Choosing the Right Presentation Layer Architecture
David Hill, Microsoft Corporation
pp 04 – 13

Information Bridge Framework
Ricard Roma i Dalfó, Microsoft Corporation
pp 14 – 18

Benchmarking a Transaction Engine Design
Richard Drayton, FiS and Arvindra Sehmi, Microsoft EMEA
pp 19 – 33

Enterprise Architecture Alignment Heuristics
Pedro Sousa, Carla Marques Pereira and José Alves Marques, Link Consulting, SA
pp 34 – 39

Razorbills
Maarten Mullender, Microsoft Corporation
pp 40 – 49

Next Generation Tools for OOD
Oren Novotny, Goldman, Sachs & Co
pp 50 – 52

JOURNAL4

JOURNAL4 MICROSOFT ARCHITECTS JOURNAL OCTOBER 2004 A NEW PUBLICATION FOR SOFTWARE ARCHITECTS

Dear Architect

The advances in technology and industry standards that enable service oriented architecture (SOA) are also driving a paradigm shift in client architecture. Individually these changes may be considered evolutionary, but in aggregate they unite to revolutionize client architecture. Historically, client-side applications were tightly bound to a specific backend system. More precisely, the primary function of the client-side application was to expose the functionality and data of a specific backend system to users. While it was possible to build clients that could interact with multiple backend systems, doing so was extremely complex and seldom undertaken.

SOA has changed all of that. In a SOA, the functionality and data of backend systems are exposed as services (or fronted by mid-tier web services), and those services are consumed by client applications. The client application is no longer tightly bound to a single backend system. Businesses now have the freedom, and means, to efficiently build radically new client applications that truly optimize user productivity and streamline workflow.

The design imperative for client side applications has shifted from system-centric models of connecting the user population at large to specific systems, to user-centric models that provide users with transparent access to all the services they need for their job or task. If one extends this model by seamlessly integrating these services with the rich capabilities of desktop applications, then one can achieve something which redefines the standards for operational excellence. And furthermore, by giving equal treatment to both user and system models within a convergent service-centric model, one can significantly enhance the integration between them with seamless functional interoperation.

In this edition of JOURNAL, we begin the intriguing journey to explore perspectives of these various models.

Enjoy!

Chris Capossela
Vice President,
Information Worker Business Group,
Microsoft Corporation

Editorial

By Arvindra Sehmi

Dear Architect
Welcome to the autumn issue of JOURNAL.

It has been several years now since .NET hit the streets and created momentum around Microsoft's vision of connecting people and processes together, anytime, anywhere and on any device. This vision was built on Web services standards implementations on .NET and broad adoption in the technical community. Both prerequisites have been achieved, not only on .NET but also on other vendor platform offerings. With this has come a better understanding of the new possibilities for application architectures, specifically SOA which, I would argue, is the first expression of that understanding.

We are seeing the architectural ideas behind SOA being adopted in many areas of the overall system solution. These are not restricted just to the application layer because of resulting productivity and business benefits. For instance, integration, interoperability, management, operations, testing, security, data and user interface aspects of system solutions can each be viewed from the perspective of service-orientation. A so-called service oriented convergence (SOC) phenomenon is taking place, at least conceptually, amongst the architectural thinkers I have been working with lately.[1]

In this issue of JOURNAL there are three papers which will add some credence to this way of thinking especially as it relates to the role of service consumers in SOC and particularly the role of smart clients and task-oriented user interfaces.

We start with a paper by David Hill, a member of the Microsoft Architecture Strategy team, who contrasts thin and smart client architecture approaches and provides guidance on how to choose between them. David shows us that "smart clients are rich clients done right", leveraging new technology and techniques to avoid the pitfalls of traditional rich client applications whilst aligning neatly with the principles of convergent service-centric system models.

Ricard Roma i Dalfó from Microsoft's Office division discusses in his article a new metadata-driven approach to building task-oriented service consumers directly within Microsoft Office applications. The Information Bridge Framework (IBF) goes beyond a service's WSDL metadata by automating the construction of user-interfaces from metadata descriptions of the entities that constitute consumed services. It does this by describing entity views, their relationships and identity references, and operations which can be performed on them.

Performance is probably one of the least understood and misquoted metrics in the field of computing today. However, as a metric for system evaluation, it is considered by most users to be one of the most important and critical factors for assessing the suitability of systems. Richard Drayton of FiS and Arvindra Sehmi from the Enterprise Architecture team in Microsoft EMEA write a paper on benchmarking a scalable transaction engine which has an architecture based on loosely coupled, stateless, message processing components arranged in a queuing network. This architecture proved invaluable from a benchmarking perspective because it is backed up by sound mathematical techniques which enable an assessment of the impact on performance of various implementation realization techniques a-priori to deciding on any specific realization technologies. Therefore, measured benchmark metrics inherit and benefit from these mathematically credible foundations.

Pedro Sousa, Carla Marques Pereira and José Alves Marques, all from Link Consulting, follow with their paper discussing how, within the context of various Enterprise Architecture Frameworks each with their own concepts, components, and methodologies, the most important concern for architects is alignment. This concern may lead to consideration of simpler architecture concepts and simpler methodologies because the focus is not to define development artifacts but to check their consistency.

Next, Maarten Mullender, Solutions Architect in Microsoft's Architecture Strategy team introduces us to Razorbills which is a proposal for describing 'what' services do, thereby dramatically improving the usefulness of service consumption. There are similarities with the second article on IBF where the description is done in terms of entities, views, actions,

[1]The Nerd, The Suit and the Fortune Teller, Tech-Ed Europe 2004.

Keep updated with additional information at http://msdn.microsoft.com/architecture/journal

references and relationships but the emphasis is on helping business analysts and users to better bridge the gap between structured and unstructured, and formal and informal content and process. Maarten goes further to discuss the relevance of this approach in entity aggregation at informational and functional levels, user interaction and collaboration.

The final paper is a short piece on Next Generation Tools by Oren Novotny of Goldman, Sachs & Co. In this he argues that development tools must support the shift that languages have already made and fully support model-driven development. He insists source code files should be eliminated as they no longer serve a useful function in this context and the current rationale for using them is made irrelevant by better and different means in these next generation tools.

Please visit http://msdn.microsoft.com/architecture to keep up-to-date on architecture thinking at Microsoft and to conveniently download JOURNAL articles.

As always, if you're interested in writing for this publication please send me a brief outline of your topic and your resume to asehmi@microsoft.com.

Wishing you a great holiday season and good reading!

Arvindra Sehmi
Architect, D&PE,
Microsoft EMEA

Choosing the Right Presentation Layer Architecture

By David Hill, Microsoft Corporation

Abstract
The presentation layer is a vitally important part of an application – an inappropriately architected presentation layer can lead to too much complexity, a lack of flexibility and an inefficient and frustrating user experience. Thin client applications have well known benefits over traditional rich client applications in terms of deployment and manageability and this has led to their popularity in recent years. However, with the advent of smart clients, the choice of presentation layer architecture is no longer straightforward. Rich clients have evolved into smart clients that can combine the central management advantages of thin clients with the flexibility, responsiveness, and performance of rich clients. This article discusses the thin and smart client approaches and provides guidance on how to choose between them.

The Importance of the Presentation Layer
The presentation layer of most applications is very often critical to the application's success. After all, the presentation layer represents the interface between the user and the rest of the application. It's where the rubber hits the road, so to speak. If the user can't interact with the application in a way that lets them perform their work in an efficient and effective manner, then the overall success of the application will be severely impaired.

Personally, I think the term *presentation layer* really does not do justice to the function and importance of this layer. It is rarely about just presenting information to user – it's often more about providing the user with interactive access to the application. Perhaps a more appropriate name for this layer is the *user interaction layer*. For simplicity, though, for this article I'll stick with the commonly accepted name for this layer.

In any case, you'll want to design this layer to provide the user with the right user experience, so that they can interact with the application in an effective and efficient way. Of course you also need to architect and then implement this layer in such a way that it adequately takes into account the development, maintenance, and operational needs of the business. Choosing the right architecture for the presentation layer of the application is vitally important in order to achieve all of these goals.

The two commonly adopted approaches to presentation layer architecture and design are the *thin client* approach and the *smart client* approach. Of course, many factors influence the decision about which approach is best for a particular application – for example, client platform requirements, application deployment and update, user experience, performance, client-side integration, offline capabilities, etc. – and each has inherent strengths and weaknesses and naturally supports a certain *style* of application. You'll find, however, that the distinction between them can blur, and this can easily lead to the wrong basic approach being applied leading to problems later on.

For example, it is possible to provide a rich user interface with a browser-based presentation layer, just as it is possible to provide a completely dynamic user interface with a smart client. Neither would be very easy to achieve, and both would very likely result in unneeded complexity, a lack of flexibility and high development and maintenance costs.

Many organizations choose a thin client architecture by default without duly considering the alternatives. While they will not be appropriate for all scenarios, a smart client architecture can offer significant advantages over a thin client approach without incurring the downsides traditionally associated with rich clients. Organizations should carefully consider each approach so that they adopt the right approach from the outset, minimizing the TCO over the lifetime of the application.

In the following sections, I'll examine the thin and smart client approaches and some of the technologies behind them. For each I'll describe the basic architecture and discuss some of the design options within each. After that I'll talk about the relative strengths and weaknesses of each approach in terms of a number of common factors and requirements that you should take into account when determining the most appropriate approach for your application.

"A smart client architecture can offer significant advantages over a thin client approach without incurring the downsides traditionally associated with rich clients."

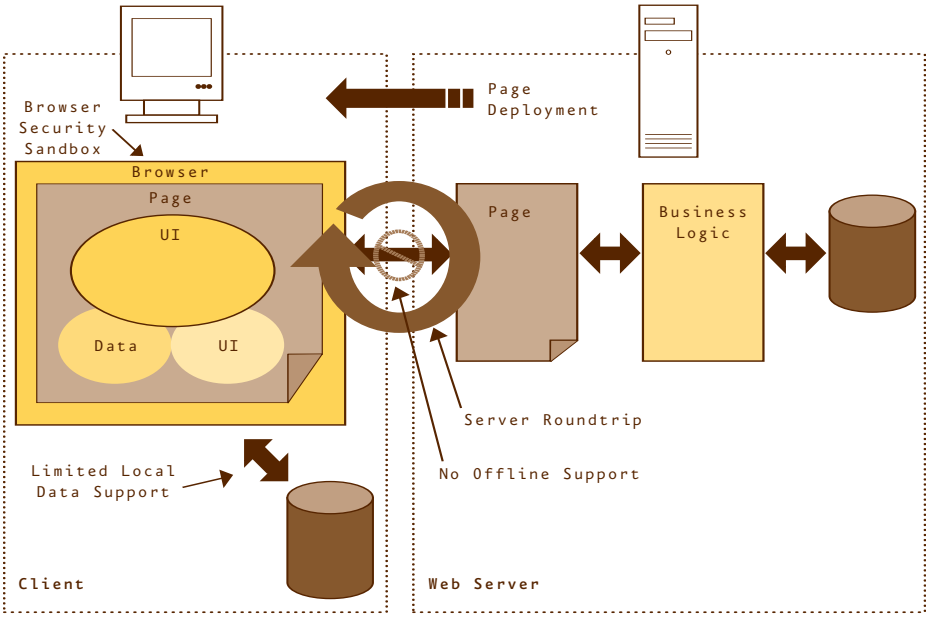


Figure 1: Schematic Overview of Thin Client Architecture

What Is a Thin Client?

Many thin client technologies pertain to the server side and there are many web server platforms and frameworks (ASP, ASP.NET, JSP, and others) to choose from. Each has particular features that try to make it easier to write thin client applications, but they all deliver the user interface to a browser on the client through a series of HTML pages. A thin client application is pretty easily defined as one that uses a browser to provide the execution environment for the application's (HTML defined) user interface.

In addition to rendering the user interface and allowing the user to interact with it, the browser also provides generic security, state management, and data handling facilities plus the execution environment for any client-side logic. For the latter, the browser typically provides a script engine and the ability to host other executable components; such as Java Applets, ActiveX and .NET controls, and so on, (though most definitions would not consider these executable components to be thin client technologies – see hybrid applications below).

An application architected to use a thin client presentation layer is decomposed into pages and each page is "deployed" to the client upon request. Each page contains the user interface description and, typically, a small amount of client-side script logic and a small amount of state/data (view-state, cookies, XML data islands, etc.). *Figure 1* shows a schematic representation of a thin client presentation layer architecture.

The browser has a limited ability to interact with the client environment (hardware and other software applications running on the client). It does provide a mechanism that permits storage of small amounts of data on the client (via cookies), and sometimes the ability to cache pages, but typically these facilities are of limited use except as a way to provide simple session management or tracking, and rudimentary read-only offline capabilities, respectively.

The browser also provides the security infrastructure so that different applications (pages) can have more or fewer permissions assigned so that they can do different things around state (such as cookies), they can host components, and execute scripts. Internet Explorer implements these facilities through different zones, trusted sites, ratings, etc.

In an attempt to provide a richer and more responsive user interface, some web applications have resorted to DHTML and similar technologies to facilitate a richer user interface. While these technologies are non-standard, in the sense that all browsers do not support them in the same way, they do provide the ability to include more advanced user interface elements such as drop down menus, drag and drop, and more with in a web page.

Other web applications have resorted to hosting complex components within

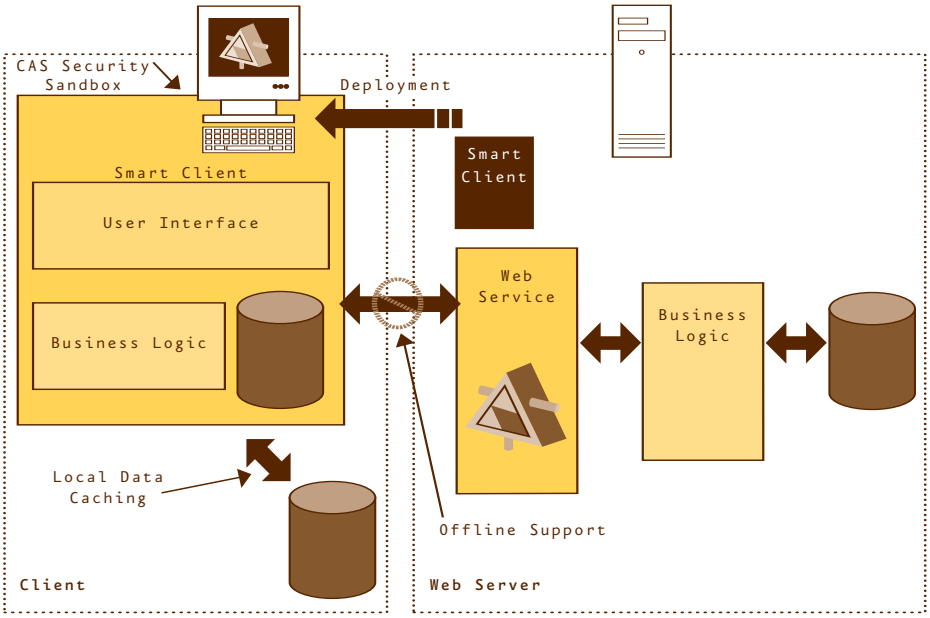


Figure 2: Schematic Overview of a Smart Client Architecture

the page including Java Applets, ActiveX and .NET components. These components provide either a more responsive user interface or client-side logic that cannot be implemented in script for performance or security reasons. This is where the thin client starts to overlap with smart client leading to so-called *hybrid* applications.

While you can certainly use such hybrid applications to leverage the strengths and weaknesses of each approach, in this document, I'll define the term thin client to mean a generic web application that does not rely on such components but just utilizes the basic facilities provided by the browser environment. Since hybrid applications need to rely on smart client capabilities to avoid management and operational problems, I'll describe hybrid applications in a later section when I discuss smart client applications.

What Is a Smart Client?

Smart client applications are maybe not as easy to define as thin client applications because they can take many different forms and are not limited to the one-size-fits-all approach of thin client applications. The essential difference between a smart client and a thin client is that a smart client does not rely on a browser to provide the execution, security and user interface environment for its operation. Also, smart clients, rather than HTML and Jscript, typically involve compiled code artifacts (components, assemblies, etc.) running on the client machine to provide the application's user interface and client-side logic.

How do smart clients relate to rich clients? Rich client applications have evolved into smart client applications. Rich clients offered many advantages over thin client applications including improved performance, responsiveness and flexibility, and the ability to work offline, but rich clients suffered from a number of operational problems when it came to deploying and updating them in a robust way. Thin client solutions of course excel in the deployment and update area and this is one of the main reasons for their popularity.

However, smart client applications represent a best-of-both-worlds approach by taking the manageability advantages of thin client applications and combining them with the benefits of rich client applications. Smart clients are rich clients done right, leveraging new technology and techniques to avoid the pitfalls of the traditional rich client applications.

For example, smart client applications built on the .NET platform can take advantage of a number of fundamental technologies that the .NET Framework provides to solve many of the problems traditionally associated with rich client applications. While it has always been possible to build rich client applications that minimized or avoided the deployment and security drawbacks, the facilities provided by the .NET Framework make it very much easier to do.

.NET provides the ability to deploy an application, or part of an application, from a web server. This technology, known as No-Touch Deployment, lets you deploy applications via a URL. This allows you to release applications to a central location (i.e. to a web server) so that the applications can be deployed automatically to the client on demand. All clients can be automatically kept up to date since the application checks for updates automatically each time the

"Smart clients are rich clients done right, leveraging new technology and techniques to avoid the pitfalls of traditional rich client applications."

application runs and each client application downloads the new code if required.

.NET also provides the Code Access Security (CAS) infrastructure. CAS assigns .NET code-specific permissions according to the *evidence* that it presents. CAS serves much the same function as the browser does in a thin client application, providing a sandbox environment in which the application operates. No-Touch Deployment integrates with (CAS). By default, applications that are deployed using No-Touch Deployment will be granted a restricted set of permissions according to the URL zone from which they were deployed. Network administrators can modify permissions using security policy so that the application can be granted, or denied, specific permissions according to the requirements.

Creating a smart client application with the .NET Framework provides less fragile applications. Traditionally, installing a rich client application could break other applications as it replaced important components and DLLs that were shared by other applications. .NET allows applications to be isolated, keeping all application artifacts in a local directory so that all assemblies are kept separate. Furthermore, such applications do not require any registration process when they are deployed, further reducing the risk of breaking other applications. In addition, the .NET Framework allows multiple versions of an assembly to be deployed side by side. This ensures that when an application executes, it runs with the exact versions of the assemblies that it was built and tested with.

An application architected to use a smart client for its presentation layer will typically provide a central deployment server, from which the smart client artifacts can be deployed to the clients, and a number of web services to provide access to the back-end business capabilities – business logic and data – and which are consumed by the smart client. Since the smart client is running code on the client, it can more cleanly separate the user interface from the client-side data and logic. In addition, depending on the permissions it has been granted, it can more freely interact with other client-side resources such as local hardware and other software running on the client. *Figure 2* shows a schematic overview of this architecture.

What does a smart client look like? Smart client applications can take many forms and the architect of such an application faces a number of design choices. The first decision to be made is to choose the most appropriate application *style* – the way in which the smart client is presented to the user. In general, there are three ways to design a smart client application:

– *Windows Applications*. Traditional Windows style applications, typically built using Windows Forms or mobile applications built on the .NET Compact Framework.
– *Office Applications*. Microsoft Office programs that are extended to include smart client capabilities, connecting the user to line of business applications and business processes.
– *Hybrid Applications*. Applications that utilize a combination of thin and smart client technologies. For example, by hosting Windows Forms controls within a browser page or by hosting the browser within a Windows Forms application.

Choosing the right application style is crucial if you want to fully realize the benefits of a smart client approach. Deployment, security, development, and offline capabilities all affect the choice of smart client application style but perhaps the biggest factor to consider is the overall user experience. Each choice represents a different type of user experience and choosing the right one will give the user the right combination of flexibility and performance they need.

Windows Applications
Users associate smart client applications with traditional Windows-style applications because they provide rich client functionality that includes toolbars, menubars, context menus, drag and drop support, context sensitive help, undo/redo, and more. Developers can build these kinds of smart client applications on the .NET Framework or the .NET Compact Framework using Windows Forms to provide these rich user interface features.

These developers can also take advantage of pre-built smart client functionality by leveraging the Application Blocks provided by the Microsoft Patterns and Practices group. These blocks provide the application with common smart client capabilities such as local data caching, seamless deployment and the ability to work offline.

Windows Forms applications provide the most control over the user experience, allowing the developer to craft the user interface and user interaction model to suit their exact needs. For applications which require

a specific user experience which cannot naturally be provided by any of the Office applications, this approach will be the best fit.

Office 2003 Smart Client Applications

Microsoft Office programs provide an extremely compelling platform for building smart client solutions. Extending Office applications so that they form part of a distributed solution, connecting them to remote data sources and business services, not only benefits the users, it also brings efficiencies to the developers that write the applications, and to those that must deploy and manage them.

There are many users who are familiar with Office and use it everyday in their work. Extending Office applications by connecting them to remote data sources and business services means that the solution can benefit from the user's familiarity, obviating or drastically reducing the need to re-train the user. The user benefits too, since they can continue to use the application with which they are familiar.

Many organizations extensively use Microsoft Office. Most business PCs – yours, your customers, your service providers, and your suppliers – have Office applications installed. Using Office as the client for line of business systems can reduce the need to install and maintain incremental client applications to access backend data sources and services. And very often, data from line of business applications is copied into Office applications such as Word or Excel for further manipulation, editing, analysis, and presentation. Copying and pasting is time consuming and introduces the potential for errors. More importantly, the link to the data is lost so the user needs constant refreshes, repeating the copy and paste process, and possibly introducing concurrency problems.

Office applications can also provide a lot of the functionality that is required to display and manipulate data, allowing the user to interact with the solution using the full power of Office. This can save a huge amount of time and effort allowing you to develop and release a solution must faster. For example, Excel provides powerful capabilities to sort, manipulate, and display data. Reusing these capabilities in your smart client solution can be very cost effective.

Of course, users have the ability to integrate additional functionality into their Office applications for a while. In some cases this has led to ad-hoc but business-critical solutions that are difficult to manage because they aren't developed or maintained by the IT department. Building these solutions using smart client technologies allows them to be more easily deployed and updated and represents a way to retain the value of the solutions whilst solving some of the manageability issues.

Office 2003 provides support for integrating smart client capabilities into Office application and connecting them to remote services that provide access to data and business processes. Some of the more important technologies Office 2003 supports for creating smart client solutions include:

– *XML Support*. Office 2003 provides a number of facilities that allow developers to more easily connect Office applications to remote data sources and business process through XML.
  – Word, Excel and InfoPath can use XML to store the structure and contents of a document in human or machine readable XML form. Microsoft has released W3C-compliant XSD schemas for these file formats and these schemas are freely available for everyone to use in their own solutions. These schemas allow Word and Excel documents and InfoPath forms to be easily constructed on the server and provided to the client through XML web services and users can readily display and edit these documents. This technology can also be used to provide document composition, indexing or searching functionality. And of course, since these documents are XML, they can be exchanged with any other system or process, providing a means for data interchange across heterogeneous systems. This technology is ideally suited to *document-centric* solutions.
  – Word, Excel, and InfoPath can also consume XML messages or documents that conform to custom or user-defined schema. Users can use their Office applications as presentation layer services in *data-centric* solutions where the business processes or services already define the message schema. This type of smart client application maps elements and attributes in the message to specific areas of the document so that the Office application can display them appropriately and allow the user to edit the values whilst ensuring that the data entered by the user conforms to the underlying schema. Specific values can be queried, set, or referenced programmatically using an XPath query statement.

"Extending Office applications so that they form part of a distributed solution, connecting them to remote data sources and business services, not only benefits the users, it also brings efficiencies to the developers that write the applications, and to those that must deploy and manage them."

– *Smart Documents*. Smart document solutions help the user to interact with a document by providing additional data and guidance to the user according to their current location within the document. As the user interacts with the document, it can display relevant information or guidance to the user using the task pane, or it can automatically fill in missing data according to the current task. Connecting this experience to remote services to obtain live data or to enable interaction with business processes allows powerful and integrated applications to be built.

– *Information Bridge Framework (IBF)*. IBF is a declarative solution that builds on smart document technology to allow documents to be connected to services through metadata. Smart tags within an Office application interact with the generic IBF infrastructure and the metadata associated with the available web services to provide access to relevant data and business processes from within the document according to the documents contents and the user's current activity. For example, if a user receives a document that refers to a specific supplier, the IBF infrastructure can access data about that company and display it in the task pane. It can also provide access to available options, allowing the document to be connected to other business processes.

– *Visual Studio Tools for Office (VSTO)*. VSTO provides access to the object models for Word and Excel to managed code extensions. Developers can build complex and comprehensive Office smart client solutions using VSTO to not only provide access to the full power of Word and Excel but also to all of the features of the .NET Framework, such as Windows Forms, that enable rich and responsive user interfaces to be easily integrated. VSTO also provides a superior development experience, allowing the developer to easily create and debug a solution. VSTO essentially provides the code behind the document to form a solution that leverages the facilities provided by the "host" application.

Hybrid Applications

Hybrid smart client applications combine the smart client and thin client approaches. They can provide a way to extend an existing thin client application with smart client capabilities, or a way to integrate a browser-based application into a smart client application.

For example, a smart client application may host an instance of a browser so that certain content and application functionality can be provided using the thin client approach. This architecture can be very useful when the application needs to integrate an existing thin client application, or when it needs to leverage a key benefit of the thin client approach to provide linked dynamic content provided by a web server. Of course, such content and functionality will only be available when the user is online but the smart client part of the application can be used to provide useful functionality when offline and enhance the application with access to the thin client functionality when online.

In some cases, the hybrid approach can be used to extend an existing thin client application by hosting smart client controls or components within a web page. These components can provide a rich and responsive user interface and specific application functionality (for example, rendering and visualizing data) while the rest of the application is delivered in a thin client way. However, this architecture is not suitable for providing offline support since the hosting web page will not be available without a connection, or for providing client-side integration of software or hardware unless suitable security policy changes are in place.

Choosing the Right Interaction Layer Architecture

Both the thin client and the smart client approaches clearly have their place. Each has its own strengths and weaknesses and the choice between them will depend on the requirements of a specific application or business need. The correct approach will provide the user with the right user experience so that they can interact with the application in an effective and efficient way, whilst adequately taking into account the development, maintenance, and operational aspects of the application.

Some organizations have a policy that dictates a thin client approach for all applications. Choosing a thin client approach by default can lead to significant technical problems for some applications because the browser platform is not able to easily support the requirements of moderately complex applications. Developing a thin client application to have the look, feel, and capabilities of a traditional rich client application can be extremely challenging and costly. Why? The browser imposes severe limitations on the developer in terms of state management, client-side logic, client-side data, and the provision of rich user interface features such as drag and drop, undo/redo, etc.

Conversely, choosing a smart client approach for all applications is not appropriate since it can result in overly complex solutions for applications that just present data dynamically and require the benefits of a highly dynamic user interface. Also, if your application must support multiple client operating systems, a smart client approach may not be appropriate due to cross-platform restrictions.

Adopting a single approach for all applications is therefore likely to result in unnecessary cost, complexity, a lack of flexibility and reduced usability. Both approaches can likely co-exist within the enterprise, according to the requirements of specific applications and the needs of the business. Choosing one approach over the other should be decided on a per-application basis, although in some cases both approaches can be combined, either by integrating thin and smart client technologies appropriately or by adopting a dual channel approach where some class of users can access the application using a thin client while users with more stringent requirements use a smart client. Either way, the key is to leverage the appropriate technology at the appropriate time to fulfill the expectations of the users and the overall needs of the business.

The thin client approach has well-known benefits in terms of reach and ease of deployment and operation. However, with the advent of smart client technology, smart clients are catching up in this area and are now a viable alternative to thin clients for many scenarios. In particular, smart clients do not suffer from the deployment and management problems that rich client solutions suffered from, and smart client solutions add benefits in terms of flexibility, responsiveness, and performance.

So, if deployment and manageability are no longer the dominant factors that influence the decision between the two approaches, how does one choose between them? With the erosion of the relative benefits of the thin client approach in this area, the balance is changing and the factors which have to be considered have increased. Depending on the relative priority of the requirements, one approach will be more suitable than the other, and choosing the correct approach will result in a faster and less complex development process, improved ease of use, and higher user satisfaction.

The features, advantages and disadvantages of each approach are described above but how do these translate into a decision given the requirements of a specific application? Some of the more important factors that companies must consider include:

– Client platform requirements
– Deployment and update requirements
– User experience requirements
– Performance requirements
– Client-side integration requirements
– Offline requirements

This is not a comprehensive list of factors and your company's IT department may add other factors that are critically important to specific applications. In particular, this list of factors is focused on operational or functional requirements and development and design-time factors have not been included here. Still, these factors may be of sufficient importance to sway the balance between one approach and the other.

Deciding on the right approach is a joint decision between the IT staff and business owners. The adopted approach should lead to a solution where both groups are happy: the IT from the management side, and the business owners from the functionality side.

Client Platform

Client platform flexibility can be important for a customer or partner facing applications where the principal users are external to the organization and for which a specific client platform cannot be mandated, or for applications that have to be accessible from non-Windows operating systems.

Thin clients offer the ability to target multiple client platforms, though this often requires the application to determine the exact type of target platform so that it can change its operation or behavior to accommodate the differences between the various browsers, especially when the target platform must include mobile devices. The thin client framework itself can handle many of these differences. For example, ASP.NET on the server can determine the target browser type and render content for each browser accordingly. Using some of the more advanced browser features, however, will likely result in having to develop specific code to handle the differences between browser types.

The smart client approach does not offer this capability although applications that target only Windows operating systems can use the .NET Framework and/or the .NET Compact Framework (for mobile applications) to deliver a smart client solution on a wide range of client devices, even for external users.

"The correct approach will provide the user with the right user experience so that they can interact with the application in an effective and efficient way, whilst adequately taking into account the development, maintenance, and operational aspects of the application."

If your application must support external users or clients running non-Windows operating systems, then the thin client approach should be strongly considered.

Deployment and Update
Both the thin and smart client approaches involve deploying the user interface, application logic, and data to the client. In both cases, these artifacts are centrally located and managed and deployed to the client on demand. In the thin client approach, these artifacts are not persistent on the client and need to be "deployed" each time the user runs the application. In the smart client approach, the client can persist these artifacts to enable offline usage or to optimize the deployment and update process.

Since both approaches allow the company to centrally locate the application's artifacts, they both provide centralized management with respect to user authorization, application deployment, update, etc. Companies can use thin client and smart client to provide solutions to ensure that users only run the absolute latest version of the application, though the smart client allows for additional flexibility, such as the ability for different users to run different versions of the application (for example, pilot groups), or for the application to be run offline. In order to realize these benefits, however, the solution may require additional security policy changes and/or an update manager component to be deployed to the client.

If your scenario requires that the application runs offline then you should strongly consider the smart client approach. On the other hand, if the application would not benefit from having artifacts be persisted on the client, then the thin client approach may make more sense. In the latter case, applications that primarily present dynamic data, or where concurrency problems (with respect to application logic or data) would be severe, tend to be better served using a thin client.

User Experience
Thin and smart client applications are each suited to specific user interface styles. Many thin client applications try to provide a rich user experience but they tend to fall short in certain important aspects due to the limitations of the browser when compared to a smart client platform. For instance, basic rich client features such as drag/drop and undo/redo are very difficult to develop in a thin client solution. The complexity associated with providing these features can be considerable and can reduce the cross-platform benefits of the thin client approach.

You should also consider how the user interacts with the application. Some applications are very linear because the user typically interacts with it in a pre-defined or similar way. Other applications are non-linear and the user may start one task only to suspend it, complete another task, and then go back to the original task. Managing the state required to provide such functionality can be challenging in a thin client solution. For example, thin client solutions have to be designed to handle the situation if the user presses the back button in the middle of an important transaction. Such a situation is easier to handle in a smart client application.

Smart clients can also take advantage of local resources to provide local data searching, sorting, visualization, and client-side validation to improve the usability of an application and enhance the user experience. Such features can lead to an increase in data quality and user satisfaction and productivity.

Performance
A significant difference between the two approaches is that smart client applications provide superior performance compared to their thin client counterparts.

At a basic level, a thin client typically uses script (which it must interpret on the fly) as the means to deliver and execute client-side application logic. In contrast, a smart client solution can deliver specific compiled code to the client. Additionally, and perhaps more importantly, client-side logic in a smart client application has fewer restricts in the way that it can interact with the user interface, local data storage, or with services located on the network. For these reasons, good smart client architecture allows the solution developer to more easily deliver a high performance solution.

The user's *perception* of performance depends on how they use the application and how they expect it to behave. Applications that are used infrequently or with which the user does not interact very much, for example, applications that simply obtain and display data, will not benefit from raw higher client-side performance. Applications that are heavily used, however, will appear to perform poorly if there is even a small delay for frequently used features. In a call center application, for example, a delay of four or five seconds to retrieve customer order details can easily add up to significant user dissatisfaction (and cost).

Of course, for functionality that sends or retrieves data over the network, both approaches will exhibit the same raw performance. However, a well-designed smart client solution can perform its network communication on a separate thread enabling the application to remain responsive while it sends and received data over the network. Such background work can be accomplished pro-actively, say in response to an incoming customer call. In addition, it is easier for smart client solutions to cache data locally, which can reduce the number of network calls or reduce the bandwidth required to perform the same function. These features can have a huge impact on the user's perceived performance of an application.

Smart client solutions can provide more rigorous client-side data validation. For example, because a smart client solution can cache data and logic locally, it is possible to cache read-only reference data that the application can use to provide field and cross-field validation. Applications that use such validation can provide early feedback to the user, improving the perceived performance of the application, reduce the number of times data gets transferred over the network, and ensure higher data quality. Thin client solutions may need to rely on complex scripts to provide the same level of functionality and may not be able to locally validate data with respect to other data that is not displayed on the current page.

A smart client solution can also take maximum advantage of the local processing, storage, and display capabilities to allow the user to query, sort, and visualize data on the client without the need to make a network call. This ability is especially evident when using an Office application such as Excel as the smart client host environment. This can result in a significant reduction in network calls to perform the same function for a thin client application.

Since the smart client solution user interface is typically provided by specific code running on the client, it can provide a more responsive user interface to the user. Rich client user interface features, such as drag and drop, undo/redo, context sensitive help, keyboard shortcuts, etc., can all help to improve the user experience, and with it the perceived performance of the application.

If performance is an important issue you should consider a smart client solution. A user's perceived performance of an application is often more important that the actual performance of individual operations. A good application's ultimate goal is to ensure that the user can perform their work effectively and efficiently in a way that user satisfaction is maintained.

Client-Side Integration

Often an application requires access to client-side resources so that they can be integrated into the overall solution. Sometimes the client-side resources include hardware (a printer, telephone, barcode reader, etc.) or software (integration of other line-of-business or desktop applications).

Of course, both the thin client and smart client approaches operate within a sandbox. In the thin client case, the browser provides the sandbox; for a smart client, the .NET Framework runtime provides the sandbox. Integrating client-side resources into a thin client application typically requires using a hybrid application architecture to host a component within a page (for example an ActiveX control) to extend beyond the browser sandbox. This approach is not very flexible and often depends on the user to make security decisions about downloading components to run on the client under the user's login account.

The .NET Framework runtime utilizes a more flexible approach and grants managed code permissions based upon the evidence that it presents and the local security policy. By default, code downloaded from a web server cannot interact with the local resources except in very limited and specific ways. However, your application logic can grant the code additional permissions to access to specific resources such as specific directories on the disk, access to other applications, local databases, etc.

This managed approach represents a more granular and more flexible mechanism for controlling the security aspects of an application, allowing the smart client to integrate other client-side resources without introducing a security risk. More importantly, the network administrator uses security policies to make security decisions rather than individual users, so the application code cannot perform actions or access resources for which it has not been granted permission.

Smart client applications often use code access security to control the caching of data and logic on the client. Such behavior is essential to providing offline capabilities and so these kinds of applications typically require security policy changes to grant specific permissions. Typically, this involves granting the application permission to cache the code and data on the local disk.

"It is important that the correct approach is applied from the outset to avoid unnecessary complexity, cost, lack of flexibility, and a poor user experience. "

If the solution requires access to client-side resources such as local hardware or other applications installed locally, then the smart client approach provides a secure and flexible solution.

Offline Capabilities

As organizations become increasingly dependent on their IT systems and the data and services that they provide, it becomes more important for users to be able to work offline. Providing support for offline access to data and services, using the *same* application whether they are on or offline, enables the user to remain productive at all times and helps to ensure consistency and data quality.

While network connectivity is becoming more and more ubiquitous, it is important to note that having a network connection is typically not enough to guarantee access to an application and the data and services that it represents. Line of business applications inside the firewall may not be accessible to users when they are out of the office unless the organization invests in a VPN infrastructure. Even in this case, forging a connection can be time consuming and expensive. Ad-hoc or brief access to the application is often not appropriate or possible leading to lost opportunities or data inconsistencies.

Sometimes users can plan to be offline. For example, you might have a sales worker who is out of the office for a specific period of time or a user who works from home. Sometimes, however, it is difficult to plan for an offline scenario. For example, a user in a warehouse with a Tablet PC may have a wireless connection that periodically drops. Another consideration is the quality of a user's connection. As organizations become increasingly distributed over the globe, network connectivity can suffer from high latency or low bandwidth problems.

In each of these cases, a smart client solution can provide robust access to the application so that the effect of connectivity changes can be minimized or eliminated. By intelligently caching data and logic on the client, and automatically deploying updates to both when required, the application can provide the user with a seamless experience independent of its connected state. In addition, a smart client can ensure that all network calls are handled on a background thread so that the application never has to wait for the network to respond, again allowing the user to carry on working regardless of the state of the network.

Realizing these goals with a thin client solution is very difficult to achieve. Some solutions try to solve this problem by providing the web application, or a subset of it, on the client using a local web server. Such solutions are difficult to maintain and require a complex infrastructure to ensure that updates to the application and its data are handled appropriately. Such solutions reduce the centralized management benefits that are often cited as the main reason for the adoption of a thin client solution, and they impose all of the other drawbacks inherent in a thin client solution.

Conclusion

Choosing the right presentation layer architecture can be critical to the overall success of an application. The right architecture will provide the right balance between the user experience, ease of development and testing, and the operational requirements of the application. Users are increasingly demanding that their part of this equation is taken into account.

The thin and smart client approaches are both well-suited to particular styles of applications. Recent advances in technology have redressed some of the imbalance between these approaches so that they need not be inappropriately applied to situations for which they are not suited. It is important that the correct approach is applied from the outset to avoid unnecessary complexity, cost, lack of flexibility, and a poor user experience.

Blanket corporate policies that favor one approach over the other are prone to incur these problems. An organization must carefully consider the overall needs of the application and compare these to the capabilities of each approach. The factors that can influence this decision are many and varied and this article has only touched on some of the more common ones. Invariably, the decision will come down to a compromise between the various factors. Understanding these factors, and their relative priorities, can help ensure that your organization chooses the right presentation layer architecture.

Resources

– Smart Client Architecture and Design Guide – Microsoft Patterns and Practices.
– Overview of Office 2003 Developer Technologies – MSDN.
– Overview of Office 2003 Developer Tools and Programs – MSDN.

David Hill
Microsoft Corporation
davidhil@microsoft.com

David Hill is a Solution Architect on the Microsoft Architecture Strategy team. For the last two years, David has been helping customers and partners architect and build successful smart client solutions on the .NET platform. He was a key contributor to the Patterns and Practices Smart Client Architecture and Design Guide, and the Offline Application Block.

Information Bridge Framework: Bringing SOA to the desktop in Office applications

By Ricard Roma i Dalfó, Microsoft Corporation

Introduction

Enterprises today are moving towards SOA as a way to expose their applications and data for consumption. By embracing SOA, enterprise assets like line of business applications or back-end systems can be used by a variety of solutions/applications built on top of the services exposed by those assets. In this world you can look at an enterprise as a set of services that expose sets of data or functionality and encapsulate the business logic behind them.

Building solutions on top of these services is fairly easy today using existing development tools. Different vendors provide tools for both exposing and developing on top of those services by using standards like SOAP or WSDL.

Once enterprises start developing a few of these solutions problems start to arise. Here are some of the most common problems:

a) Solutions are one off. They only talk to one or a set of predefined services and the solution itself is hard to reuse. Changes on the services require a rebuild/redeployment of the solution.
b) Knowledge about *what* the service exposes is in people's heads rather than in the definition of the service itself. Current standards only cover the *how* you get to those services.
c) It is hard to bring different services together. There are no predefined aggregation mechanisms and there is no definition on how one service relates to another (services do not know about each other).
d) Solutions UI is hard to do and usually poor (unless huge investments are made) by most common users standards. It is hard to emulate current off the shelf applications UIs in a one off solution.
e) Most users are fairly familiar with applications like Office Suite (Word, Excel, Outlook, etc.) but they need to be trained if a new application/ solution is rolled out increasing the costs of such deployment.

Because all of the above a better mechanism for building solutions on top of existing services is required.

The Metadata approach

Today web services expose quite a bit of information about how the service can be consumed but offer very little help in understanding what type of information or functionality is offered. Web services usually expose WSDL so tools can easily discover what methods and parameters the web service exposes but offer little clues about what business entities are defined behind those methods or even if they affect the back-end systems at all (no way to say if a method will update the back-end system for example). It seems that WSDL is not enough for representing what today's services expose.

We propose a new set of Metadata that is associated with a service and explains the kind of things that a user of the service (a solution developer) will require to know.

In this new Metadata we will expose concepts like:

a) Entities – Abstract business or user definitions that will encapsulate a set of data or functionality. For example, we can have a Customer entity.
b) Views – A schema associated with an entity that describes a subset of data about it. For example, for the Customer entity we may have several views like Customer Contact Information or Customer Financial Information. Each view complies to a particular schema and is a representation of the entity for a given context.
c) Relationships – Entities/Views can be related to others and these relationships should be described in this Metadata. For example, Customer entity may be related to an Orders entity. Relationships allow for navigations between the entities by just executing the Metadata description. A relationship then will describe how to get from one entity into another.
d) References – A reference is a common way to point to a set of information. It is a schema and represents the minimum set of information needed to retrieve a piece of data, e.g. Customer Id for retrieving a Customer. There can be multiple ways you can retrieve a piece of information, e.g. a customer could be retrieved by name, Id, SSN, etc.
e) Operations – These are the methods that are available for a given entity/view to operate on. You could think of GetCustomer or UpdateCustomer or ReleaseOrder as examples of such operations.

Describing Metadata for existing Services only solves half of the equation. The other half (the solutions developed on top of these services) also requires a Metadata description. We think that you can build most solutions by thinking in terms of Actions that are executable by the end user. These Actions are constructed on top of the Services Entities/Views and provide actionability on top of them. A Customer action will certainly have an Action to display its data and maybe another action to update it. The action

"Today web services expose quite a bit of information about how the service can be consumed but offer very little help in understanding what type of information or functionality is offered."

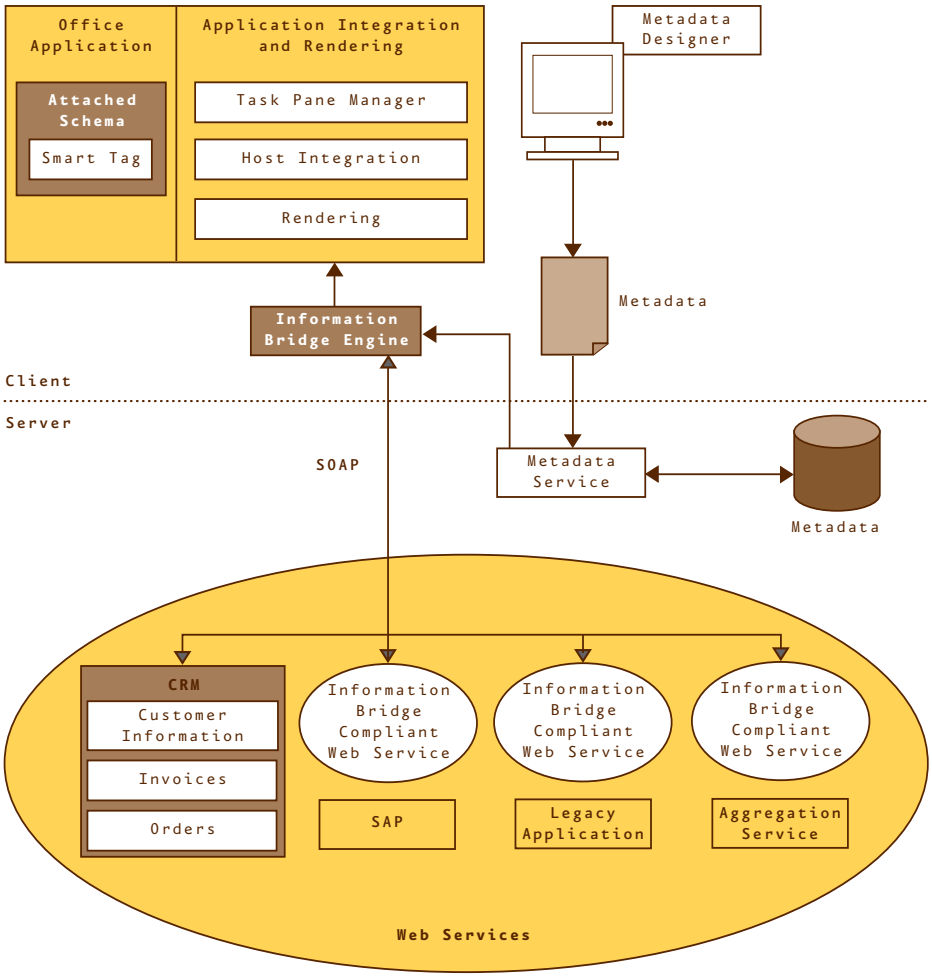


Figure 1: IBF Architecture

description should link the data retrieved from the service into the UI or solution functionality that will use it.

Information Bridge Framework

Information Bridge Framework (IBF) is the Microsoft response to the above challenges and Metadata approach. IBF allows connecting LOBs and back-end systems with Office applications and creating solutions on top of web services via a Metadata approach.

IBF accomplishes several things:

a) Create Metadata description for services
b) Create Metadata infrastructure for building solutions/applications on top of services
c) High level of reusability across solutions
d) Easy maintenance and deployment of solutions
e) High level of integration with Office applications
f) Very low learning curve for existing Office users

Information Bridge Architecture

IBF architecture (as seen in *Figure 1*) includes the following components:

a) An IBF compliant web service that encapsulates the LOB or back-end system. We discuss the compliance issue in the next section (Designing and developing an IBF solution).
b) A Metadata repository (Metadata Service) that includes both the Service and the Solution Metadata. The repository is exposed itself as a web service that provides access to the Metadata. There is one central repository where all Services and Solutions are described. Clients will download subsets of this Metadata on as needed basis for execution based on their permissions.
c) IBF Client Engine. This last piece has two distinct components:
   a. The Engine which downloads the Metadata from the Metadata Service when needed and keeps a local cache of it. It also understands Metadata and executes it based on the current context. It performs all the non UI related operations like SOAP calls, transformations, etc. This component is UI agnostic.
   b. The UI Engine which is the part that understands about the application where it's being hosted (Word, Excel, etc.) and will render the UI and provide services specific to the host application. It creates an abstraction layer on top of the hosting application so Solutions built with IBF don't need to know about differences between hosting apps.
d) Metadata Designer is a Visual Studio based tool that allows for editing/importing of Metadata to the Metadata Service.

"We propose a new set of Metadata that is associated with a service and explains the kind of things that a user of the service (a solution developer) will require to know."

Designing and developing an Information Bridge solution

When designing an IBF (Information Bridge Framework) solution you must separate it into three distinct blocks (see *Figure 2*). On one end you need to describe an IBF compliant web service that encapsulates the functionality of your back-end application that you want to offer your end users. On the other end you need to design the UI and experience that you want to offer to the users of your solution. The final step is to link both your Service and the UI solution you have built by using IBF metadata. By separating these three stages you can allocate different resources to each one of them and then can operate in an independent manner and only agreeing on the interfaces (or common schemas) that they will share.

Creating an IBF compliant Service

IBF requires Services that will provide the data and interaction with the data needed for your solution. IBF currently supports two kinds of Services: Web Services and CLR Components. Web Services are the most common way to expose back-end data, and most of the IBF examples use them for the Service description. If you require offlining of data or caching (for performance reasons) a CLR implementation is also possible.

When designing a Service for IBF you should keep in mind that you are building a Service for user consumption, so you want to expose data and methods that are meaningful to the user.

There are also a few concepts you need to be aware of when building these Services:

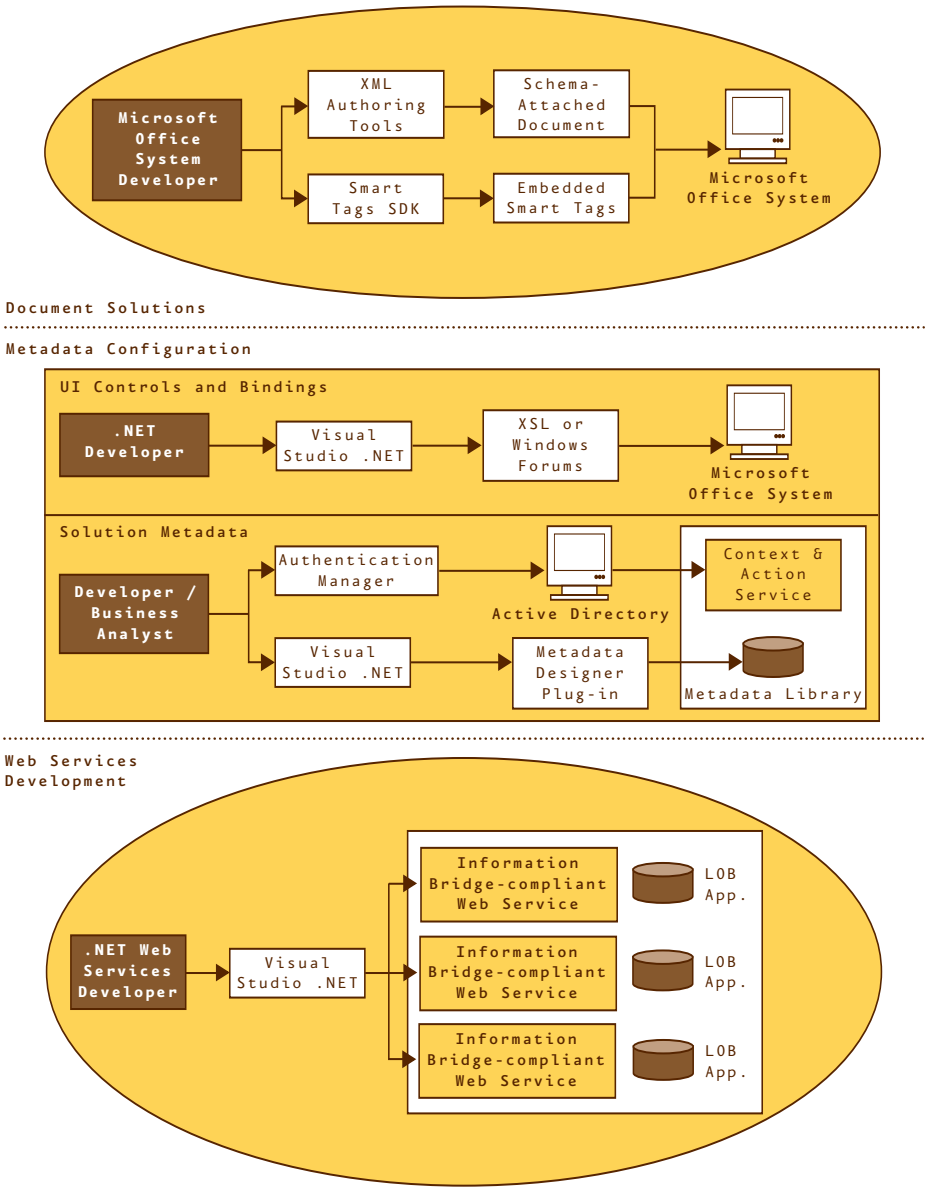


Figure 2: The three distinct blocks of an IBF solution

– Entities – You can think of an entity as a business object that has a particular meaning to the user and that the user will be able to act upon. An example of an entity can be Customer, Order or Opportunity. All of those have some data associated with them and are actionable from the user point of view. For example, the customer entity might have data associated with a particular customer (name, address, location, etc.) as well

"IBF allows connecting LOBs and back-end systems with Office applications and creating solutions on top of web services via a Metadata approach."

as methods that allow the user to act on the entity, such as UpdateCustomerInformation or SendEmailToCustomer. It might also be a starting point to other entities via relationships like CustomerOrders or CustomerOpportunities.
- Views – IBF partitions entities in different views. A view is a subset of information related to the entity. For a customer you might have a Customer Contact Information view and a Customer Financial view.
- References – A reference in IBF world is a piece of information that uniquely identifies an instance of an entity/view. For the previous example a reference could be Customer Id or Customer Name if that allows you to uniquely identify the customer.
- Relationships – Some of the entity/views will have relationships between them and the Metadata we build should describe those. An example would be Customer and Orders since you can relate a Customer with its orders and an Order with its Customer.

Based on the previous concepts when you build a Service you will identify three different kinds of methods:

- Get – A get method is one that allows you to retrieve the data for an entity/view by passing a Reference. An example would be a method called GetCustomerContactInformation that would accept a Customer Id Reference parameter.
- Put – This is a method that allows you to modify the content of an entity/view by updating the back-end system. It accepts two inputs, the Reference to the entity/view to update and the data to be updated.
- Act – This kind of method allows for doing things that are not related to getting/updating an entity/view or across multiple entities.

When you understand these concepts, you can build a Service around them. The service will expose a collection of methods of type Get/Put/Act and by doing so will define the schemas for the references and the views (data return by Get operations).

For the Service to be complete it has to expose IBF Metadata that describes the previously explained concepts. IBF provides tools that automatically generate Metadata from a web service and you can then increment the Metadata by annotating the methods exposed in there around Entities/Views and map them to the right References.

Creating UI components

IBF allows your documents to contain live links to back-end data. The way these documents contain information about what back-end data to obtain is via SmartTags or by having an attached schema to the document. The SmartTag or element node in the schema will store information about what back-end piece of information is pointing to. As discussed in the previous topic on how to create an IBF Service, these are References. SmartTags, for example, will contain a Reference to the back-end piece of information. Your solution will have to define how it wants these SmartTags to get into the document and IBF provides/recommends several ways for doing so. You can automatically generate a document with the SmartTags embedded (this would be useful if the emails/documents are dynamically generated by some process); you can use a SmartTag recognizer to detect pieces of text based on a regular expression or by doing a look up and dynamically insert a SmartTag in them; and you can also use the built in Search capability in IBF for the user to find the instance of information they are interested in and allowing them to paste it into the document.

The remaining UI pieces are what will get displayed to the user. IBF provides a Window Pane approach that hosts regions that are fully definable by the solution provider. IBF supports .NET CLR controls and HTML regions (and menus for those regions). Creating a piece of UI is just a matter of creating a control and implementing one interface that will get the data into the control. The control itself does not need to know how or where the data is coming from. The control only needs to know the type of data that will be provided. IBF will dynamically instantiate the control at run time and will pass the right data to the control. This allows for a separation of displaying of data from how you get to that data. Following the previous example you could create a control that knows how to render Customer information (it knows about the schema of a Customer and that it contains its name, address and so on).

Creating Solution Metadata

The final step for creating an IBF solution is to create the Metadata that will link the Service description with the UI elements that have been defined for it. IBF provides a few concepts that allow for easy creation of these Metadata based solutions:

- Actions – These are the executable units from a user point of view and can contain both Service and UI

methods/operations. In the previous example you would have a DisplayInformation action that would use the Service entity/view on CustomerContactInformation and would link it to the user control we created for displaying customer information.

– Transformations – Because data from the Service and the data required by the UI elements might not be the same, IBF allows you to transform the data. XSL transformations, regular expressions or calling CLR components are all supported ways to transform data.
– Relationships – Your solution may have relationships beyond those provided by the Service. It may also know about relationships across Services. As an example I may be able to relate a Customer in one of my legacy apps with my Customer in my CRM system.

Deployment and Security

You can think of IBF as a central repository of Metadata, Service description and UI elements that will be deployed dynamically as solutions are used by the IBF client components. No code/Metadata other than the IBF client needs to be installed on client machines. The IBF client component connects to the appropriate Metadata Service to obtain all Metadata and UI elements needed for a given context. After it has the Metadata description and UI elements, it will execute them along with the Service method calls, and it will construct the UI and user experience as needed.

Because IBF uses CLR components for UI rendering it builds on top of .NET security, all components are dynamically downloaded and cached locally, and are executed in a sandboxed environment so they cannot harm the client machine. If you need your controls to have a higher level of control, you can sign those controls and increase their privileges by using standard .NET Security Policies.

This provides for a robust and deployment free environment for your enterprise solutions.

Conclusion

IBF, by separating the Service layer from the UI layer and linking them via Metadata, allows for a high level of abstraction and reusability of both your Services and your UI components. This provides a very powerful platform for specifying the back-end assets in an enterprise and creating solutions around them that can be linked or combined without coding. This Metadata approach adds a lot of flexibility and allows for further refinement of solutions around customer scenarios in a Metadata driven approach. IBF provides powerful UI constructs to help build a complete UI experience and integration with Office applications. It also provides for a secure and deployment free environment of new solutions by building on top of .NET technologies.

Resources

For more information about IBF go to the site:
http://msdn.microsoft.com/ibframework

Ricard Roma i Dalfó
Microsoft Corporation
ricardrd@microsoft.com

Ricard Roma i Dalfó is the development lead for the Information Bridge Framework project. He is working on next versions of IBF and solving connectivity problems with Line of Business applications. He had been previously in the Office team and helped release Office 2000, XP and 2003 in various development roles. He holds a M.S. in Computer Science from Polytechnic University of Catalunya.

Benchmarking a Transaction Engine Design

By Richard Drayton, FiS and Arvindra Sehmi, Microsoft EMEA

Performance is probably one of the least understood and misquoted metrics in the field of computing today. It is common practise amongst technologists and application vendors to regard performance as an issue that can safely be left to a tuning exercise performed at the end of a project or system implementation. In contrast, performance, as a metric for system evaluation, is considered by most users to be one of the most important and indeed critical factors for assessing the suitability of a system for a particular purpose. This paper describes a benchmarking exercise undertaken during October 2002 at Microsoft's ISV Laboratories (ISV Labs) in Redmond, Washington as part of joint project between the Capital Markets Company (Capco) and Microsoft.

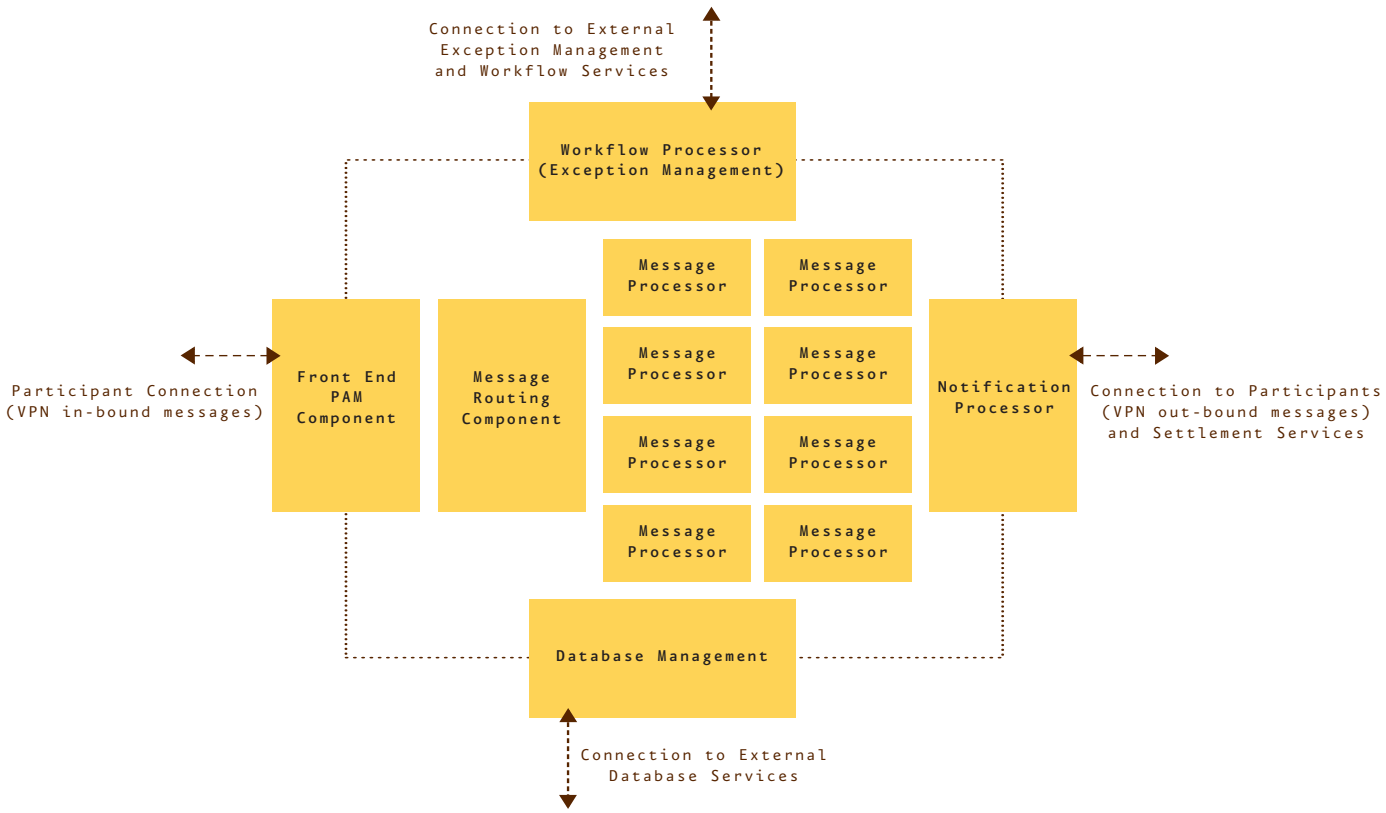
The project was initiated when Capco was commissioned by the Singapore Exchange Limited (SGX) to provide a business assessment and to develop a technical architecture for the exchange's centralised transaction processing utility. The utility was to provide matching services for post-trade, pre-settlement interactions of participants in the equities and fixed income trading areas of the Singapore market. The design of the main processing engine was done following a process known as Software Performance Engineering (SPE) [SMITH90, SMWIL02] in which the entire design and validation exercise is modelled from a performance perspective rather than from a traditional object-oriented design perspective.

Two subsystems were created by Capco, known respectively as STP Bridge (a communications infrastructure and exchange gateway) and STE (a scalable transaction engine). Both were used in the benchmarking exercise.

The Processing Model

The architecture of STE was based on loosely coupled, stateless, message processing components arranged in a queuing network for high scalability, high performance, and extremely high

Figure 1: Processing Engine Model



"Unrealisable performance is a common characteristic of benchmarks"

transaction throughputs. For SGX, a worst case transaction processing load of approximately 600 messages per second was estimated by analysis of their trading history over the previous few years; which turned out to be the highest processing level required during the previous Asian financial crisis in 1998 when exchange dealings were considered abnormally high. This value was used as the baseline/target processing load. A stretch load target of 2000 messages per second was set to ensure the architecture, if successful, would have adequate headroom to cope with future expected trading volume growth and volatility.

The decision to use loosely coupled components communicating through message queues rather than through more traditional component interfaces (API's) requires information to be passed from component to component either directly through the messages themselves (*persistent message flow*) or by enriching the messages as they pass through various processing steps (*temporal message flow*). The processing components themselves are largely independent of each other and stateless. This results in benefits such as lower software development risk and realisation costs for individual components together with higher scalability and flexibility characteristics of the processing engine as a whole when compared to traditional monolithic application development approaches. Most conventional designs support only one of the two possible scalability dimensions: scale-up – increased processing power through increased processor resources (memory, CPU, etc.), or scale-out – increased processing power through increased number of processing nodes. This architecture supports both types of scaling.

The overall architecture of the STE processing engine is shown in *Figure 1*.

This consists of a number of STE components which have responsibility for processing sub areas of the underlying trading business activities. The business process support provided by the engine is realised by breaking down the entire trade lifecycle into a set of related atomic messages. Each market participant generates and receives various subsets of these atomic messages during execution of the trade.

An implication of the architecture is that the business process itself must be capable of being represented by fully asynchronous commutative set of operations, that is, it must be able to process messages in any order. This removes the necessity to synchronise message processing and business operations throughout the engine, a task which would result in an incredibly slow and complex application. Note that synchronization is different to correlation of messages which is applied in normal processing. Several other "autonomous computing" requirements are catered for in the processing model. Amongst these are the notions of distrusting systems, idempotent operations, state management, message correlation, context management, tentative operations, message cancellation, and transaction compensation[1].

Business-level message flow through the processing engine is based, in part, on the execution model proposed by the Global Straight Through Processing Association (GSTPA) which proposed a similar centralised utility model for cross-border trade settlement. The SGX processing model was likely to require operational links to other centralised utilities like the GSTPA, so the message sets used were based, in part, on those used by the GSTPA to help with inter-operability in the future[2].

The business process for the Singapore market reduced to a set of four principal message types, namely, Notice of Execution (NOE), Trade Allocation (ALC), Net Proceeds Allocation (NPR) and Settlement Instructions (SET). The process surrounding these message types involved a number of interactions leading to a set of some 35 message variants that formed the entire business operating model. As recent events in the financial services industry illustrated, the range of processing volumes that could be experienced by the utility would be comparatively large. For this reason a heavy focus was placed on the ability of the architecture to support high scalability requirements.

Modelling Performance

A valid benchmarking exercise allows other institutions to repeat the benchmark for themselves and to be able to achieve similar results. It was also important to support the

[1] Pat Helland's "Autonomous computing: Fiefdoms and Emissaries" [PHEL02] webcast gives more details on the autonomous computing model.

[2] The GSTPA is now defunct but this unfortunate event has no impact on the substance of this study. Additionally, the original proposal championed by the GSTPA could not be used directly in the Singapore market, so specific processing schemes were created for the SGX processing engine.

arguments for the architecture model through sound mathematical techniques which would enable an assessment of the impact on performance of various implementation realisation techniques a-priori to deciding on any specific realisation technologies. *Figure 2* shows the queuing network model used for the performance analysis in the benchmark. Note, this works hand-in-hand with a client-side Participant Access Module (PAM).

By conducting the benchmark in this way it was felt that the results of the exercise, good or bad, would be credible and valid and that the results would serve as the basis for a case study into the application of queuing network models for the processing of Post Trade, Pre-Settlement information as part of an overall high performance straight through processing (STP) initiative in the financial services industry.

Measuring Performance

One of the most frustrating aspects of performance engineering occurs when a performance unit which permits valid comparisons to be made between similar systems is not identified.

Performance is a subjective quantity whose absolute value is often determined solely by the user of a system and not by any systematic algorithm or process. This nebulous aspect of performance gives rise to variations in the perceived value of performance characteristics for a specific system even without accompanying physical changes to the underlying technology whatsoever.

The results of any benchmark are subject to wide interpretation and this can jeopardise any comparative analysis of software systems. *Unrealisable performance* is a common characteristic of benchmarks that has little relevance to a user's perspective of performance. The existence of unrealisable performance can be shown in numerous examples of published benchmarks where results often indicate high levels of performance that are simply unobtainable in the real world. For example, an ADSL connection offers a theoretical download speed of 512 K, but in reality is limited by contention with other users on the same exchange. So, a more realistic way to do performance comparisons between systems is to establish metrics for an operational situation which can be readily reproduced outside the test environment and is meaningful from the system user's perspective.

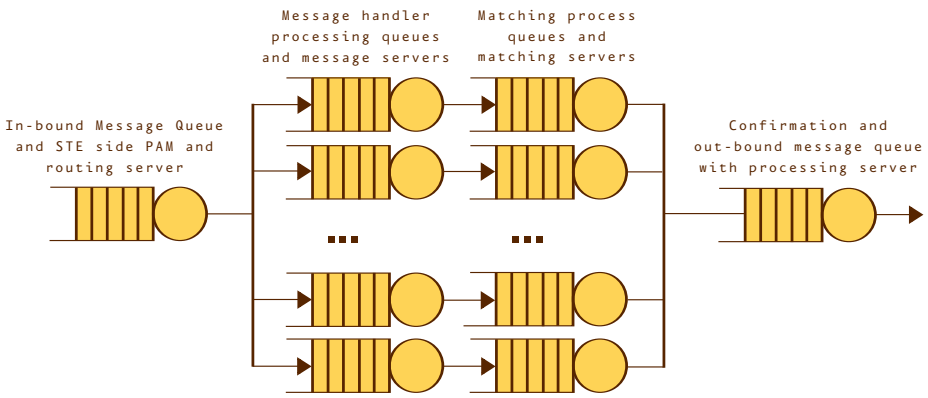
To avoid these sorts of frustrations with the benchmarking process, a practical technique was required which would provide a supportable and reproducible figure for the performance of the realised system. The technique chosen was based on Buzen and Denning's work [BDEN78] allowing credible performance figures to be obtained based upon sound mathematical principles.

The Operational Benchmark

The purpose of the benchmark was to validate the architectural design on Microsoft platform technologies and to establish the credibility of the benchmark in an environment highly correlated with the operational requirements that would be presented to potential system users. This goal could only be achieved if the benchmark tests were firstly performed using volumes and process loading levels consistent with those that would be experienced in the final realised utility, and secondly performed in a manner that could be reproduced on-site as well as in the laboratory.

For the laboratory exercise a set of test management components were created which allowed an end-to-end analysis to be performed. These consisted of a scalable message driver and corresponding scalable message sink with performance measurements being taken from these two points. The performance figures were calculated on the time taken to fully process a known number of messages forming a known number of financial trades.

Figure 2: Queuing Network Model

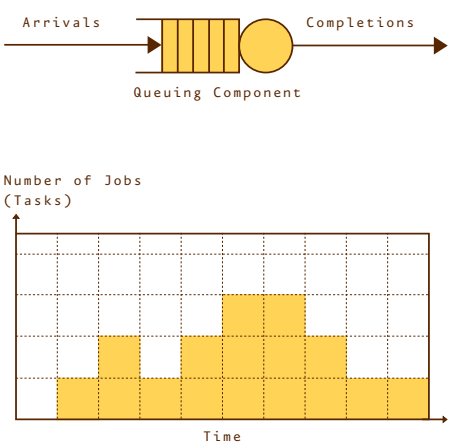


The Buzen & Denning Method

Buzen and Denning [BDEN78] described a practical technique for evaluating the performance of queuing network software systems. In essence, they argued that a simple and easily obtainable measurement of queue length over time could be used to establish all the performance metrics which they had defined for queuing networks. A realisation of the Buzen and Denning technique produces an execution graph for the network model similar to that shown in *Figure 3*.

A simple queuing model can be viewed as a queue together with an associated server, as shown in *Figure 3* where messages arrive (arrivals) on an in-bound queue on the left hand side, are processed by the server, and exit (completions) on the right hand side. The Buzen and Denning technique involves sampling the queue length of the server at regular intervals and recording the number of message remaining in the queue and continuing the observations over a period of time.

Figure 3: Simple Execution Graph



The resulting graph of observations is known the execution graph and clearly shows the arrivals (i.e., increases in the height of the graph) and completions (i.e., decreases in the height of the graph) of messages. Using these figures, Buzen and Denning derived a set of formulae from an original result established by J.D.C. Little (known as Little's Law [LITTL61]) which enabled all of the required performance metrics to be determined. A summary of the formulae is shown in *Table 1*.

Measuring Performance

To complete the performance measurement a standard set of business messages was needed representing the mean business process life cycle for the utility. A standard "trade" was created using an NOE (notice of execution) message, two ALC (trade allocation) messages, two NPR (net proceeds) messages and two SET (settlement instruction) messages. A standard trade therefore consisted of 7 base messages which together with

Table 1: Buzen and Denning Formulae

Metric	Symbol	Definition
Length of Observation	T	Total number of time units over which the observation has been made.
Arrivals	N	Total number of Arrivals over the length of observation.
Completions	C	Total number of Completions over the length of the observation.
Busy Time	B	The number of time units where the number of messages in the system exceeds zero.
Utilisation	U	The calculated value: $U = \frac{B}{T}$
Throughput	X	The calculated value: $X = \frac{C}{T}$
Mean Service Time	S	The calculated value: $S = \frac{B}{C}$
Execution Distribution	A	The calculated value: $A = \sum_{t=0}^{T}(Messages_t)$
Mean Queue Length	L	The calculated value: $L = \frac{A}{T}$
Residence Time	RT	The calculated value: $RT = \frac{A}{C}$
Queuing Time	Q	The calculated value: RT – S

"One of the most frustrating aspects of performance engineering occurs when a performance unit which permits valid comparisons to be made between similar systems is not identified"

Table 2: Message Matching Criteria

Criteria	Definition
Trade Reference	The trade reference identification information must match for all elements of the trade.
Buyer Reference	The Banking Identification Code (BIC) for the buyer involved in the trade for all elements must match.
Seller Reference	The Banking Identification Code (BIC) for the seller involved in the trade for all elements must match.
Security Reference	The security reference code (ISIN, QSIP, SEDOL etc.) must match for all messages in the trade where security designation is included.
Security Type	The security reference type (ISIN) must match.
Trade Date	The specified date of the trade must match on all messages containing a trade date. Additionally, the trade date must be a valid trading day for the locations specified in the messages.
Settlement Date	The specified settlement data of the trade must match on all messages containing a settlement date. Additionally, the settlement date must be a valid trading day for the locations specified in the messages.
Trade Currency	The trade currency designation of the trade must match on all messages where the trade currency is specified.
Settlement Currency	The settlement currency designation of the trade must match on all messages where the trade currency is specified.
Trade Quantity	The quantity of the security or fixed income trade must match in all relevant messages. This criteria required also that the sum of the allocations for a trade matched the execution quantity.
Trade Type	The type of the business trade being performed must match on all relevant messages.
Trade Direction	The buy/sell status of the trade must match on all relevant messages.
Unit Price	The price per unit traded must match on all relevant messages.
Trade Fees	Fees and commissions contained within the trade must match on all relevant messages.
Country Reference	Base country designations must match on all relevant messages.
Gross Amount	The calculated value of quantity * price must match the value contained in the relevant messages.
Rate	The fixed income interest rate must match on all relevant messages.

the requisite acknowledgement and confirmation messages between participants comprised the entire information set for the tests. The completion of a trade's processing by the system generated one further message which was designated the CLS (clearing and settlement) message.

In the GSTPA specification a combination of up to seventeen different elements in the messages were required to match before the entire trade could be considered valid. The requirements differed depending on the particular message types being matched but at some point all seventeen matching criteria had to be applied for the trade to be processed. The full set of matching criteria is shown in *Table 2* for reference. The matching process entailed scanning the message database matching table for all elements of the trade specified in each message as each message was received. When the scan yielded the required seven matching atomic messages a valid CLS message was sent to the out-bound queue of the test system.

In addition to the matching process, the business process for the Singapore market also requires validation of the contents of the messages for items such as currency and country designations, security reference codes, payment and settlement dates and participant identification information. A collection of static data was created to form the base reference data for the utility and is shown in *Table 3* for reference.

To establish the benchmark result test message sets were generated using the static data. A message builder component was used to randomly select appropriate static data and then

Data Item	Records	Comment
Participant	38	Reference data designating the valid participant codes available to the system.
Security	20,000	Reference data designating the valid security codes available to the system. For the Singapore market only ISIN codes are used.
Currency	172	The ISO 4217 standard codes for currencies.
Country	240	The ISO 3166 standard codes for countries.
Profile	38	The profile data for active participants. The profile static data contained the X.509 public key data for the signed messages to be returned to participants and the private key data for the participant designed as the utility. All messages were digitally signed and check for each message transferred.
Routing Data	44	Internal and external queue routing information. Each participant was designated an out-bound reference queue to which relevant messages were routed. Internal queue references for the grid computing part of the processing utility were also contained in this data block.
Queue Data	44	Additional queue information relating to the host machine IP address and TCP/IP post number for data connections (used on the MQSeries version of the software only).

Table 3: Static Data Parameters

Machine	Operating Sys.	Support Software
PARC4603 PARC4602 PARC4504 PARC4503 PARC4502 PARC4501 PARC4202 PARC4201	Windows Server 2003 build 3680.main	MSMQ 3.0; Capco STE 3.0 (C/C++) with Microsoft Libraries
PARC4208 PARC4107 PARC4507 PARC4508	Windows Server 2003 build 3680.main	MSMQ 3.0; MSSQL 2000 sp2

Table 5: Software Operating Environment

combine it into the standard trade of seven independent messages. Two core message sets were created; the first contained blocks of 250,000 trades (1,750,000 messages) and the second contained 1,000,000 trades (7,000,000 messages). A normal business cycle would require all matched or unmatched trades existing in the system after 3 days to be cleared from the system database; however, it was decided to keep all processed tradeinformation in to investigate the degradation in database performance as volumes increased. The message sets were stored in standard text files in XML format ready for transmission into the processing engine.

To monitor the benchmark and obtain the required performance metrics a set of management components was constructed. These were a message driver component, a message sink component, and an additional component to monitor the queue lengths of both the in-bound and out-bound message queues as well as the internal queue lengths at regular intervals. The message driver component processes the test message set files sequentially and applies a digital signature prior to sending them to the in-bound queue of the processing engine. Each message driver running on its own without contention with other processes in the system can achieve in the region of 8000 messages per second. This figure is very close to the benchmark results provided by Microsoft for the MSMQ product. A message sink component reads the information destined for the CLS queue and monitors the time taken to process a given number of messages through the system. The monitoring components are shown in *Figure 4*.

"The results of any benchmark are subject to wide interpretation and this can jeopardise any comparative analysis of software systems"

Machine	Manufacturer	CPU	RAM	Disk	Network	Ext. Storage
PARCXXXX	Dell 1550	2	2GB	36GB	Gigabit	
PARCXXXX	Dell 1550	2	2GB	36GB	Gigabit	
PARCXXXX	IBM X370	8	8GB	2 x 73GB	Gigabit	Compaq MSA 1000

Table 4: Basic Hardware Environment (x4)

Hardware Environment

The nature of the architecture proposed for the processing utility lends itself to higher performance where multiple machines are concerned. To remove resource contention issues within the overall architecture, it is better to have multiple single CPU machines than a single multiple CPU machine. The hardware used was 4 instances of the set of machines shown in *Table 4*.

To scale-out processing during the benchmark we simply deployed multiple copies of this basic hardware environment.

In addition to the hardware listed in *Table 4*, a further 8 single CPU machines hosted the message drivers, monitor and message sink components used to record the benchmark results.

The database was put on Compaq MSA 1000 RAID (Redundant Array of Inexpensive Disks) storage device configured for level 0 support (maximum throughput, minimum recoverability). Since the business scenario for the trading exchange utility required local (i.e. client-side) database support for each participant connected to the utility, disaster recovery sites, and on-line dual system redundancy, the loss of RAID recoverability was considered a small price to pay compared to the gain in performance provided by a RAID level 0 configuration.

We initially thought a single database storage device could manage the full trade transaction volumes. But it soon became apparent during execution of the benchmark tests that internal disk queuing reached excessive levels. We'll see later how this problem was avoided.

Software Environment

The software operating environment used for the various hardware platforms and support software components is listed in *Table 5*. Although the RTM release of Windows™ Server 2003 had not occurred at the time of the benchmark, the release candidate (RC1) was available and was considered to be sufficiently stable and complete to be a valid part of the benchmark. All applicable service packs were applied to the operating environment including any third party drivers used for peripheral devices.

Scaling the Architecture

For the benchmark tests the scale-up model involved executing the software components on 4 CPU machines (an increase of 2 over the basic processing node) and also executing the code on the 64-bit Itanium processors. Although the Itanium machines contained only 2 processors, the available bus and I/O bandwidth was considerably higher than the standard 32-bit platforms and the results obtained

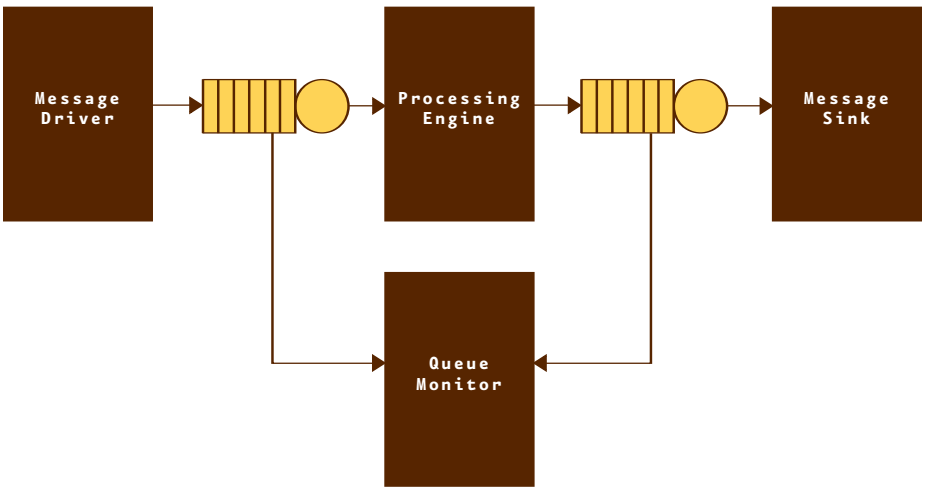


Figure 4: Benchmark Monitoring Arrangement

were certainly encouraging. We did not have time to investigate the scale-up model thoroughly.

The scale-out model employed was to increase the number of processing nodes from 1 through 8 processing engines and from 1 through 4 database processors. We spent the bulk of our time investigating the scale-out model.

Software Processing

The STE engine components were written in C/C++. They process XML messages using an exception based heuristic in which messages are assumed to be correct for both content and format until such time as an error is detected (missing or incorrect data). On exception, the message in question is directed to a nominated exception queue for further processing. The subsequent exception resolution processes were not included in the scope of the benchmark tests.

Validation of the content of each message was carried out against static data which had been pre-loaded into memory-based arrays. The validation process involved organising the required static data into sorted arrays, using C/C++ *qsort* function, and validating the existence of XML elements by locating an entry within the array, using C/C++ *bsearch* function. Data elements in the XML messages are accessed using the standard C/C++ *strstr* function.

Benchmark Results

The benchmark tests produced some interesting results. Some validated the application design, while others led to architectural changes to address identified performance issues. The basic lesson we learned was that you generally need to "tune for percentages and re-architect for orders of magnitude improvements in performance."

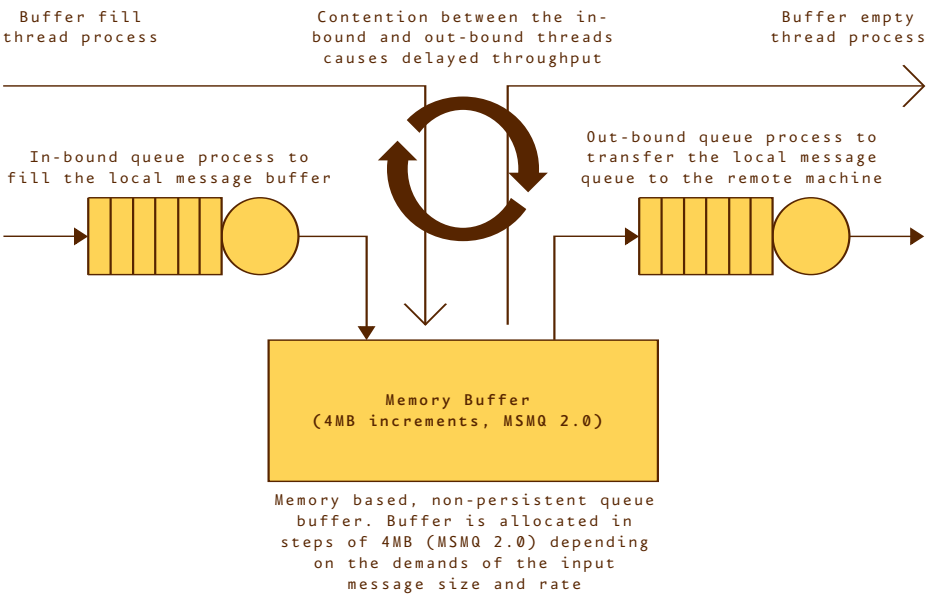


Figure 5: Single Queue Manager Contention Process

Message Queue Processing

The benchmark test was conducted at two basic levels, the first having 250,000 trades (1.75 million messages) and the second having 1,000,000 trades (7 million messages). All processed information was left in the database as an added processing load on the system. The arrangement shown in *Figure 4* was used as the basis for the benchmark evaluation.

A significant latency time was noted when the message driver components were first started due mainly to contention issues within the individual queue managers processing the message streams. The insert process rate was so high that the individual queue manager processes had insufficient time during the start of each run to transfer messages across the network to the remote processing machines. This gave the effect of the initial transfers taking several seconds before the performance monitors picked up any activity on the processing nodes of the system. *Figure 5* explains this queue manager contention process. In practise it is extremely unlikely that several million messages would arrive as a single input into the utility and therefore the latency effect could be ignored for the purposes of the benchmark evaluation. This effect would be common to all asynchronous message transfer systems, not just MSMQ which was used for the benchmark.

A similar effect could be observed in the processing components when large numbers of messages would arrive in their in-bound queue (remote host message buffer) as a single input. This meant that the processing components would be pushed to approximately 100% loading during the initial start of a run and would stabilise to a lower

figure once flow through the machines had evened out. A typical performance monitor output for a single processing node is shown in *Figure 6*. The effect of the initial message burst can be clearly seen in the response curve for the combined dual processor machine.

Furthermore, the effect of processing completion of the in-bound messages flowing into the queue manager message buffer can be seen in the later part of the response graph for the host machine. Here, increasing resources (or more accurately reducing contention) would allow an increased use of processor power in the latter stages of the processing cycle.

To counter both these effects of hitting message queue buffer limits too quickly and uneven processor utilization during the message injection phase, we added a larger number of in-bound queue processes and in-bound queues. This enabled the in-bound message load to be aggregated over more resources thus reducing the latency times involved. Perhaps the biggest difference was made by using MSMQ 3.0 instead of MSMQ 2.0. The former has a 4GB memory buffer size limit before a new buffer allocation is made, which is three orders of magnitude above the 4MB buffer size in MSMQ 2.0[3].

Multi-node Processing

In the original designs of STE a single database was used. As the number of processing nodes (dual processor machines running the component software) increased, a distinct drop in the overall processing rate was noticed. This drop in processing throughput is shown in *Figure 7* and was caused by contention within the database component.

The cause of the contention was not due to operating system or component software problems but due to excessive disk queuing occurring in the RAID array. This meant that available bandwidth for transferring information onto the disks of the RAID array was insufficient to meet the demands made by STE's software elements. This effect is most easily seen when examining the insertion rate of information into the database. The performance graph for the single database server is shown in *Figure 8*. Here, the corresponding performance graph to the one shown in *Figure 7* shows the dramatic reduction in inserts into the database cause by disk queuing in the RAID array as the number of processing nodes increases. The contention for available resources caused by this queuing means that the system couldn't reasonably cope with more than 2 processing nodes in the original design.

The next section discusses how this issue was overcome.

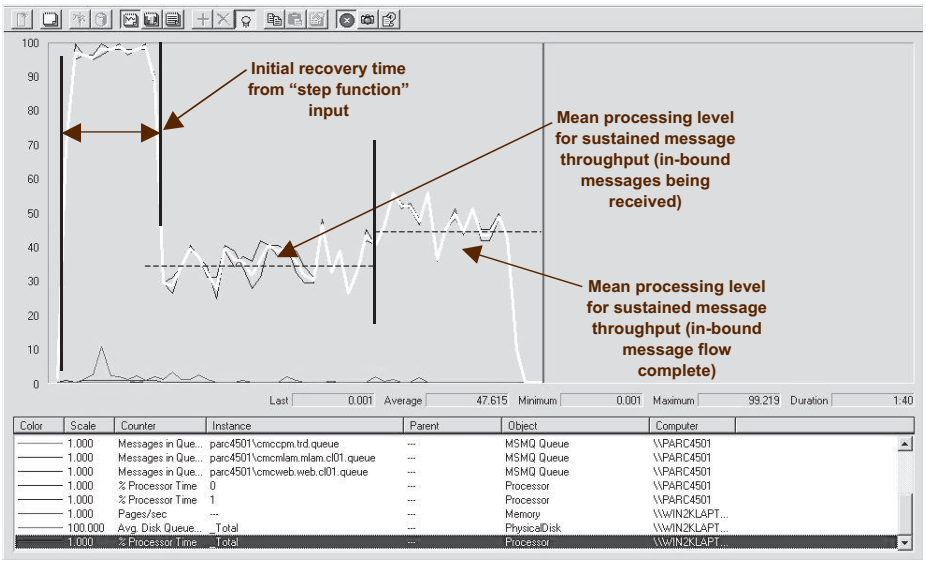


Figure 6: Processor Time for a Typical Component

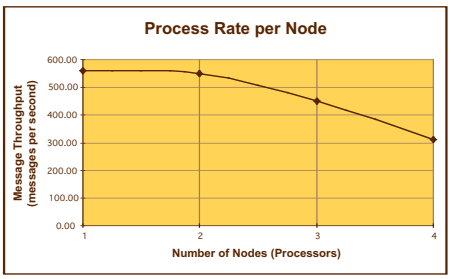


Figure 7: Process Rate per Node

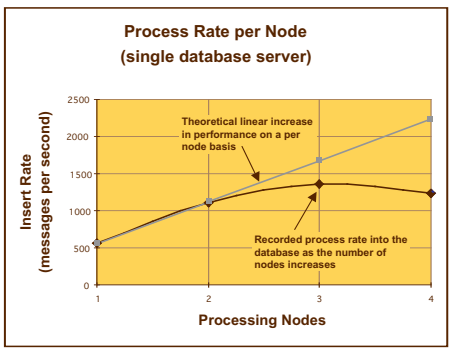


Figure 8: Process Rate per Node (Single Database Engine)

[3] The individual message size is limited to 4MB in both MSMQ 3.0 and MSMQ 2.0, but this was not an issue in our scenario.

Server Hashing Algorithm

After discussions with the Microsoft SQL Server team regarding techniques for improving the available bandwidth for disk operations the use of a hashing algorithm was proposed which would enable multiple database serves to be incorporated into the overall solution. The purpose of the hashing algorithm is to use a unique key on the trade derived from the message data, and a corresponding hash function which produces a single numeric value resolving to a unique instance of a database server. A hashing algorithm was chosen to reflect our business need to always direct constituent messages of specific trades to the same database server. For the benchmark the key consisted of a subset of the matching criteria defined in *Table 2*. The key was constructed by concatenating a chosen subset of matching criteria values and converted to a single, very long integer number. This number was then divided by a binary value representing the number of proposed database servers (or instances) as shown in the following formula:

$$S = \text{remainder}\left(\left\lfloor \frac{K}{2^N} \right\rfloor\right)$$

Using this formula[4] we very effectively improved performance by federating multiple database servers (or instances) with a hash.

The infrastructure architecture for the proposed utility was revised to reflect the inclusion of the multiple database solution using the hashing algorithm as shown in *Figure 9*. For ongoing benchmark testing support for up to 16 database servers was made within the software components, however, the system was tested to a maximum of 4 such servers.

Repeating the tests using the hashing algorithm to distribute the database load across four database servers yielded very impressive results.

The Little's Law Curve

The performance metrics determined by Buzen and Denning are based on a fundamental result produced by J.D.C. Little [LITTL61]. The generalised performance characteristic discovered by Little is shown in *Figure 10*.

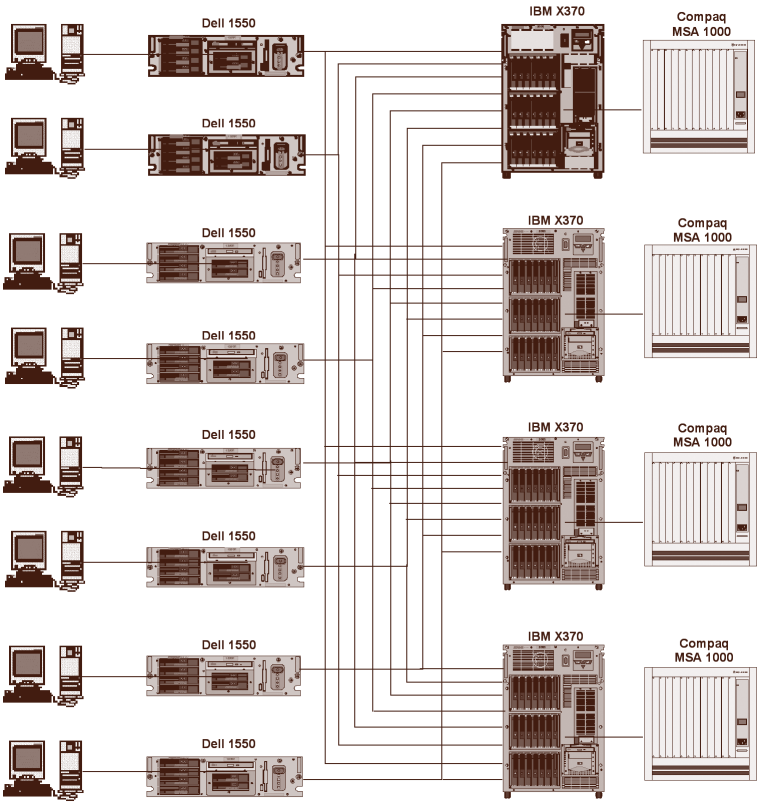


Figure 9: Final Hardware Infrastructure Arrangement

[4] S denotes the server number (base 0) in the range 0-(N-1); K denotes the very large integer value determined by the key for the message and N denotes the proposed number of database servers (clearly N must be greater than 0 and less than K). The calculation of the modulus function may seem complex, however, by choosing a binary multiple for the divisor the calculation reduces to right shifting the value of K by N bit position for N > 0. For example if N were set to 3 then the divisor would be 8 and shifting the value of K right by 3 bit positions would accomplish the required division. The remainder on division by 8 is therefore simply the value of the least significant 3 bits of K (an integer in the range 0 through 7).

"The impressive results of the benchmark largely speak for themselves with the overall performance, scalability and flexibility well established."

For any queuing model, when processing a specific number of tasks the response time increases as the arrival rate (and for a balanced queuing model the completion rate) increases. The curve is characterised by having an almost linear portion in the early stages getting progressively more asymptotic as the input (and completion rate) increases.

The first observable results, *Figure 11*, showed that the increase of 400% in the available database bandwidth placed the system in the linear portion of the of the performance graph producing an almost linear response characteristic when the processing components are scaled from 1 to 4 nodes.

In fact the measured results showed an extremely linear scaling between 1 and 4 processing nodes with only a minimal divergence from the linear model being observable. However, if the input rate is increased (in this case by increasing the number of processing nodes to 8) a divergence from the linear scaling case can be observed. This measured effect is shown in *Figure 12*.

Using measured results the Little's Law curve can be drawn for the test queuing network model as shown in *Figure 13*. The result shows that the operational performance of the queuing model will suffer increasing reduction in performance as the number of processing nodes increases past 4 components with a significant reduction at 8 processing components. At this point it is worth noting the scale on the left hand side of the graph in *Figure 12* showing the throughput rate of the entire STE queuing model measured at some 7734 messages per second.

Clearly the next scaling option to use would be to increase the number of database servers to 8 (i.e., the next available binary multiple). With this we would reasonably expect to see the message processing throughput rate reach in excess of 15,000 messages per second given a suitable increase in the number of processing nodes used.

Degradation with Stored Volume

At the measured processing rate the queuing network would achieve a sustained rate in excess of 27,842,400 messages per hour or 222,739,200 per operating day. It is reasonable to ask if such a high processing rate measured over several minutes could be sustained over time. To determine the characteristics of the model as stored volumes increase, the processing load of some 2,000,000 messages was used as a base figure and a subsequent processing run of 7,000,000 messages was used to determine the effect on the overall performance as the database volumes increased. The measured degradation in message throughput with volume is shown in *Figure 14*.

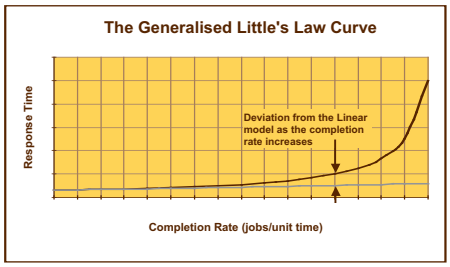


Figure 10: Generalised Little's Law Curve

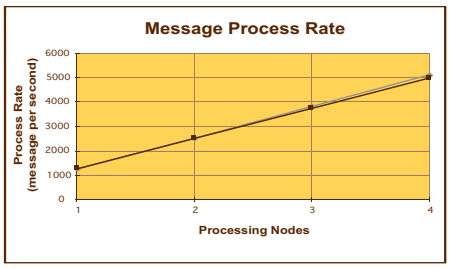


Figure 11: Linear Scaling with Multiple Database Instances (Single Database Engine)

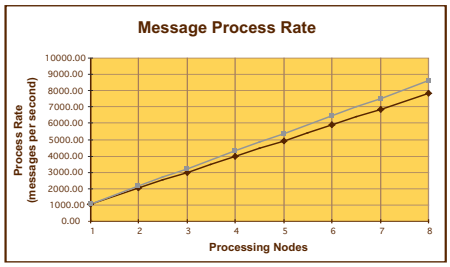


Figure 12: Scalability with Eight Processing Nodes

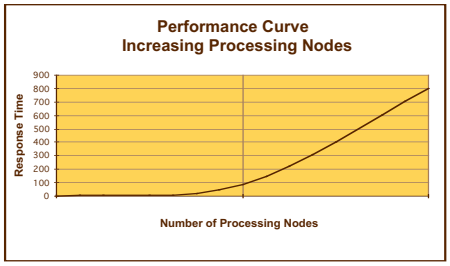


Figure 13: The Measured Little's Law Curve

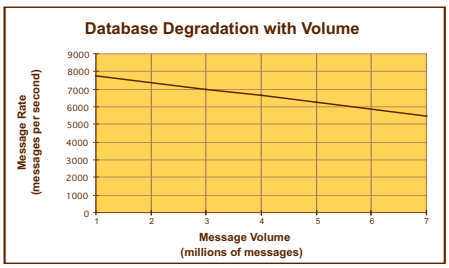


Figure 14: Database Degradation with Volume

Here the processing rate dropped to some 5,500 messages per second as the volume processed reached the 7,000,000 messages target. Even at this extreme level, the queuing model was achieving some 19,800,000 message per hour or 158,400,000 per operating day. The granularity of the result did not permit a more accurate measurement of the degradation effect other than the linear approximation shown here. If a finer granularity observation had been made the degradation rate would have been seen as a curve rather than a straight line indicating that a reduced degradation effect would be experienced as volumes increased further (possibly a characteristic of the paging schemas used for the B-Tree structure of modern RDBMS's). The 7,000,000 messages processed during this test represented 1,000,000 trades processed in a relatively short period of time.

It is worth noting that there are many examples of existing transaction engines in the financial services industry that have failed to reach this operating level using technology rated to a higher performance level than the Windows and Intel based machines used in these tests.

The Buzen & Denning Results

To determine the performance of the individual components the Buzen and Denning metrics need to be determined. The monitoring process measured the length of each of the processing queues used for the queuing network model and the performance metrics were calculated. A sample result from the calculation process is shown in *Table 6*.

The sample shows the processing of approximately 398,573 messages through the queuing model (taken as a sample from one of two processing nodes). The host machine supporting the 7 software components (one NOE and two each of ALC, NPR and SET software modules) reached an average 93% utilisation during the monitoring interval (U) according to the Windows performance monitoring tools. For the throughput calculation it must be borne in mind that there were two processing components running for each of the ALC, NPR and SET message types. The processing network therefore achieved a mean throughput rate of approximately 1,162 messages per time unit during the test with a latency time of approximately 14.97 seconds. Latency, in this case refers to the time difference between input and output message processing. A message entering the network will appear on the output side approximately 14.97 seconds later at the measured

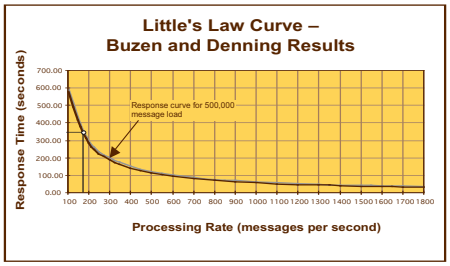


Figure 15: Measured Performance Curve

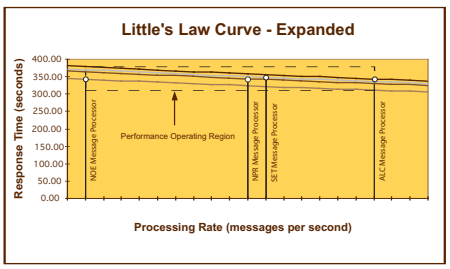


Figure 16: Expanded Performance Curve

Parameter	NOE	ALC	NPR	SET
Number of Processors	1	2	2	2
Observation Time - T units	343	343	343	343
Arrivals – N	53476	59040.5	56659	56849
Completions – C	53476	59040.5	56659	56849
Busy Time – B	315	329	308	322
Utilisation – U	0.92	0.96	0.90	0.94
Throughput – X	155.9067	172.1297	165.1866	165.7405
Mean Service Time - S	0.00589	0.005572	0.005436	0.005664
Execution Distribution - A	817,385	887,046.5	818,094	860,110
Mean Queue Length - L	2383.047	2586.141	2385.114	2507.609
Residence Time - RT	15.28508	15.02437	14.43891	15.12973
Queuing Time	15.27919	15.0188	14.43347	15.12407

Table 6: Sample Buzen and Denning Results Calculation

processing levels. The sample rate was set to a 7 second interval (owing to the use of an un-calibrated tick interval) and the sample data set was 500,000 messages (two messages drivers each with 250,000 messages from the standard test data set). In this case there were two process engines (or host machines) and the results shown were taken from one of those processing engines (note that the results for the ALC, NPR and SET components are aggregated across two components since the original data was measured from the message queue from which both components were being fed).

The marginally higher service time (S) for the NOE messages reflects the higher processing level required of this component because of audit trail persistence and validation processing. The performance curve for each of the processing components is shown in *Figure 15* with the overall performance point marked for clarity. This is fundamentally the generalised Little's Law curve for the STE processing engine; however, this generalised view does not give all of the detail necessary to accurately predict the operating performance of the engine. It is apparent that, for the individual components there are different completion rates and therefore depending on which view of the performance metrics you wish to take there will be a different corresponding performance values.

This gives rise to the Performance Operating Region (POR) for the network as shown in the shaded area of the graph in *Figure 16*. In this particular instance the results are reasonably close and therefore the corresponding performance operating region is narrow. This is not always the case, however, and there are examples of systems where the POR covers a region exceeding 400% of the mean performance level. The prediction of the POR requires some complicated mathematics that were beyond the scope of this benchmark exercise, however, the effect of the POR is included here to explain the variation in the measured results during repetitive tests.

Comments and Conclusions

The impressive results of the benchmark largely speak for themselves with the overall performance, scalability and flexibility well established. The throughput rate for the overall engine certainly places it amongst some of the largest transaction engines for post trade processing infrastructure in the financial services industry. The target and stretch performance levels were exceeded with comfortable margins, and there are strong indications that the overall architectural approach will support even greater message throughputs. It is certainly worth stating that, with the achieved performance levels, current Microsoft technology offerings are capable of operating within the enterprise layer of any financial institution. Some aspects of operation within the enterprise layer, like resilience and reliability were not tested during this benchmark and remain to be proven for this design. However, networked or grid computing based architectures like this one have inherent characteristics to support extremely high levels of resilience and reliability. The use, therefore, of efficient grid based processing machines and low cost software technology would seem to be a winning combination.

Low Cost and Efficient Realisation

Probably one of the more significant results of the entire benchmark process is the now proven ability of Microsoft technology to perform and scale to enterprise levels. The processing rates achieved with the queuing architecture certainly place the Microsoft operating system in the upper band for capability and scalability in networked computing. The second most significant result of the testing was the relatively low cost of implementation of the system in the Microsoft environment.

Potential Improvement Areas

In addition to the monitoring tools used to detail the benchmark results, Microsoft was able to provide additional process monitoring tools (which will be available in Visual Studio 2005) that gave a detailed view of the execution of the software elements. The Microsoft analysis tools indicated that the software components were spending on average 30%-35% of their time performing functions related to data extraction from the XML messages. This was not an overly surprising result since the main function of the software components was to validate and process string types. To access required information the C/C++ *strstr* search function was used and we treated the entire message as one complex string. (Note: for our problem domain this was faster than directly using an XML parser and DOM objects with XSLT.) Although in general circumstances the use of *strstr* produces adequate performance levels, there are more efficient techniques that can be employed to extract information from string-based message structures.

R.S. Boyer and J.S. Moore [BYMR77] described a very efficient mechanism

for searching well structured strings. The algorithm works well where the structure of the string is known in advance and is used predominantly in applications where searching large text strings is required, such as in digital library applications, editors or word processors. For the queuing network the use of the algorithm would at first sight seem inappropriate since we have no way of determining the nature or structure of the next received message within the network. However, for the processing components the structure of the message is known since we route the message according to type for further processing. The use of the Boyer-Moore algorithm could yield improved results over the existing implementation of the network, however, the relatively small size of the XML messages (an average of 1,500 bytes per message) might be too small for the Boyer-Moore algorithm to yield results that would justify the work required to implement the algorithm.

Itanium and 64-Bit Processing

The operation of the queuing network model was tested using a beta version of SQL Server running on new (at least it was then) Itanium 64-bit hardware and the Windows operating system. Although this was not acceptable as a production benchmarking environment (because of using beta software) the results would be a useful indicator of future performance gains that could be obtained using this future Intel/ Microsoft technology.

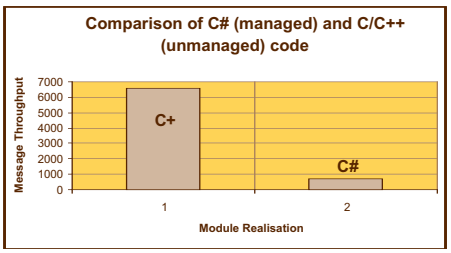


Figure 17: Comparison of Managed and Unmanaged Code

On this hardware the measured throughput was an average of 872 messages per second which was considered extremely high considering the environment in which the test took place. Firstly, this result was obtained using a standard SCSI disk unit as opposed to the RAID arrays used in the main benchmark exercise. Standard SCSI performance rates would have been considerably slower than the RAID performance rates. Secondly, the Itanium database server had only two processors installed against the 8 processors used for the database engines in the benchmark. The opportunity to perform a full benchmark test within a 64-bit environment is eagerly awaited.

C# and Managed Code

The software components were also generated for use in the Microsoft C# Managed Code environment where a direct comparison could be made between the C/C++ and C# versions. As a simple test the operation of the message drivers was compared between the operating models. The process involved was fairly simple so that the effects of inefficient coding could be ignored (the actual number of active lines of code was very small). The process was to take a prepared message file and stream the data into a code loop. Processing would continue until a message separation character was received. The resultant message was then wrapped in a standard GSTPA header and a digital signature applied to the message block. The message was then written to the message queue for the queuing network model for processing. This process continued until the entire prepared file was read.

The parameter of interest to us was the throughput rate at which message could be read off the data file and queued. The results of the test are shown in *Figure 17*.

[These results clearly show the enhanced performance of C/C++ over the managed code environment (.NET Framework 1.1). It is also fair to point out that the results also include the comparison of efficiency of the interoperability layer between C/C++ and C# which is crossed for accessing MSMQ.

At first sight it could be argued that from a performance perspective, the managed code environment should never be implemented in place of a C/C++ installation. This view, however, would be misleading since all system solutions are a compromise between cost, performance and reliability. The overall performance results for the managed code environment reflected the test performed on the (simple) message driver component producing a throughput rate of approximately 2000 messages per second. Although this rate is around 25% of the base C/C++ level, there are definitely compensating factors that must be considered. The production of C# code is significantly more efficient that that obtainable with C/C++ code, in fact the rate at which operational code could be developed in C# was extremely impressive. Note, a lower rate of 2000 messages per second (which is 7,200,000 messages per hour or 57,600,000 messages per day) is still considered in the upper bracket of transaction engine benchmarks and it will only get better as managed code gets faster!

Furthermore, care must be taken when comparing managed and unmanaged code environments. The use of virtual machine environments like the Common Language Runtime (CLR) used in the .NET Framework and indeed Java/ J2EE VM based environments can produce benchmark figures comparable with that of C/C++ code where memory-based operation are concerned. Unfortunately, such benchmark comparisons give a false picture of the overall performance levels that can be expected because most applications include elements of local and remote I/O operations and dynamic object creation and deletion. Conversely, the ease of implementation and potential improvements in reliability and manageability may well allow managed code environment to out-perform C/C++ in the creation of applications sacrificing performance for lower costs and faster implementation times.

Given that we have significant unrealisable performance (see the above discussion on this topic) with our unmanaged code implementation of the STE system – because the architecture scaled-up so remarkably well, we actually find ourselves in an opportune position to trade-off this unrealisable performance for the benefits of using managed code in future implementations of this generalised architecture. For some insights into this statement see [SEVA04].

References

[BDEN78] Buzen J.P. and Denning P.J., "The Operational Analysis of Queuing Network Models", ACM Computing surveys, 10, 3, Sept. 1978, 225-261.

[BYMR77] Boyer R.S, Moore J.S, 1977, A fast string searching algorithm. *Communications of the ACM*. 20:762-772.

[LITTL61] Little J.D.C, "A Proof of the Queuing Formula $L = \lambda W$", Operations Research, 9, 1961.

[PHEL02] Pat Helland, 2002, "Autonomous computing: Fiefdoms and Emissaries", Microsoft Webcast, http://microsoft.com/usa/Webcasts/ondemand/892.asp

[SMITH90] Connie U. Smith, "Performance Engineering of Software Systems", Addison-Wesley, ISBN 0-201-53769-9, 1990.

[SEVA04] Arvindra Sehmi and Clemens Vasters, FABRIQ GotDotNet Workspace: http://workspaces.gotdotnet.com/fabriq

[SMWIL02] Connie U. Smith and Lloyd Williams, "Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software", Addison-Wesley, ISBN 0-201-72229-1, 2002.

[UASDS01] Susan D. Urban, Akash Saxena, Suzanne W. Dietrich, and Amy Sundermier, "An Evaluation of Distributed Computing Options for a Rule-Based Approach to Black-Box Software Component Integration", Proceedings of the Third Int'l Workshop Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS'01), 2001.

[ZHES01] Huaxin Zhang and Eleni Stroulia, "Babel: Representing Business Rules in XML for Application Integration", Proceedings of the 23rd International Conference on Software Engineering (ICSE'01), 2001.

Richard Drayton
FiS Group
richard.drayton@btinternet.com

Richard Drayton has been actively involved in technology architecture and design for 30 years. Recognized for his work in the field of software performance engineering, he is a practicing member of the ACM's SIGMETRICS group. For the past 15 years Richard has been active in the design and development of Financial Trading and Processing systems for many of the worlds leading investment banks. Previously Richard was the head of Front Office Technology for Deutsche Bank in Frankfurt, Global Head of Architecture for Commerz Financial Products and Global Head of Architecture for Dresdner Bank. As part of his work with the IEEE and ACM, he has focused on the adaptation of high performance computing solutions to the financial services industry with particular focus on the implementation of end to end STP market infrastructure solutions. He is the Executive Director responsible for technology solutions with the Financial Infrastructure Solutions Group. He holds an M.Sc. in Electronic System Design as well as degrees in Pure Mathematics and Communications Technology.

Arvindra Sehmi
Microsoft EMEA
asehmi@microsoft.com

Arvindra Sehmi is an Architect in Microsoft EMEA Developer and Platform Evangelism Group. He focuses on enterprise software-engineering best practice adoption throughout the EMEA developer and architect community and leads Architecture Evangelism in EMEA for the Financial Services Industry where his current interest is in high performance asynchronous computing architectures. Arvindra is the executive editor of JOURNAL. He holds a Ph.D. in Bio-medical Engineering and a Masters degree in Business.

Enterprise Architecture Alignment Heuristics

By Pedro Sousa, Carla Marques Pereira and José Alves Marques, Link Consulting, SA

Introduction

The alignment between Business Processes (BP) and Information Technologies (IT) is a major issue in most organizations, as it directly impacts on the organization's agility and flexibility to change according to business needs. The concepts upon which alignment is perceived are addressed in what is called today the "Enterprise Architecture", gathering business and IT together.

Many Enterprise Architecture Frameworks have been proposed, focusing on different concerns and with different approaches for guiding the development of an IT infra-structure well suited for the organization. Each Enterprise Architecture Framework has its own concepts, components, and methodologies to derive the component all the required artifact. However, when the main concern is alignment, we may consider simpler architecture concepts and simpler methodologies because the focus is not to define development artifacts but only to check their consistency.

The focus of this paper is to show how alignment between Business and IT can be stated in terms of the components found in most Enterprise Architectures.

In the next section, we briefly introduce three well known Enterprise Architecture Frameworks, namely: The Zachman Framework, Capgemini's Integrated Architecture Framework and the Microsoft Enterprise Architecture. We do not intend to fully describe them, but solely present the main aspects.

Next, we present the basic concepts common to these frameworks, focusing on their generic properties and leaving out specificities of each framework. We will consider four basic components of an Enterprise Architecture: Business Architecture, Information Architecture, Application Architecture and Technical Architecture.

Finally, we show how alignment between Business and IT can be disaggregated into alignment between these basic components, we present general heuristics defined in terms of the architectural components and present the work in progress. We will not address the Technical Architecture; the main reason being that technical alignment is mostly dependent on the technology itself.

Enterprise Architecture Frameworks

The Zachman Framework

The Zachman Framework for Enterprise Architecture (www.zifa.com) proposes a logical structure for classifying and organizing the descriptive representations of an enterprise. It considers six dimensions, which can be analyzed in different

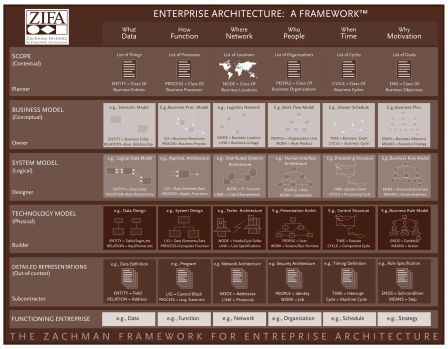


Figure 1 – The Zachman Framework, © John A. Zachman International

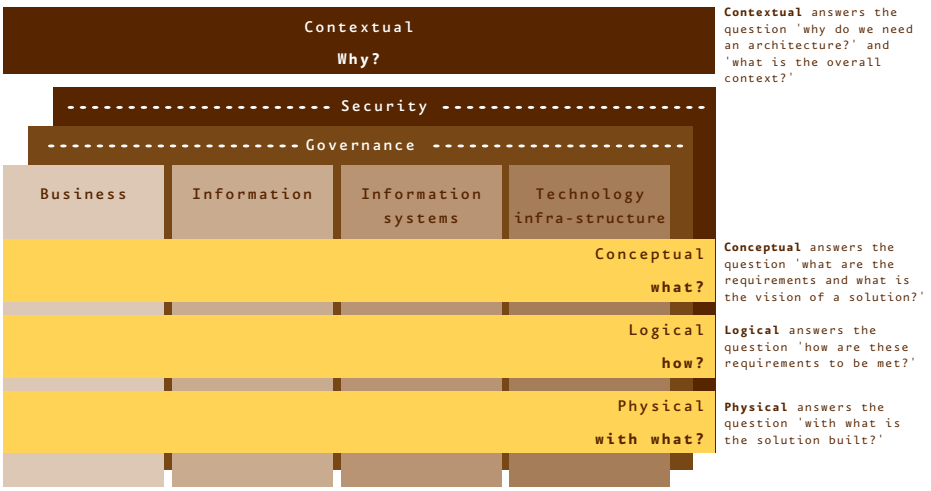
perspectives, as presented in *Figure 1*; the rows represent the perspectives and the columns the dimensions within a perspective.

The Framework is structured around the views of different users involved in planning, designing, building and maintaining an enterprise's Information Systems:

– Scope (Planner's Perspective) – The planner is concerned with the strategic aspects of the organization, thus addressing the context of its environment and scope.

– Enterprise Model (Owner's Perspective) – The owner is

Figure 2 – The Integrated Architecture Framework

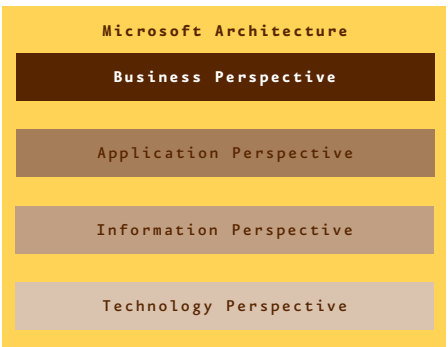


"We developed Alignment's Heuristics as a common sense rule (or a set of rules) to increase the probability of finding an easier way to achieve Business, Information and Application Architectures alignment."

interested in the business perspective of the organization, how it delivers and how it will be used.
– System Model (Designer's Perspective) – The designer is concerned with the systems of the organization to ensure that they will, in fact, fulfill the owner's expectations.
– Technology Model (Builder's Perspective) – The builder is concerned with the technology used to support the systems and the business in the organization.
– Detailed Representations (Subcontractor's Perspective) – This perspective addresses the builder's specifications of system components to be subcontracted to third parties.

While the rows describe the organization users' views, the columns allow focusing on each dimension:
– Data (What?) – Each of the cells in this column addresses the information of the organization. Each of the above perspectives should have an understanding of enterprise's data and how it is used.
– Function (How?) – The cells in the function column describe the process of translating the mission of the organization into the business and into successively more detailed definitions of its operations.

Figure 3 – Microsoft Enterprise Architecture Perspectives



– Network (Where?) – This column is concerned with the geographical distribution of the organization's activities and artifacts, and how they relate with each perspective of the organization.
– People (Who?) – This column describes who is related with each artifact of the organization, namely Business processes, information and IT. At higher level cells, the "who" refers to organizational units, whereas in lower cells it refers to system users and usernames.
– Time (When?) – This column describes how each artifact of the organizations relates to a timeline, in each perspective.
– Motivation (Why?) – This column is concerned with the translation of goals in each row into actions and objectives in lower rows.

Capgemini's Integrated Architecture Framework

Capgemini has developed an approach to the analysis and development of enterprise and project-level architectures know as the Integrated Architecture Framework (IAF) shown in *Figure 2* [AM04].

IAF breaks down the overall problem into a number of the related areas covering Business (people and processes), Information (including knowledge), Information Systems, and Technology Infrastructure, with two special areas addressing the Governance and Security aspects across all of these. Analysis of each of these areas is structured into four levels of abstraction: Contextual, Conceptual, Logical and Physical.

The Contextual view presents the overall justification for the organization and describes the contextual

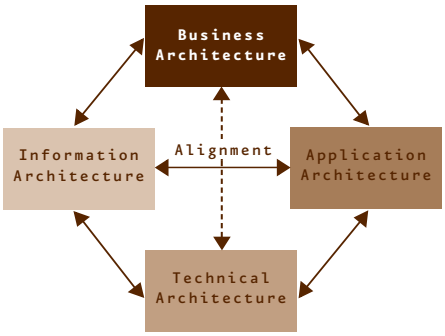


Figure 4 – Decomposing Business and IT Alignment into Architectural Components

environment. It corresponds largely to Zachman's Planner's Perspective row.

The Conceptual view describes what the requirements are and what the vision for the solution is. The Logical view describes how these requirements and vision are met. Finally, the Physical view describes the artifacts of the solution.

These views have no direct relation to Zachman's perspectives because in IAF, Business, Information, Information Systems and Technology Infrastructure are the artifacts of the architecture whereas in Zachman, Business, Information Systems and Technology are views (perspectives).

Microsoft Enterprise Architecture

Microsoft Enterprise Architecture shown in *Figure 3*, is a two dimensional framework, that considers four basic perspectives (business, application, information, and technology), and four different levels of detail (conceptual, logical and physical and implementation).

The business perspective describes how a business works. It includes broad business strategies along with plans for moving the organization from its current state to an envisaged future state.

"The issue of alignment is based on the coherency between elements of Business Architecture, elements of Information Architecture and elements of Application Architecture."

The application perspective defines the enterprise's application portfolio and is application-centered. The application perspective may represent cross-organization services, information, and functionality, linking users of different skills and job functions in order to achieve common business objectives.

The information perspective describes what the organization needs to know to run its business processes and operations. The information perspective also describes how data is bound into the business processes, including structured data stores such as databases, and unstructured data stores such as documents, spreadsheets, and presentations that exist throughout the organization.

The technology perspective provides a logical, vendor-independent description of infrastructure and system components that are necessary to support the application and information perspectives. It defines the set of technology standards and services needed to execute the business mission.

Each of these perspectives has a conceptual view, a logical view and a physical view. Elements of the physical view also have an implementation view. Microsoft Enterprise Architecture is described in detail at msdn.microsoft.com/architecture/enterprise/

Making a parallel between the Microsoft's and Zachman's Enterprise Framework, the Business perspective corresponds to *Planner's* and *Owner's* perspectives; Application perspective to Zachman *Designer's* perspective; Technology perspective to *Builder's* and Subcontractor's perspectives and finally, the Microsoft Information perspective corresponds to the Data column in the Zachman framework.

Identifying Enterprise Architecture Components from an Alignment perspective

As we could see in previous sections, different Enterprise Frameworks have different ways to model artifacts of the Enterprise, their perspectives and the different levels at which they can be described.

The Enterprise Frameworks address a large number of problems and therefore have a degree of complexity far larger than needed if the sole problem is the alignment of the business and IT architecture. Thus we can simplify these models and just consider the sub-architectures which have a certain commonality. It is out of the scope of this paper to fully study and justify similar concepts in these Enterprise Architecture Frameworks. If alignment is the main concern, an Enterprise Architecture has four fundamentals components: Business Architecture, Information Architecture, Application Architecture and Technical Architecture. This is not new, and it has been long accepted in the Enterprise Architect Community (for instance see www.eacommunity.com).

We will address the issue of alignment based on the coherency between elements of Business Architecture, elements of Information Architecture and elements of Application Architecture. The more elements each of these Architectures has, the more rich and complex is the concept of alignment, because more rules and heuristics need to be stated to govern the relation between these elements. So, in order to build up alignment, one must first clarify the elements of each architecture (see *Figure 4*).

In what concerns the Technical Architecture, its alignment is mostly dependent on the technology itself. We are currently investigating how Service Oriented Architecture (SOA) concepts overlap with previous architectures and how alignment could be formulated in its model. This is ongoing work and is beyond the scope of this paper.

Business Architecture

The Business Architecture is the result of defining business strategies, processes, and functional requirements. It is the base for identifying the requirements for the information systems that support business activities. It typically includes the following: the enterprise's high-level objectives and goals; the business processes carried out by the enterprise as a whole, or at least a significant part; the business functions performed; major organizational structures; and the relationships between these elements.

In this paper, we consider a simpler case where the Business Architecture includes only Business Processes, each business process is composed by a flow of business activities and each activity is associated with information entities, time, and people. Business Processes have attributes such as criticalness, security level, delayed execution (batch), on-line, and so on.

Information Architecture

The Information Architecture describes what the organization needs to know to run its processes and operations, as described in the Business Architecture. It provides a view of the business

"Applications that manage information entities should provide means to make the entity information distributable across the organization using agreed-on protocols and formats."

information independent of the IT view of databases. In the Information Architecture, business information is structured in *Information Entities*, each having a business responsible for its management and performing operations like: Acquisition, Classification, Quality Control, Presentation, Distribution, Assessment, and so on.

Information Entities must have an identifier, defined from a business perspective, a description, and a set of attributes. Attributes are related to business processes (that use or produce them) and to applications (that create, read, update and delete them). Attributes are classified according to different properties such as Security, Availability and so on.

As an example, *Client* and *Employee* are typical Information Entities. *Employee* has attributes such as "courses taken", "competences", "labor accidents", and "career". Each of these attributes can be physically supported by a complex database schema in different databases used by several applications.

Application Architecture

The Application Architecture describes the applications required to fulfill two major goals: (i) support the business requirements, and (ii) allow efficient management of Information Entities. Application Architecture is normally derived from the analyses of both Business and Information Architectures.

Application Architecture typically include: descriptions of automated services that support the business processes; descriptions of the interaction and interdependencies (interfaces) of the organization's application systems; plans for developing new applications and revision of old applications based on the enterprises objectives, goals, and evolving technology platforms.

Applications also have required attributes, such as availability (up time), scalability (ability to scale up perform), profile based accesses (ability to identify who does each tasks).

Alignment and Architecture Components

After identifying the major architectural components from an alignment point of view, we are now in position to address the relations between these components in terms of alignment.

Alignment between Business and Applications

In a fully aligned Business and Applications scenario, the time and effort business people spent to run the business should be only devoted to "reasoning" functions. On the contrary, misalignments force business people to perform extra and mechanic work such as:

– Inserting the same data multiple times in different applications.
– Logging in multiple times, once for each application they need to access.
– Recovering from a failed operation across multiple systems, requiring careful human analyses to rollback to a coherent state.
– Overcoming inappropriate application functionality. For example, printing invoices one by one because applications do not have an interface for multiple printing.

Notice that alignment between Business and Applications in the above context, does not imply a flexible and agile IT Architecture, in fact, a measure of a flexible and agile IT Architecture is the effort IT people make to keep the Business and Applications aligned when Business is changing. This topic is addressed next.

Alignment between Information and Applications

In fully aligned Information and Applications Architectures, IT people only spent effort and time coding business functions and logics. On the contrary, misalignments between information and application require IT people to do extra coding for:
– Keeping multiple replicas of the same data coherent, because they are updated by multiple applications.
– Assuring coherency from multiple transactions, because a single business process crosses multiple applications.
– Gathering information from multiple systems and coding rules to produce a coherent view of the organization's business information.
– Transforming data structures when data migrates between applications.

The extra coding is required to consistently modify both architectures. However, since the information critical to run the business (Information Architecture) is far more stable than the applications to support it (Applications Architecture), most effort really is put into changing in the Applications.

Alignment between Business and Information

Information and Business Architectures are aligned when business people have the information they need to run the business. This

means accurate, with the right level of detail, and on time information. Unlike the previous misalignments, here the impact is neither time nor effort, but the impossibility of getting the adequate piece of information relevant for the business.

Examples are abundant; a CEO asks for some report, where sales figures need to be disaggregated by type of services. Assuming the report requested by the CEO has either actual or foreseen business relevance, the possibility/ impossibility to produce such report is an evidence of the alignment/misalignment between information and Business Architectures. To produce the report we must have the adequate basic data and data exploring applications, and thus this is an issue that should be dealt by the previous Alignments (Information/ Applications and Business/Applications).

Alignment Heuristics

We developed Alignment's Heuristics as a common sense rule (or a set of rules) to increase the probability of finding an easier way to achieve Business, Information and Application Architectures alignment.

Heuristics presented result from mapping our experience, both as academic teaching at university and professional consultancy services, into the context of the Business, Information and Application Architectures presented in this paper. We present the heuristics that we consider to have a greater value, given its simplicity and its results.

The main heuristics to consider when checking alignment between Business and Application Architectures are:

– Each business process should be supported by a minimum number of applications. This simplifies user interfaces among applications, reduces the need for application integration, and also minimizes the number of applications that must be modified when the business process changes.
– Business activities should be supported by a single application. This reduces the need for distributed transactions among applications.
– Critical business processes should be supported by scalable and highly available applications.
– Critical business processes/activities should be supported by different applications than the non critical business processes/activities. This helps to keep critical hardware and permanent maintenance teams as small as possible.
– Each application's functionality should support at least one business process activity. Otherwise, it plays no role in supporting the business.
– Information required for critical processes should be also supported by scalable and highly available systems.
– Business processes activities requiring on-line/batch support should be supported by applications running on different infra-structures, making easier the tuning of the systems for operating window.

The main heuristics to check alignment between Application and Information Architectures are:

– An information entity is managed by only one application. This means that entities are identified, created and reused by a single application.
– Information entities are created when identifiers are assigned to them, even if at that time no attributes are known. For example, if the Client information entity may be created before its name and address and other attributes are known. Even so, the application that manages Client information entity must be the application that manages their IDs.
– Applications that manage information entities should provide means to make the entity information distributable across the organization using agreed-on protocols and formats.
– Exporting and distributing information entities across organization applications should make use of a "data store", rather than a point to point Application integration. Applications managing a given information entity should export its contents to the data store when its contents have changed. Applications requiring a given information entity should inquire the data store for up-to-date information. This allows for computational independence between applications, and make possible to size the HW required to run an application without knowing the rate that other applications demand information from it. Further, if the application goes down, it allows other to continue operation using best possible data.

– Whenever possible, applications should manage information entities of the same security level. This simplifies the implementation of controls and procedures in accordance with the security policy.

Finally the main heuristics to apply for Business and Information alignment are:

– All business processes activity create, update and/or delete at least one information entity.
– All information entities attributes are read at least by one business process activity.
– All information entities have an identifier understood by business people.
– All information entities must have a mean of being transformed for presentation to appropriate audiences using enterprise-standard applications and tools.
– All information entities must derive from known sources, and must have a business people responsible for its coherency, accuracy, relevance and quality control.
– All information entities must be classified and named within the Information Architecture.
– For each information entity, Business people should be responsible for assessing the usefulness and cost/benefits of information and sustain its continued use.

Final Remarks

The heuristics presented have been validated and tested in real projects. In some cases, different heuristics produce opposite recommendations. This means that, a compromise solution must be reached. In other cases, heuristics do not favor optimal solutions from an engineering point of view, because optimal solutions do not normally take into account flexibility and ease of change.

Another remark is the heuristics presented intend to validate alignment among architectures, but assume that each architecture is "aligned within itself". This means that we are not checking if, for example, the Business Architecture presents a good and a coherent schema of the business processes and activities. Likewise, we are not checking if Applications Architecture makes sense or not. This requires more complex models, such as the ones initially proposed in original Frameworks (Zachman, IAF and Microsoft).

The work presented was developed using the Zachman Framework as the general approach, but is strongly focused in the alignment issues. We have coded a large percentage of the heuristics presented in a modeling tool (System Architect from Popkin Software) and we are able to derive a measurement for alignment given a Business, Application and Information Architecture. We have started work to include these heuristics in Microsoft Visio.

We consider the heuristics to be valuable because they force architects to think about the justification of their decisions, leaving better documented and solid architectures.

References

[AM04] Andrew Macaulay, Enterprise Architecture Design and the Integrated Architecture Framework, JOURNAL1: Microsoft Architects Journal, Issue 1, January 2004.

Pedro Sousa
pedro.sousa@link.pt
Pedro Sousa is responsible for Enterprise Architecture professional services at Link Consulting SA, where he has been involved in several Enterprise Architecture projects for the past six years. He is an Associate Professor at the Technical University of Lisbon (IST), where he teaches on theses subjects in Masters of Computer Science courses. He has published a series of papers about Business and IT alignment.

Carla Marques Pereira
Carla Marques Pereira has a MSc in Computer Science at the Technical University of Lisbon (IST). She is reading for a PhD on the subject of "Business and IT alignment". Carla is a member of Link Consulting SA's Enterprise Architects team and a researcher in the Center for Organizational Engineering (ceo.inesc.pt).

José Alves Marques
José Alves Marques is CEO of Link Consulting SA and a full Professor at Technical University of Lisbon (IST). Technically, his main focus is Distributed Systems Architectures and Services Oriented Architectures. He has a long track of published papers and projects on these domains.

Razorbills:
What and How of Service Consumption

By Maarten Mullender, Microsoft Corporation

Why did the razorbill raise her bill?
So that the sea urchin would see her chin!

Introduction
Almost exactly seven years ago, I read a Microsoft internal memo called *The Internet Applications Manifesto, Take II* or *The Sea Urchins Memo*. This paper pushed for using XML as the communication mechanism between services. To quote:

I call this model the sea urchin model because I think of sites already as sea urchins with many spikes sticking out of them (URLs) each dispensing UI (HTML). These URLs as most have already realized are the "methods" of the site. In this brave new world, I expect sites to also routinely become Information dispensers with the "methods" of the site returning not UI but information. Thus one can think of the Web as a sea filled with spiky sea urchins, all willing to emit information of various kinds if one but knew to ask. Initially, one will have to know to ask. The evolution of an infrastructure that can automatically find such information will depend upon the emergence of what I call "Schema", namely universally agreed upon types of semantics that allow multiple independent consumers and producers to agree on how to share information in specific domains. See opportunities below for more on this.

Since the *Sea Urchins* memo, we have made great progress in describing the *how* of services. With *"Razorbills"*, I propose putting a similar effort behind providing a description of the *'what'* of services. Describing the *'what'* will add dramatically to the usefulness of services and "raise her bill".

I propose standardizing the *'what'* description using Entities, Views, Actions, References and Relationships.

Service Oriented Architecture (SOA) is as much, if not more, about consuming services as it is about building services. To enable better consumption I believe we need infrastructure that allows people that may be less technical but do have domain expertise (the so called business analysts) to design and build solutions. We need infrastructure that allows users of any application, structured forms as well as free form documents, to really use the functionality offered by services. Standardization on a way of describing services in business terminology will help do that. It will help business analysts and users to better bridge the gap between the structured and the unstructured, the formal and informal, both in content and in process.

What and How
Current service descriptions such as WSDL and the WS-Standards describe how to access functionality. Once the caller has decided which functionality to invoke, the WSDL describes how to encode the message, how to route the message, how to secure the message, how to authenticate the caller and everything else needed to describe how to send the message. It does not describe what should be called. For any dedicated client or consumer of a service that already knows what to call, this is great and since the infrastructure and the description are based on standards, the code is reusable and many technologies can interoperate. This means that you can create communications between many parts, based on a single infrastructure.

However, this implies that the logic of what is being called is coded into the application. When organizations want to be flexible, they may want to access services to get information, aggregate information, and reorder information, or they may want to change process flows. Currently, to do this, they require developers. This may make the desired changes expensive – in many cases, too expensive. Organizations facing this problem would rather have business analysts, and even business users, make these changes. This would make change cheaper, shorten the change cycle, and make them more agile.

If our software could communicate this *what* to the business people, then they could define new aggregation of information, and change or create the flow of processes without the help of developers.

If we describe what services have to offer and we describe this in terms understood by both business people and software, we would not only have a description of the how, but also of the *'what'*.

This information is available today, but it is in the heads of the designers and developers – at worst – and in some design documents at best. The consuming applications are being written using this knowledge. However, it is generally not available to tools and other software that could be used by analysts and users to solve their challenges. We need to find a standard way of describing this information.

In the following I will explain the fundamental idea of such a description of what services expose and then I will highlight possible use of such

"With *'Razorbills'*, I propose putting a similar effort behind providing a description of the *what* of services."

description. I believe and I hope that I will be incomplete; going down this road will undoubtedly provide us with more insight and more ideas.

What
Services expose business functionality to their consumers. Business analysts and business users often prefer to think in terms of the entities they use. They understand the relationships between these entities and the actions that can be taken on those entities.

Entities
Most services encapsulate *entities*. However concrete these entities were for the designers of the service, for the consumer of the service they are abstract. The consumer cannot touch them or manipulate them; the consumer does not know whether they are stored in a single table or in a set of files. The consumer, however, does understand what they represent: Customer, Order, Problem report, etc. The entity is made up of the combination of all of its state, its behavior as well as its relations to other entities.

Views
The service exposes *views* on these entities. The consumer may obtain an address view, or a financial view, or a historical view on the customer. These views are snapshots of the entity, taken from a specific viewpoint. The views are concrete. They have a well-defined schema exposing attributes or properties of the entity. It must be noted however, that views do not necessarily present a subset of the entity information. They may very well contain information from related entities and overlap with views on those entities. The view on an order may contain the customer's name and address as well as the product descriptions so that it may be used in the user interface or to populate a document.

Actions
The service also exposes methods or *actions*. Many of the actions are related to an entity. For example, "release order" or "upgrade customer" are clearly related to specific entities. And, if these actions require input information, this information often comes from views on other entities. For example, a request to a supplier to create a sales order is built using the information from a purchase proposal. A view on the purchase proposal is converted into the sales order request.

Entities, views and actions are artifacts that we can describe, that a user can understand and that may help us describe services in ways that can be understood by business analysts.

I want to mention other artifacts that are less often used.

References
Many entities can be identified using key attributes. However, they can often also be uniquely identified using other attributes. Customers can be identified using their social security number, tax number, or DUNS number. A list of customers can be retrieved via the zip code, the region, or the account manager. We can formalize this through the concept of *reference*. A reference defines either a unique instance or a list of instances. A reference, like a view, has a schema.

The actions exposed by the service accept references to identify the entities they operate on. The references are a more formal way of defining possible input and a more generic way as well.

References formulate queries to services. Such a query may be as simple as the list of key attributes or as complex as the formulation of a complete set of conditions. Often a reference will consist of a combination of required and optional attributes allowing clients that have different information to obtain the same views and use the same actions through the same interfaces. Design of reference schemas is just as important and just as complex as the design of view schemas.

Relationships
Now, if we have a view of one entity, an order say, it should not be too hard to build a reference to a related entity, e.g. the customer for that order. In other words, we can transform the order-view into a customer-reference which refers to the customer entity. This effectively gives us a *relationship* between the order view and the customer entity.

A relationship uses the data in one view to construct a reference to another entity. Such a relationship can be defined using the schemas of a view and a reference. Consequently, relationships are not constrained to a single service; they can define relationships between entities encapsulated in different services.

If we are able to describe these entities, the views on these entities, the relationships between these entities and the actions on these entities in terms that are understood both by the business people and the software, we would have a description of the *'what'*. This description should

Example
Let me give a simple example. Suppose we retrieve an order – i.e. a view on an order – from a Web service. The order view may come in XML that, after much simplification, looks something like:

```
<ERP:Order OrderID="8245" >
    <Customer CustomerID="4711" >
        <CustomerAddress
              Name="Microsoft Corporation"
              Country="USA"
              ...
        />
    </Customer>
    <OrderLines>
        ...
    </OrderLines>
</ERP:Order>
```

Table 1: Simplified Order View

The view is not restricted to what is stored in the order table or in the order object. This view on an order includes some customer information and possibly some product information, so that it can be displayed or printed.

From such an order, we may derive references to the customer for the order:

`<ERP:CustomerReference CustomerID="4711" />`	Such a reference is useful to access services that share the same customer numbers
`<CRM:CustomerReference CustomerName="Microsoft Corporation" State="WA" ... />`	This reference would be useful to access services that do not share the unique key.
`<My:CustomerReference CustomerID="4711" CustomerName="Microsoft Corporation" State="WA" ... />`	Of course you can add redundant information, so that the reference can be used for all kinds of services. Ultimately there is no reason why the view itself could not be used as a reference and it may sometimes be very useful.

Table 2: Examples of Customer References

not only describe the artifacts and the relation between them, but also their behavior, what people can do with them and what software can do with them. This description can easily be linked to, and in fact has to build on, the existing description of the *how*.

Even though much of this information can be provided by the designer of the service interface similar to how WSDL is currently provided, an extension to WSDL cannot solve all of our needs. Relationships between entities offered by different services should not be stored in either, but would require storage external to these services. Instance specific information, such as the current availability of relationships and actions, as well as information created at run-time, such as tasks in a collaborative workflow, require run-time interrogation of the service and thus a run-time description. This is very much related to service contracts and the required infrastructure for providing run-time information about the state of a specific conversation.

Since views are what the designer exposes of the entities, much of the information is available in the systems today. Every service that was designed around entities provides most of the information needed and in many other systems the analysis has been done and captured in some form of metadata.

The values in these references came from the view on the order. The names of the elements and attributes may be different between the view and the reference. Sometimes it is even useful to merge in values that did not come from the original, if you want to restrict the result set to a specific department for instance, or, if you need a combination

"It is only a logical evolution that leads us from *how* to access services to *what* services offer."

of views as the source (e.g.: all customers that bought a product xyz from account manager abc). The recipe that gets you from the view or the set of views to the reference is called a relationship. The relationship can be defined as an XSLT or as a piece of code, it doesn't really matter as long as the consumer understands how to interpret the metadata and construct the reference. The relationship describes how the consumer of the services can construct a reference from a view.

References may refer to entities, thus allowing the consumer (the user or the client-side code) to decide which view to retrieve, or they may refer to a specific view on an entity. The reference schema alone is not sufficient for the client to retrieve a view. The client needs a description of which service provides access to the service, which method needs to be called, and how to call that method using the given reference. For any given view, there may be one or more references that can be used to obtain that view and for each valid combination we need the description of which method to call. Not all customer references can be used to retrieve all customer views. For example, an ERP service may not provide a method to retrieve customer information using the Social Security Number (SSN), but the CRM service may. So, you may go to the CRM service to retrieve enough customer information to build a reference to the ERP customer.

Lists have views too: a customer address list is completely different from a customer balance list. Both are views on a list of customers.

The descriptions of Entities, Views, Actions, References and Relationships would not be limited to schemas and recipes. These descriptions would encompass information *about* those artifacts. The descriptions need to be useful both to humans and to software, including describing:

– Which method needs to be invoked to retrieve a specific view from a reference.
– How long a view is guaranteed to remain valid, or how long it may be used to formulate requests. The prices in the catalog from Ikea are valid for a year, but what information can I use from a production schedule, and, for how long? What can we describe about these different types of information contained in views?
– Caching behavior and how to optimally retrieve and manage the offline views.
– Access rights on references and thus on relationships.
– Relevance of relationships for certain roles, allowing the user interface to display the most relevant relationships.
– Types of relationships, such as one-to-one relationships, one-to-many, one-way, hierarchical, sibling, 'hard' and 'soft', etc.
– How an aggregated view is composed and what the caching policy for the constituent views and thus for the aggregated view is.
– When an action should be enabled or disabled in the user interface.
– The action's impact on the service's state. Does it update state, is the update transactional, does it throw exceptions or offer retries, etc.

References do not always return a single instance. It is equally possible to have references to lists, like for instance:

`<ERP:OrderListReference CustomerID="4711" />`	A reference to find all orders for a specific customer.
`<ERP:OrderListReference ProductID="B-747" />`	A reference to a list of orders requiring a more complex query over the order items.
`<ERP:CustomerReference CustomerID="4711" />`	There is no problem offering a method that retrieves orders using a customer reference.

Table 3: Examples of OrderList References

Entities and Views in SOA

The differentiation between entities and views is important in a distributed environment. Data is copied between the systems and whenever the consumer of a service retrieves data, it will get a copy. When I request information about an entity, I will receive a copy. This copy is a view on, or a snapshot of, the state of the entity from a particular angle. I may receive the customer's financial view, or the customer's satisfaction view, or the customer's contact view. These views depend on what I want to do and what I am allowed to see. When I receive that information only a few

milliseconds later, the service may already have moved on and changed its internal state. The view may no longer reflect the internal state of the service even when I receive it, and the chance of my view being out of sync increases the longer I hold on to it. The differentiation between *view* and *entity* is an important recognition of the fact that the consumer has an outside view on the entity and that there is a time difference between creating that view and receiving it. It's like taking a picture; the picture is frozen in time, whereas the subject of the picture may move on. If you take a picture of a building, you will not see the entire building; you will only see it from one particular angle. The building will not change much after the picture has been taken, but the people on the picture, the clouds on the picture and even the trees on the picture will move. The snapshot is taken from a particular angle or viewpoint and frozen in time.

Because views are copies of data and may be or may get out of sync, the concept of references is important. They provide access to the source of the information, to the entity itself. They allow us to convey and store information and let the service that encapsulates the entity deal with the concurrency issues. References are less likely to get out of sync. Normally, that will only happen if the entity ceases to exist. To avoid this from happening, I would recommend that the entity be marked as being obsolete rather than to delete it.

As I said above, the entity is made up of the combination of all of its state, its behavior and its relations to other entities. Understanding the mechanisms that cover the relationships between entities and their views, their behavior and other entities should form one of the fundamentals of any distributed architecture and thus of any SOA. It would be a huge step to capture these mechanisms in ways that can be used by the business people, the designers, and developers as well as by the systems.

These concepts are very similar to what we know from object orientation. However, there are differences. The most important one, I find, is the notion of distance between consumer and supplier of the functionality. The data has to travel from within the service to the consumer. One of the consequences of this is that, since the data will not be locked on behalf of the consumer, the consumer has to be prepared for changes to the service's state[i]. This leads to the concept of views. This notion of physical distance and the consequential temporal distance between requesting information, providing information and using that information leads to a whole new field requiring exploration[ii]. For instance, what is the behavior of the service when it is receives a request that uses older data? What are the guarantees the service will give about that data? When I issue a price list with prices valid through January 1, I guarantee that I will honor those prices. When I request a stock price, there is no guarantee whatsoever. What are the possible guarantees or service level agreements and how can we describe these?

How Can *What* Be Used?

Exposing all this information is extra work for the designers and developers of the service. This extra work will only pay off if these services are consumed using this new description. We don't only need infrastructure to capture and expose the functionality offered by the service, we also need the infrastructure to consume that information and offer it to business analysts and users so that they can do useful things. In the following, I present only a few examples of infrastructure to provide better service to users and thus to offer better human interaction.

Entity Aggregation

Formalizing the concepts of entities, views, and references provides a good starting point for a model for entity aggregation and for information integration in general and thus for an information architecture. By providing infrastructure and tools for information integration, building on a description of services in terms a domain expert understands, a business analyst will have more influence on the way the business is organized.

Let me revisit entities. Earlier I wrote that a service encapsulates entities. If we look at the customer entity, many organizations will have multiple services defining customer entities. There could be an ERP system with an

[i] See for instance my article: "Dealing with Concurrency: Designing Interaction Between Services and Their Agents" at http://www.msdn.microsoft.com/library/en-us/dnbda/html/concurev4M.asp.

[ii] Pat Helland has an interesting article on this subject "Data on the Outside vs. Data on the Inside" at http://www.msdn.microsoft.com/library/en-us/dnbda/html/dataoutsideinside.asp.

"Relationships are not constrained to a single service; they can define relationships between entities encapsulated in different services."

ERP customer entity, a CRM system with a CRM customer entity and many other systems as well. Users might want to see an aggregated customer entity that has aggregated views as well as the original ones and that has aggregated actions as well as the original ones and that has all the relationships from the original ones and possibly some new ones. We want the business analyst to define these aggregations and reduce the required amount of support from the developers.

In entity aggregation, entities are combined to form new entities. These entities may live in the same service or may come from different services. Entity aggregation, or information integration, has two main aspects:

– Aggregating information
– Aggregating functionality

I will discuss these separately.

Aggregating information

An aggregated entity is defined by the views on that aggregated entity and by specifying how these views are built out of the underlying views. A business analyst can build new schemas using the existing ones, taking into account constraints such as whether the access rights to these underlying views match the intended audience for the aggregated view.

A tool should be able to see how the information can be retrieved from the specification of the composition. For instance the tool could decide that when presented with a CRM customer reference, the CRM view should be retrieved first and then a relationship from that view to an ERP customer reference could be used to obtain the ERP information. It could define a different path when presented with an ERP customer reference.

When the information needs to be replicated, the tool could look at the caching policies of the constituent views and provide replication for those and offer aggregation based on the cached views. In other cases it may try to minimize the number of constituent views. For example, by retrieving an aggregate view of the ERP customer, retrieve and cache that and then derive the smaller views from the cached aggregate.

Whichever the best algorithm for the specific problem, a tool should be able to offer working solutions using the information given by the description of what the services have to offer. The metadata should provide information that helps the business analyst define useful combinations and make appropriate policy choices and it should provide information for tools and run-time to support and execute both the decision process and the outcome.

Aggregating functionality

Actions on such an aggregated entity will often have to touch the underlying entities. In some cases this has to be an "all or nothing" action and in other cases it has to be made certain that all individual steps are executed. Both are examples of what I call short running processes or activities. A tool could concatenate actions based on their inputs and outputs. Moreover, given a description of the underlying actions or methods, including a description of their transactional behavior such as whether they update state, throw exceptions or offer retries, a tool could concatenate the underlying actions in a logical sequence to minimize chance of failure and inconsistency and even raise an event or send an email in case something does go wrong after all. Of course there is no reason to restrict this to entity aggregation, these types of activities happen all the time. This is generally applicable in the area of service orchestration.

Again, a description in user understandable terms allows a user or an analyst to combine and order a set of actions and build larger actions. A description of behaviors and side effects of actions allows a tool to make sure that such aggregated actions behave predictably and can be controlled and managed.

Even with good tool support, defining good schemas for views *on* and references *to* entities remains difficult. We should put more effort in providing guidance in this area. One of the major challenges in this area is designing for change, where the different artifacts have a different rate of change. Data obviously changes much faster than schemas do and some types of data change faster than others. The definition of the data may change while the definition of the processes remains unchanged. The reverse of this may happen too. Views, references, actions, code and process definitions will often change at different rates and at different times. A discussion on this goes beyond the scope of this document. Nonetheless, it must be clear that a good design of all of these artifacts must include a concept of versioning even in the first version.

Collaboration

References play an important role when crossing the boundary of organizations where one wants to make sure that both parties collaborate on the same subject. Often this is

complicated by the fact that not all information can or may be shared. For instance when one physician sends information about a patient to another it is important to be able to send an exact reference to the other party, to unambiguously identify the patient, but both parties will have access to their own private additional information. The other party will have different information available and will access different services to obtain that information, but the reference may still be standardized for the domain and be communicated. The key attributes and therefore the data model do not have to be the same on both sides. It suffices to agree on an interchange format for the references.

On the other side, the work of collaborating users that do have access to the same services could be coordinated by those services if the users communicate references instead of chunks of information. Multiple users can request changes from the service and the service will resolve the concurrency issues. The single truth is clear to all in the coordinating service.

Microsoft Office Information Bridge Framework (IBF) uses references in documents to communicate information and to allow the user to add business context to a document. References can be used to convey not only a limited piece of information, such as an expense report, but also to provide access to a wealth of information, such as the trip report, the customer opportunities, and other information related to such an expense report. If the recipient of the information has a description of the references and of relationships used in the document, this user can then do more than just consume the information compiled by the sender. Related relevant information will automatically be available and offered as well.

Many collaborations or workflows designed today store the data with the flow. The flow then just runs by itself based on that data. It breaks as soon as someone does anything in the backend system that influences the flow. For example, when designing a flow for doing a stock trade, the progress of the trade is stored in the workflow system and the flow tries to just go ahead. If someone blocks that trade in the backend system, the flow runs into an exception at some next step. In other words the designer should have synchronized the state of the flow with the state of the backend system.

References are also a way to avoid unneeded duplication or even multiplication of state – state that may, and often will, become out of sync. Formalizing references will reduce, not completely solve, the problem.

This is true for workflows and for document collaboration. There is no difference between using references in workflows, documents, e-mails or in instant messenger. In all cases I have the choice between sending the data itself or a reference to that data and in all cases I have to manage change between many copies when I send the data or I have a service manage it for me. If I send a document to ten people for review; each person gets a copy and everyone makes changes. If instead I send everyone a reference to the document, each person can edit the document and the changes are synchronized by the service. Using the references avoids making many copies and having to manage those copies. This can now be delegated to a service that was already designed to deal with concurrent updates and that service does not care whether the references were sent by email, instant messenger or as part of a workflow.

The main points here are that in designing for collaboration and processes in general:

1. Avoid building a parallel data structure, but use references instead to make it easier to develop robust flows and collaborations
2. Eliminating different types of payload makes designing the system easier. References have behavior: the actions defined by the service description of the entity. The workflow can use this behavior.

The concept of references helps reduce synchronization issues and this helps reduce the need for conflict resolution. It also helps convey uniquely identifiable information without the need to align the systems, which then can be used to identify related information by following the relationships. Capturing these concepts makes them available to tools, infrastructure, and thus to humans.

User Interaction

The description of views and actions effectively exports this functionality to the consumer of the service. A consumer can use this to provide presentation for views on entities in the user interface. The step to portals seems a natural continuation. For instance a portal can define parts to show views on entities. A part showing the content of such a view to the user can provide access to the actions, using buttons, or smart tags,

thus exposing these actions to end users. The metadata can provide hints to the user interface on how to expose the actions and the run-time description can provide hints as to whether certain actions are available at this time for the instance that is currently displayed.

If these parts use portal infrastructure based in the proposed metadata, then that infrastructure would know which view on which entity instance is being shown. It could then offer the user a choice of relevant related instances that the user could navigate to. IBF was the first instantiation of this idea and offered a portal in the task pane.

Not only presentation, but also offline availability would be aided by a description of the views and their relationships. Service agents manage offline and cached information on behalf of their consumers. I could see a design where if such a service agent is triggered to take an instance offline, it uses the relationships to see what further relevant information should be taken offline. It uses the relationships to find the relevant instances and to prune the offline tree by weighing that relevance. For instance, if the calendar takes an appointment offline for a customer visit, by following the reference in the appointment the customer information can be taken offline, and by following further relationships, the problem reports can be taken offline as well.

The concepts of portal parts and service agent infrastructure form the basis for a smart portal infrastructure. The addition of the extended service description provides user interface with the ability to generically provide actions to the user and enable or disable these actions as appropriate. Moreover the description of relationships between the views provides navigation between the parts. The description also offers service agents the opportunity for more intelligent offline behavior. All in all, the description of services offers much to make a smarter portal.

What Should Happen Next?

It is only a logical evolution that leads us from *how* to access services to *what* services offer. Many organizations and vendors already describe the functionality of their services in metadata. It is only a reasonable to expect that the industry will standardize on the way to describe these services. It is equally reasonable to expect that it will take time, a lot of time.

The *Sea Urchins* memo was about *how* to communicate. Razorbills is about *what* to communicate; until she can "raise her bill" the sea urchin will never "see her chin." Just like the original *Sea Urchins* memo discussed the need for both a method and tools for sharing data, for functionality, we will need infrastructure to describe and expose *what* functionality services have to offer. But such infrastructure alone doesn't bring about change. So, we will need applications and application infrastructure to use these descriptions. One will not be available before the other, they will have to grow up together and in the mean time, we will want to use some of the concepts already. We don't want to wait for this entire infrastructure to be available, but we do want to use it as it becomes available.

Raise Her Bill

Standardization on exposing what services have to offer requires an industry effort in the following areas:

Extensible model to describe what services offer

This, like the current WS standards, should offer a fixed basis and be extensible in most areas. For instance, multiple standards describing caching behavior or the guarantees on data validity might evolve and each of these needs to be plugged in to the overall description.

Extensible infrastructure for that model

It is important to provide a model that, like WSDL, can be queried by both tools and service consumers. However, it is equally important to not just provide static information. The effort needs to align with our thinking on service contracts and should include dynamic definition and redefinition of views, actions, references, relationships and even entities. Such as, providing the current availability of actions and relationships on a particular instance or provide information about an instance of an ad-hoc workflow. Building on the extensible SOAP infrastructure, this should become part of the .NET Framework.

Shared infrastructure for storing inter-service description

The description of everything that transcends a single service needs a place to go. This includes relationships between entities in different services as well as the description of user interface or hints for user interface. I am hesitant to call for a directory infrastructure on top of UDDI. Distributed directory infrastructures are gaining popularity and seem to offer good alternatives. Important in this context is only that consumers of these services can store and query a description of inter-service relationships.

After the *Sea Urchins* memo, it took us a few years to be convinced that XML was the way to communicate and we're still struggling to define all of the standards and required infrastructure. It will take time before the above starts being realized and the razorbill will have proudly raised her bill. However, would this be enough for the sea urchin to see her chin?

See Her Chin

For the above infrastructure to be visible and usable, the industry would have to provide tools, applications, and higher level infrastructure that allows the users, the business analysts, and even developers to build on that infrastructure. I will suggest a few here and others will be able to extend this short list with many other useful examples. Not all of the above infrastructure needs to be in place. Some of these can start using more proprietary solutions. When solutions like these become available, everybody, including the sea urchin, will be able to see the razorbill's chin.

Build tools for Entity Aggregation and Service Orchestration

Bring aggregation and orchestration to the domain of business analysts where it belongs. In Microsoft, BizTalk would form an ideal basis to build tools for this.

Provide User Interface infrastructure to browse the business

Build a Smarter Portal defining default presentation for views, thus allowing navigation from view to view, across entities and across services.

Provide a Framework for Service Agents

Such a framework would enable smart clients to deal with intermittent connectivity and caching and queuing. Using the relationships, service agents could not only take views offline, but could also provision relevant related information.

What Should We Do in the Mean Time?

Putting all this infrastructure in place will be a multi-year effort. To prepare for this future, I recommend those building services:

Differentiate between entities and views

Distributed computing and therefore Service Oriented Architecture and Design introduce the necessity to think about the distance between the services and between the service and its consumer. Differentiate between the entity information contained in the service and the copies of information defined by the views. Put these views into schemas and assemble a collection of reusable schemas. Build on existing industry specific schemas when available.

Define and describe the properties of these views. For example, manage access to information by means of views and examine how the service honors the information it handed out in accepting requests that use that information.

Formalize the concept of references

When defining an information model, model references, build services to use these references, abstract away the key attributes, think about the defining attributes of the entities and schematize this to allow other services and humans to interact with these services.

Define your own metadata
Regardless of where the metadata standardization effort is going, you will be able to reuse the effort put into the analysis of service interaction. Making the parts of your software that consume services more generic, providing more consistent user interaction and providing more relevant information to the users will be easier if you have this metadata in place.

Conclusion
The information architecture is the model for interaction between services at the semantic level. It is the model for interaction about *what*. Much like the programming model describes how to design services for programming their communication and coordination, the information architecture describes how to design services for human interactions.

Capturing and exposing what services have to offer, in such a way that the functionality can be used is an important and crucial part of the information architecture. Doing this in business terms will provide business analysts and users access to more, and more relevant, information and functionality. Opening up services to business analysts and users allows them to create flexible content and to create flexible processes, and it provides them with information and functionality. This not only drives down the cost of business and the cost of change, it also gives them the chance to seize opportunities that were not even visible before.

Addressing these challenges will help our industry address another set of business requirements: Information should not just be available; people should be able to find it, be able to identify the relevant parts, act on it, convey it and interact with it.

The sea urchin not only lives, it started swimming. We use XML as the communication mechanism between services, and the industry as a whole is working hard to describe even better how to communicate between services.

Now the razorbill should raise her bill to show what services can offer so they can be used by other software and humans alike. If the industry takes on this challenge, the razorbill will not only swim, but fly as well.

Maarten Mullender
Microsoft Corporation
maartenm@microsoft.com

Maarten Mullender is a solutions architect for the .NET Enterprise Architecture Team; this team is responsible for architectural guidance to enterprise architects. During the last seven years, Maarten has been working at Microsoft with a select group of enterprise partners and customers on projects that were exploring the boundaries of Microsoft's product offerings. The main objectives of these projects were to solve a real customer need and, in doing so, to generate feedback to the product groups. Maarten has been working for more than twenty years as a program manager, as a product manager and as an architect on a wide range of products.

Next Generation Tools for Object-Oriented Development

By Oren Novotny, Goldman, Sachs & Co

Background
The file has been the central container of work in software development for over thirty years. All of a project's structure and logic is ultimately reduced down to files and directories. The tools that surround software development are built around this concept, too; compilers, linkers, and even language features rely on source code files as input. Version control systems mirror the file system's structure, maintaining a copy of every version of each file monitored. For decades this has been acceptable and has even become standard. Languages such as C are built upon file references in the source code via include directives.

The Problem
In the mid-1990s, the idea of Object-Oriented development finally began to gain momentum. Languages like C++, once formerly constrained to small research projects, now became the mainstream. With this fundamental shift, the file-based system of development was dragged along as a relic of old. C++, like its predecessor, also uses file includes to resolve dependencies. Despite new techniques and languages for modelling and designing software, such as UML, in the end source files still needed to be created and linked to each other. This linkage leads to a duplication of effort as models must then be translated into files and source code. Complex build scripts must also be written to ensure the proper building of the application. Should a model change, the source needs to be updated to reflect the change; conversely, should the source code change, the models need to be updated. A number of different software solutions have been created to smooth the synchronization of these disparate files, but they're still developed under the fundamental notion that the file is the central unit of output.

The file also proves to be a difficult challenge when working in a team environment. If multiple developers check a file out from source control, then the version control system must ensure that conflicting changes do not occur. If they do occur, then manual intervention is required to resolve the differences. The source control systems are limited since they rely on the file as the unit to control. The source control systems are completely unaware of the language structure contained within the file. It seems clear that the file is a relic that has outlived its usefulness for software development.

Model Driven Architecture
In late 2001, a new methodology for software development began to emerge: model driven architecture (MDA). The fundamental principal behind MDA is that the model, not the code, should be the central focus. Platform independent models are created in UML and then undergo various levels of transformation to eventually wind up as source code for a specific platform. It stands to reason then, that the model, not the file, should become the new unit of output. A model has many different views at different levels of abstraction. At the highest level, platform independent components can be specified in analysis; at the lowest level there is a platform specific implementation that reduces to a set of classes in code. The available modelling notation allows for the definition of all necessary features needed to replace files; specifically, dependencies between classes can easily be marked by means of an association arrow with a dependency stereotype.

Tools that support the creation of models should ideally allow for version control on each individual model element. Each element should have the ability to be checked in and out of a repository. Similarly, the modelling tool should work with the repository to provide branching and merging support at the element level. For example, a developer should be able to connect to the repository and obtain a list of elements that meet a given need. The developer should then be able to branch the repository element for inclusion in a new model. At a later date, another developer might wish to reconcile differences between branched elements in different models. The development environment should support this. One common example of when this functionality is needed is for bug fixes. If a bug in a model element is fixed in one project, then other projects that are using the same model element might wish to incorporate those changes; this should be an easy task to accomplish.

The Solution
The solution to these problems is nothing short of a complete paradigm shift. The development tools must support the shift that languages have already made and fully support MDA development. It is easy to argue that the source code is, in reality, just another model. The argument is exemplified by the existence of a language-independent Code Document Object Model (CodeDOM). When you get down to the basics, the source code and the UML classes are simply different views of the same underlying data. Any separation between them is an artificial construct that needs to be eliminated. A developer should be able to create classes as UML, complete with methods and attributes, and toggle between the UML and the

"Development tools must support the shift that languages have already made and fully support MDA development."

source code. The source code should be available in many different languages – simultaneously. This is largely possible today with CodeDOM, which is essentially an object-oriented meta-model that can be easily mapped to UML. The code structure is easily portable between most .NET compatible languages and the tools should support this.

One artifact that must be eliminated is the source code file. It serves no function in MDA that cannot be better done by different means. In place of source files, an IDE would instead connect directly to a project repository. The project would then list the objects directly, bypassing the need for files. This new functionality would work similarly to the "Class View" currently available in Visual Studio. The view would be organized by nested namespaces rather than directories and by classes, structs, and resources instead of files. When a developer double-clicks the class, the code is opened in the editor.

The Class View would also contain UML diagrams. A diagram would be able to be created at any place in the hierarchy, and the developer should be able to drag the classes onto the designer surface. Each namespace would have a default diagram and any number of supplementary diagrams showing different parts of the system. The tool would have a graphical view of the classes and their relationships in addition to a code view.

Ideally, the code should not be stored in any single language; it should be stored as a model with additional CodeDOM structures. Minor additions to the CodeDOM classes could enable the representation of all language features, eliminating the need for language-specific code snippets. Each method in the model can contain language-independent CodeDOM constructs which can then be translated into any supported language for easy editing. Just like there can be a split-view of design and HTML view in FrontPage, a development environment should provide a split-view of model and code.

When file-based development is replaced with model-based development, a new world of possibilities is opened. Rather than designing websites by creating web pages directly, Activity Diagrams can be constructed to model the user's interaction with the site. Upon compilation, the tools would generate the necessary output files to support the diagrams. Each activity on the diagram that requires a user interface would contain a UI design. For example, for any given activity, a web UI design and Windows UI design could be a part of the activity. The tools could allow the import/export of HTML files to allow designers the ability to use their own tools rather than the default IDE. After a designer is done, the HTML file would then be imported back into the activity. This whole concept falls squarely in line with the new "Whitehorse" technologies Microsoft is including in Visual Studio 2005.

Visual Studio 2005 will include support for Microsoft's Dynamic Systems Initiative (DSI) by providing a tool able to create a System Definition Model (SDM) linking components to hardware. Whitehorse includes a set of new designers in support of MDA called Software Factories [JG04]; a UML-like class designer is available for source code and logical application and infrastructure and designers allow developers to declare the application component structure, configuration and deployment settings early on in the design process. Like code files are now, these models are currently stored as files within the solution. All models should be stored within the repository directly with APIs created to access the models from the repository.

The source control system should also be cognizant of the inherent structure present in the model and code. It should support the check-in and check-out of model elements down to the function level. That way, two developers could work on different parts of a class without the potential for later reconciliation.

The build tools also need to support this new environment. Rather than compile files, what they really need to do is compile a model. By way of a deployment model, classes would be associated with assemblies and executables. The compilers would look to the deployment model to determine the physical separation of the project's classes and resources, and to the class dependencies to resolve symbols. The build process would output many different files depending on needs; for a web project, the ASPX or ASMX pages would be created alongside the binary assemblies. The build process would also read the SDM and generate the appropriate setup files and export the SDM for import into a deployment tool.

For compatibility purposes, the tools should allow for the extraction and import of source code files in any supported language. A developer could select a given set of classes to export and the tool would generate the required files. A developer could then

modify the files and then import them back into the project. The tool would automatically create the model elements and prompt for reconciliation changes if needed.

Conclusion

The modern development environment has not yet fully caught up to the object-oriented shift. All of the tools still rely on a file to be the container of source code. While some contain modelling capabilities as well, the models exist as different entities than the file and its code. The tools need to evolve to support modern software development; they need to merge the model view and the code view into a single entity.

While this may seem like a radical shift, and it is, the new tools will be natural to developers. Developers already expect their classes to be organized by namespace; they do not really care about which file or directory contains what code. If a developer changes a method signature, then he will want, and expect, to find all references to the method to fix them. Architects already construct models of software and systems so they can understand them.

References

[JG04] Jack Greenfield, The Case for Software Factories, JOURNAL3: Microsoft Architects Journal, Issue 3, July 2004.

Oren Novotny
Goldman, Sachs & Co
oren@novotny.org

Oren Novotny is a software developer with Goldman, Sachs & Co with an interest in all technical things. Previously he was a solutions architect with Unisys Corporation. Oren is also a freelance consultant frequently helping in various open source projects.

JOURNAL4

Project Management
Content Master Ltd.

Design Direction
venturethree, London
www.venturethree.com
Orb Solutions, London
www.orb-solutions.co.uk

Orchestration
Richard Hughes
Program Manager, Developer and Platform Evangelism Group, Architecture Strategy, Microsoft Corporation

Foreword Contributor
Chris Capossela
Vice President, Information Worker Business Group, Microsoft Corporation

Part No: 098-101154