis on effective user experience and optimized leverage of smart client device capabilities is driving changes in client platforms. In many application contexts, the conventional Web-user experience enabled by HTML and JavaScript is adequate, but far from ideal. The near future will see a rise in rich Web-based user interfaces dynamically rendered by presentation servers using technologies such as Flash or Java. These presentation servers typically support a declarative programming model that supports rapid development and customization. Examples include Macromedia Flex and Microsoft's forthcoming Longhorn presentation subsystem, Avalon, as well as products from startup players Droplets, Laszlo, and Nexaweb.

Changes in information management and presentation patterns are also leading to a shift from browser-based to smart client-optimized applications and tools. Examples of this shift include the rapid growth of RSS aggregator clients (displacing traditional Web site browsing), readily personalized portal clients, and multipurpose collaborative workspaces (e.g., WebEx, Microsoft Office Live Meeting, and Groove).

Tool Trends

The leading integrated development environments (IDEs), including products from Borland, IBM, Microsoft, and Oracle, as well as open source initiatives such as Eclipse.org and NetBeans.org, are broad and deep, tightly integrated tool suites that can be used for applications ranging from simple database-driven forms to intricate system programming needs. And these tools now support automatic generation of Web services, which will increase both the consistency and speed with which business developers can leverage the NAP and its underlying infrastructure.

Specialized tools are still important in some situations, and developer preferences will always be a key consideration. (In other words, it's okay to use emacs or vi if you prefer, and these editors will continue to work well with leading-edge options such as Macromedia Flex and Microsoft Longhorn's Avalon). But today's leading tools focus on integrated tool environments, and they're rapidly expanding to include tools for model-driven development and policy-guided implementation.

The combination of these “complete solution” tool environments with the increasingly rich open source initiatives poses stiff competition for new startups in the tools market. Probable survivors include established vendors of leading-edge presentation servers, such as Adobe, Borland, and Macromedia. Other probable survivors include vendors of leading-edge XML composition tools, such as Altova and Sonic Software.

New tools will emerge that capitalize on the mix-and-match capabilities of SOA and the NAP. These composite application development tools will focus on helping developers refactor, compose, and recompose services to create new applications. The current crop of orchestration tools today still relies on proprietary workflow engines and orchestration languages, although standards in this area are maturing. Vendors will need to adopt WSF and orchestration standards to survive. Please see the *Application Platform Strategies* overview, “Orchestrating Web Services: Driving Distributed Process Execution Through Workflow Technology,“ for more information on this emerging trend.

Application Trends

Platforms and tools are evolving more rapidly than applications (quite reasonably, since applications are created with tools and run on platforms), but application domains are also undergoing major transformations.

In enterprise application categories such as enterprise resource planning (ERP), customer relationship management (CRM), and sales force automation (SFA), the shift to an SOA-enabled application composition framework is driving demand for enterprise applications that can be leveraged as collections of components and services. “Silo” applications that can't be readily leveraged in other application contexts are far less useful for application developers working with today's IDEs and targeting new client and server platforms. Companies such as E.piphany, Oracle, PeopleSoft, SAP, and Siebel are slowly but surely adding support for Web services to their application suites (although no doubt with some trepidation as they relinquish their lock-in strategies).

Meanwhile, in other areas, SOA and Web services seem to be going mainstream. Microsoft, certainly, is augmenting its Office and small business application software suites with Web services support. The rapid growth of higher-level service-oriented applications is also encouraging as application service providers such as Salesforce.com challenge the conventional definition of enterprise applications, and collaboration-oriented vendors such as Microsoft, RainDance, and WebEx make their wares available as services.

Infrastructure Trends

It will take a bit longer before SOA truly permeates platform infrastructure and vendors deliver infrastructure functionality as services available on the service bus. Today, many infrastructure services are firmly embedded within the application platform frameworks. But even here, the industry is making rapid progress, in part because of the standards-focused collaboration among leading vendors working on and implementing the extended Web services standards stack known as WS-*.

The tools and services commonly included in platforms, ranging from foundation-level services such as IdM and authentication to more user-oriented collaboration and communication capabilities, enable application developers to focus more on business processes and less on system-level infrastructure. While progress has been uneven in defining standards for identity, authentication, transactions, and other core services in the ISM, there's ample room for cautious optimism. It's still not easy or common for application developers to exploit infrastructure services that buffer them from the idiosyncrasies and intricacies of cross-domain computing, but the services, along with the platforms, tools, and applications required to fully exploit them, are all rapidly evolving.

Risks and Opportunities for ISVs

The reordering of the server stack discussed earlier in this section will create opportunities for the independent software vendor (ISV) community—at least for vendors that embrace standard protocols. Conversely, vendors that rely on proprietary protocols and vendor lock-in strategies, especially enterprise application integration (EAI), enterprise information integration (EII), and application development tools vendors will find themselves squeezed out of the market by the platform vendors and the SOA infrastructure players.

The future ISV market will be led by vendors that implement, integrate, and innovate with leading-edge SOA-centric products. In particular, opportunities abound for vendors that can supply SOA-based infrastructure services that support cross-platform federation and policy administration. Web services infrastructure players, such as Systinet and webMethods, who consistently deliver first-to-market implementations of the WS-* specifications, provide leading-edge support for security, reliability, and transactions. Security vendors, such as Entrust, Netegrity, Oblix, and Verisign, provide security, identity, and provisioning services based on models that conform to the ISM. Web services management vendors, such as Actional and AmberPoint, provide crucial SOA and runtime policy management facilities. And WebLayers provides SOA design and development policy management facilities.

Recommendations

No single vendor can deliver the NAP as a complete product because it is, by definition, a multivendor environment. The NAP comprises a wide variety of elements in the computing infrastructure, including application platforms, portals, application servers, integration brokers, database systems, transaction management systems, registries, IdM systems, security services, and management frameworks. The point of integration for all these elements is an open, extensible service bus based on the WSF.

As stated in “The Cusp of Change”section of this overview, enterprises today face a strategic imperative to get on the path toward the NAP. That means enterprises need a clear plan for getting there. That plan must allow enough flexibility for adapting to emerging new standards and changing business dynamics. But it must also accommodate enough specificity to ensure that the organization arrives at the right destination.

In other words, enterprises shouldn't let the relative immaturity of the overall architecture stop them from undertaking their own NAP initiatives. The business imperatives around the NAP are clear, and the organizations that are able to manage the migration to the NAP and SOA effectively will have a distinct business advantage.

The most important deployment consideration is a simple one: Remember the business problem you're trying to solve. Today, application architectures must relate to business objectives. Consequently, enterprise application architects must relate SOA and the rise of the NAP to core business objectives that upper management will support. It's also clear that one platform, no matter how well-integrated it is, won't solve all application development problems. As we said earlier, the NAP will emerge as a pervasive and evolving infrastructure, one that transcends corporate boundaries and vendor product lines.

In that light, it's important to understand that the NAP is a goal state. In working toward that goal state, enterprises should identify starting points, understanding that it will take years to achieve the NAP. For now IT departments will probably have to do some custom integration work to make products work together seamlessly. In the long run, though, platforms, tools, applications, and infrastructure products will eventually supply inherent support for the NAP. At that point enterprises will be able to buy new products and simply snap them into place in the NAP.

Here, we outline our basic recommendations for getting on the path toward, and planning support for, the NAP.

SOA Buy-In

As stated earlier, SOA is more about mindset than it is about technology. An SOA roadmap begins with organizational groundwork. SOA initiatives will not be successful without management buy-in. SOA requires a commitment from multiple organizations to work together to promote sharing and reduce redundancy. If management does not yet believe in the value of SOA, the first step must concentrate on education and evangelism. Put together a business plan and lobby it to senior management and the various business groups. Make an effort to get buy-in from developers, architects, and operations staff, too.

Build a NAP Service Bus

A number of vendors, including IBM and Sonic Software, have been promoting products classified as ESBs. Most of these products are based on a JMS infrastructure. While they address many integration requirements, however, these products fall short of a true NAP service bus.

A JMS infrastructure introduces specific product dependencies, gives undue preference to the Java platform, and constrains intercorporate communications. Today, JMS requires the installation of a specific JMS product in various locations throughout an organization and in other organizations that want to communicate using the bus. Although a JMS-based ESB can be used as part of the service bus, it should not be used as the core foundation for the environment. A service bus must be completely vendor- and platform-neutral, and it must support unfettered intercorporate communications. That means using the Web services framework as the foundation for the service bus, and considering the bus in the context of the Web services infrastructure. A Web services infrastructure consists of WSPs, WSM, and other ISM infrastructure services.

Please see the Reference Architecture Template, "Web Services Infrastructure" for an overview of these technologies. Also, please see the *Application Platform Strategies* overview, "Selecting a Java Web Services Platform: An Evaluation Framework," for a thorough analysis of the required features and capabilities of a WSP, and the *Application Platform Strategies* report, "Web Services Management: Gaining Control of Distributed Services," for a review of WSM products.

Encapsulate then Migrate

Developers should capitalize on opportunities to encapsulate legacy applications. Whenever an application project requires integration with a legacy application, developers should replace existing point-to-point connections with a service-oriented facade, thereby enabling an abstraction layer for access to legacy assets through the service bus. Decoupling systems in this fashion will make it easier, over time, to change either side of the connection without causing substantial disruptions to connected systems.

Application encapsulation goes only so far, however. Over time, enterprises will experience increasing pressure to migrate, retire, or refactor these legacy applications. Eventually, enterprises will need to migrate the legacy systems to true SOA compliance to reap the full benefits of the NAP. The opportunity costs associated with staying with a monolithic architecture will escalate, especially when changes are required to comply with legal regulations. The encapsulation facade will ease the migration process, however.

Factor Application Functionality: Create Services

The most profound change in design practices is associated with application factoring. Developers must get out of the habit of building monolithic applications. Instead, application functionality should be factored into multiple services, where each service implements a discrete business task. Each service may be used in any number of applications. An application is a choreographed process flow that invokes the appropriate set of services to accomplish a specific business process.

Refactoring and abstraction are constant themes in SOA. Developers should consider refactoring a service that implements multiple tasks. Developers should be careful not to go overboard, though. A service is typically larger than an object or a component.

When following good service design practices, the interface should hide the details of the service implementation. A service should expose its capabilities as a set of abstract business functions. A calling application shouldn't need to know whether the service retrieves data from a database, aggregates information from a variety of data sources, or generates the information using internal algorithms.

Certainly there are times when it's inappropriate to break an application into multiple, loosely coupled services. Some applications require a tightly coupled, tiered architecture in order to support performance or transaction requirements. Developers must weigh the benefits of flexibility and reusability versus speed and expediency. Note also that the code that implements the service typically uses tightly coupled connections among its internal components.

SOA Governance

Organizations should be aware that most current application development tools do not encourage developers to follow the NAP SDP guidelines. In many tools, the path of least resistance leads developers to build fine-grained, tightly coupled, object-oriented applications with integrated infrastructure functionality. In order to encourage good design, organizations must define a set of design and development policies and establish processes to manage the governance of these policies. Developers should be required to provide justification for any violation of the defined policies.

Because the market for Web services and SOA development tools is immature and in its formative stages, there are a variety of small vendors that provide tools that can help enterprises with SOA governance. As is always the case, enterprises should approach such products and vendors with the understanding that the larger platform vendors are likely to make such products a part of their platform and tool product lines in the future. But many offer leading indicators of the kind of functionality that enterprises need at a minimum, and some may be useful in both the short and long term.

For example, developers should test their services for conformance with the WS-I BP. Mindreef provides a Web services diagnostic system called SOAPscope that includes a WS-I BP conformance tester. SOAPscope also supports debugging, testing, and tuning of Web services.

A diagnostic system doesn't necessarily help developers design flexible, reusable interfaces, though. Developers need training and guidance to help them learn the art of interoperable XML Schema design. Swingtide provides services and an interoperability lab that some companies might find useful.

Training is essential, but it only goes so far. Organizations that are really committed to SOA must define and govern design and development policies. These policies can vary in scope from general directives (such as all Web services must conform to the WS-I BP) to specific coding principles (such as conventions for defining XML Schema elements or namespaces).

For example, WebLayers provides a policy server for this purpose. WebLayers easily integrates with an enterprise's normal development process. It provides facilities to define policies for design and development, to test code for conformance with these policies, and to implement governance and justification processes.

Implement Incentive Programs

Organizations should also devise incentive programs to encourage developers to embrace SOA and reusable service design practices. For example, developer success factors should include accomplishments such as:

- Reducing new lines of code by reusing pre-existing services
- Decommissioning existing applications in favor of a reusable service
- Authoring a service that is used by other projects

Model-Driven Development

One way to support good design practices is to use model-driven development methodologies. Many application development tools, such as Borland Enterprise Studio for Java, IBM WebSphere Application Developer, Microsoft Visual Studio, and Oracle JDeveloper, now support integrated modeling tools.

Model-driven development systems allow developers to work at a higher level of abstraction. Developers use a modeling tool to portray the application data structures, functionality, and process flow. When the model is complete, the tool generates code that implements the model. Most tools provide mechanisms to customize the generation process to ensure that the generated code conforms to specified architectural policies.

Dealing with Loosely Coupled Connections

Transactions become much more complicated in a loosely coupled environment. The coordinating application typically doesn't have the ability to hold and control locks in the resource tier. If errors occur during processing, the database can't automatically back out the changes. Instead the coordinating application needs to invoke a compensating transaction that reverses the effects of the failed transaction. For the moment this scenario shifts the burden of transaction management from the database to the shoulders of the developer. The Web services standardization community, though, is working to define automated transaction management technology that works in a loosely coupled environment.

One point to remember is that, in the NAP, infrastructure functionality is also exposed as services in the ISM. An application can call these infrastructure services using the same technologies and protocols that it uses to call application services. For example, the WS-Coordination specification, which describes a loosely coupled transaction-coordination framework, defines three types of infrastructure services:

- Activation service, which creates a transaction activity
- Registration service, which allows an application or service to register to participate in a transaction activity
- Coordination service, which manages the completion or recovery of a transaction activity

Over the long term, the emerging tools that support composite application development will automate the use of these infrastructure services using declarative policies and runtime frameworks. For example, Microsoft's Indigo, which is a part of the company's "Longhorn wave," promises to support the automated development of Web services on Microsoft platforms. But it will take years for these tools to evolve to the level of functionality that enterprises need.

Standards Conformance for Interoperability

Good service design is paramount, but so is standards conformance. The service bus works only if the services communicate using standard formats and protocols. For maximum interoperability, developers should build services in conformance with the profiles defined by the Web Services Interoperability Organization (WS-I). The WS-I Basic Profile (WS-I BP) provides guidelines for building interoperable Web services based on SOAP 1.1, WSDL 1.1, and UDDI 2.0. The WS-I BP defines a set of constraints on these specifications that reduces the number of variables and ambiguities inherent in the WSF specifications. The most significant constraint in the WS-I BP is that it prohibits the use of SOAP encoding. As a result, developers should always build document/literal style services.

WS-I is also producing profiles to support message attachments and basic security. Developers should abide by these profiles once they have been ratified.

Facilitate Service Sharing

Organizations should implement a development infrastructure that facilitates service sharing. In particular, organizations should use the WSF standard for service advertising and discovery, the UDDI service. Most Web services development tools and services support integration with UDDI.

Conclusion

Enterprise IT departments are being asked to deliver increasingly high levels of integration, both internally and externally, on shrinking budgets. These factors are driving both the commoditization and standardization of integration software and the emergence of the network application platform (NAP). Based on service-oriented architecture (SOA) and the Web services framework (WSF), the NAP defines a vendor-independent infrastructure that is turning the network into a practical platform for a new generation of applications. Given today's business environment, enterprises must factor the emergence of the NAP into their long-term plans, and change both their mindset and architectures for application development.