MIDDLEWARESPECTRA

incorporating FINANCIAL MIDDLEWARESPECTRA

Contents

November 2005

Volume 19 Report 4

10 signs that may indicate your IT architect has lost the plot

Adam Richards
Managing Director,
Architecture and Performance Engineering
JP Morgan Chase

Management introduction

Adam Richards was Vice President, Architecture and Oversight at Charles Schwab, also in San Francisco for many years. He joined Schwab from IBM's Hursley Laboratory in the UK.

Mr. Richards has since become Managing Director of Architecture and Performance Engineering in the Data Center Technology Group, which is part of the Global Technology Infrastructure Group, within JP Morgan Chase (JPMC); he is based in San Francisco. One of his objectives is to deliver cross-pollination between skills and specialties in infrastructure — so as better to support the Bank's individual lines of business.

In this mildly 'tongue-in-cheek' analysis — written before he joined JPMC — Mr. Richards reflects on his experience over the past two decades in IT to produce a commentary on the some of the warning signs that exist which can indicate that an IT architect (or group of architects) may no longer have your organization's best interests at the center of their focus. He has organized his commentary into a classic 'Top Ten' list — running from 10 to 1 — of the most common indicators that he has encountered.

Background

All IT Architects need to be partly dreamers and sayers of sooth. That makes them somewhat eccentric when compared to the 'normal' IT crowd. It is, I think, an unspoken truth that in every organization there are a few architects who pursue agendas which are not really aligned to the corporate agenda but are more personal than that and who, if suitably funded, lead to much wasted time, effort and money.

In old horror and James Bond movies there is a cliché, that before Dracula or the head villain appears we learn of his presence by hearing the organ music playing. The assumption is that the solo control of such a powerful instrument externally is a symptom for the madness within. So, by analogy, I have compiled a list of signs I have observed which tend to suggest that somewhere nearby lurks an architect who has, as the Australians would say, "a few 'roos lose in their top paddock".

#10: Open source is free

One common symptom of an architect having lost the plot can be recognized if you hear the words: 'open source is free'. This is a comment I have heard argued far too frequently and recently.

Of course, 'open source' is free — that is until you actually want to use some of it in production for real. At this point the organization 'discovers' it has to pay somebody, even if it is only itself, to provide the support — and possibly the improvements to adapt it to your environment.

In reality, most organizations do not wish to rely on themselves for 24x7 support of mission-critical systems. Rarely do they want to pay to have experts in-house who know everything about all the technologies they run.

The alternative is to hire third parties to deliver the support — which is common in the Linux world. That can work out extremely well if the open source code you wish to use is something that is a pure commodity — say, Linux or an open source database like mySQL — when the supplier can deliver sufficient economies of scale to make that particular form of open source attractive.

But, if you want to exploit some esoteric open source code — that maybe you or another individual (such as the architect) has invented — then simply saying that it is 'open source' will not solve the problem about the cost of support. There is no way round this.

The only people or organizations that can afford to support open source directly are associated with large enterprises. They can afford the support cost because they can spread the staffing load, and cost, broadly enough to make the investment worthwhile. In my personal view, some much smaller operations can afford unsupported open source — if they do not have too much at risk from any failure: it (the open source) either works or it does not work. If it does not work the loss or exposure is minimal (and manual methods, or other alternatives, can be applied).

The key to using a piece of open source code is not that it is free but that it possess a critical mass of people supporting it. Only when this has happened do you gain the opportunity to ride the open source coat-tails — and even then it will still cost something, somewhere. In other words, using open source to try and gain some sustained strategic advantage can be a risky approach to adopt. Believing your architect when he or she says 'open source is free' can prove expensive.

#9 Use a perpetual data machine

You can recognize a crazed architect when he or she believes that all data can be both completely available and completely consistent at all times. This is what I call the perpetual data machine.

This is a concept which is devoutly to be wished for — but, unfortunately, breaks the laws of physics. Likewise, it would be wonderful to possess a machine which could run forever and never consume any energy. The brutal reality is that you cannot. The notion of a perpetual data machine is not possible. You simply cannot have data be entirely consistent and completely available all the time. You have to pick one of these (much as Heisenberg's Law intimates).

Nevertheless, there are scary people out in large and small organizations (especially large ones, however) who seem to think that because they would like something to be true, it is just a matter of will to make it true. This is a dangerous delusion, and one that can be very expensive if it is not nipped in the bud. I have even seen some, unshackled, architects attempt to build such a monster, only to be frustrated by cold reality.

Spotting the symptoms is relatively straightforward (usually). A person who believes in the perpetual data machine will say things like 'DB2 and Oracle have no magic'.

As so often, this is true at one level: these databases do not have any magic. But they do represent the fruits of an enormous amount of hard work put in by many innovative people over many years.

Another indicator of this kind of devotee is the pursuit of any new database (or similar) start-up software company and the consequent belief that anything that the start-up says, regardless of how small it might be, must be right. Thus a 2-person start up which says 'we can do this particular sort of thing' will enable the devotee instantly to be thinking that this is software that their organization must try out, and try out right now.

#8 Data everywhere is a good thing

The statement that 'data everywhere is a good thing' is the most obvious indicator of this particular delusion. It has connections with #9 but is not quite the same. Typically you will hear suggestions like 'why don't we solve this problem by just setting up another replication bridge'.

The difficulty with data everywhere is that the concept of the 'data of record' becomes so diluted that it becomes useless. It is then an unworkable concept.

Another indicator is the statement 'data storage is now so cheap that we can store anything, anywhere'. Physically and economically this may be true — but it is hardly manageable.

A variant is the notion that it is 'better for everybody to have local control over data rather than it be located somewhere else where there is certainty about the state of that data'. You may not have met them but I assure you that there exist some people who believe that infinite distribution (of data) is simple and 'a good thing'. The difficulty is that what you end up with, over a period of time, is chaos. You do not know what is correct.

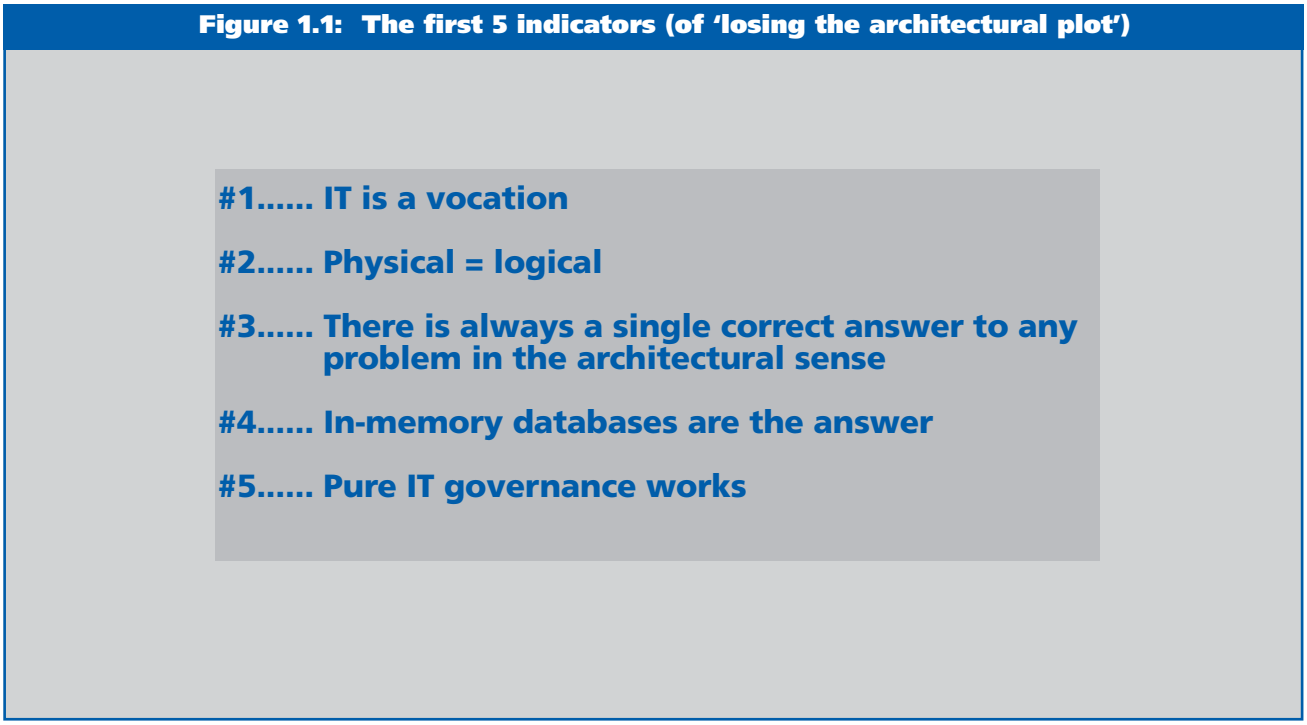
Much of this 'delusion' has to do with the natural tendency for some individuals to want control. I have a friend who has a rule of thumb which says 'if you cannot prove to somebody that a centralized, shared system is 10-15% more effective than a non-shared one, then the attraction of local control will almost always win'. These devotees want to possess their own data — and not to share it.

What happens over time is that you end up with all data everywhere and nobody is certain as to what the right answer to any question is. In a sense this is what data on the Internet is. It is great if what you are doing is trying to find recipes for pot roast. It is extremely bad if you are trying to work out accurately how much somebody's net worth is or what is the balance on an account — and convince customers that you, as a commercial firm, actually know completely and accurately what their financial position really is.

#7 The completely static enterprise data and service model is achievable

This type of IT architect is a person who understands that

Figure 1.1: The first 5 indicators (of 'losing the architectural plot')



componentization is a good idea. There is no doubt that object orientation can be useful. But then to go out and say that an entire large enterprise can be completely modeled is going too far.

What I mean is that any decent sized enterprise is constantly changing — especially the ones with which I am familiar in the finance sector. In practice it is my experience that most enterprises are changing faster than you can model them.

Indeed, there are specific forces at work which prevent you from being able to model them — simple aspects like unifying name spaces, even conceptually, are a complete a nightmare to model. People just do not want to adopt a single definition of customer or supplier or product code or whatever.

The classic symptoms of the person who thinks that accepting such a negative position is just hide-bound thinking — which you just need to 'get over' — says that the first step in any architecture is to 'decide on a common nomenclature'. You then endure a six year project attempting this, which invariably comes to nothing.

Another example was when I heard this said: 'how dare they call that a customer?'. Why did they ask that? Because the disputed definition conflicted with the 'standard' definition of 'customer'.

I have even heard of one company which succeeded in building a master data dictionary which held all the definitions for that enterprise. It was intended for the developers but it used a low priority batch process to make enquires to the data dictionary; this had to be run during the overnight batch window to obtain a desired definition. If the batch window closed before the request was run, a developer might be waiting two days for his definition. (This dictionary took years to refine but did not outlast a new CIO who saw that, however accurate it was in practice, it was an expensive liability rather than an asset.)

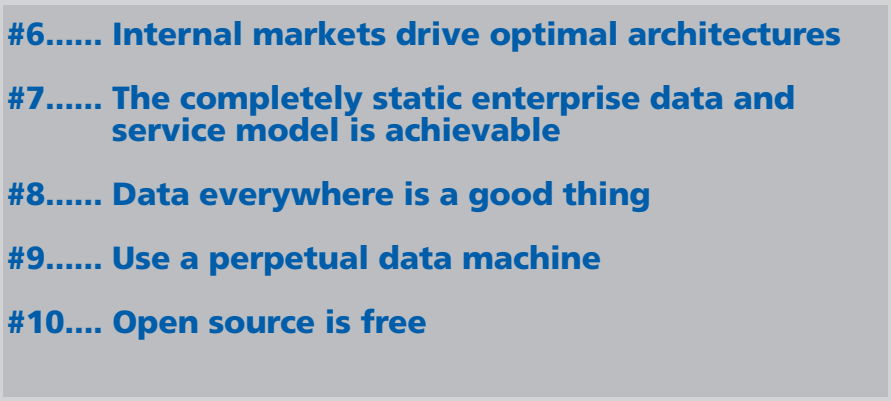
#6 Internal markets drive optimal architectures

These architects wish to believe that an internal market will drive an optimal architecture. On one level you have the kind of people who have the sort of vision that says 'there is a wonderful architecture out there, that gleaming city on a hill, and we should all march towards it'. On the other side you have the laissez faire attitude that says 'it does not matter, it is all going to be chaos anyway and chaos is good'.

I use a couple of illustrations from the real world to show that unregulated markets are not always pretty. In fact they are almost always never pretty.

The San Francisco sewer system is my first example.

Figure 1.2: The next five indicators (of 'losing the architectural plot')



Someone decided years ago that it would be cheaper to introduce a unification of the storm-water drain system and the foul-water sewer system. The practical result is that every time it rains hard in San Francisco, the drainage system is overloaded and sewage flows out into the street. Once this sort of erred thinking has been put in place, it becomes almost impossibly expensive to retrofit or fix.

The other illustration from California involves the environment in the Sierras around where the gold rush occurred in the nineteenth century. There people washed away whole mountains in order to try to find the gold that was alleged to be in the rock — and the resulting tailings washed downstream to block up rivers. Even today there remain high levels of lead and all kinds of other metals in some of California's rivers as a result of this.

There was no control. People did whatever the market wanted. Get in; get out — quick.

In IT, the symptoms of this are someone saying something like 'why pay to retire anything?' A different way of presenting this argument is 'planning is a nuisance and not worth doing'. Often this is derived from a general kind of feeling that you think you recognize.

The inverse is actually true. In general the one thing you most want to eliminate is going to be the oldest system and, because it is the oldest system, it is:

- **probably the one that you depend on most, often in a cash-cow part of the business**
- **in an area of the business where no one wishes to invest anymore, because it is a cash-cow part of the business.**

Let me put this in a different light. Imagine yourself being part of a group that has to have to support some technology that you would not invest in if you were starting from scratch. In fact the problem can become worse — in some cases you may actually want to get out of such a business — but there is not even the money to deliver an exit.

In my own view, saying 'the internal IT market will deliver change' implies that that business will eventually disappear. It will be displaced by a competitor that is prepared to invest and which, in so doing, offers a product or service that is fundamentally cheaper or better or more attractive.

A variation on this architect's error is saying 'people local to the decision will make the best choice'. If you think about this is in the context of pure market forces, this is tantamount to agreeing that whatever a person's view is, that view is the best way forward for the whole organization. To me there is only a small possibility that such a decision can be taken in a narrow, local, context — and still deliver economies scale to the whole organization.

But a lot does depend on the nature of the enterprise. If it is a GE-type of holding company where there are many disparate businesses, then local autonomy (to the business unit) may be best. But most enterprises can obtain economies of scale because the business model works only when their many parts work together: the whole is bigger than the sum of the parts. In this instance, internal markets do not drive optimal architectures.

#5 Pure IT governance works

Pure governance in IT is a superficially attractive chimera. Pure governance is where an architect, or architecture group, selects and then sets a bunch of standards in the belief that these can describe accurately enough the criteria that are needed to deliver any platform or application selection. The notion is that any idiot, or monkey, arriving at the same decision point will come to the same decision.

My personal view is less charitable. I say that pure governance is the 'best' way to educate and train people in today's fastest growing engineering discipline — which is reverse engineering. People arrive with 'I want platform X; I cannot have platform X; how can I change the criteria and use the rules so that platform X is the result?'

You can design as elaborate a system of checks and balances as you like but what you are doing is swimming upstream. What really goes on involves a kind of gentlemen's agreement — to pay lip service to the standards but then work around them. This is not only counter-productive but invariably expensive.

My preferred alternative is what I refer to as 'non-pure governance' — where there is a set of rules and there is also a set of people. The people are injecting their view of the world into each critical decision, using rules as guidance but not as law. This is the only way that you will actually start driving towards coherency, at least in my opinion.

#4 In-memory databases are the answer

I do not know why but in-memory databases are constantly popular as the 'new thing'. For whatever reason, everywhere I go I seem to run across people who have rediscovered the in-memory database and think that Oracle and

DB2 represent giant conspiracies to prevent them doing something useful — and why won't everybody else wake up to this analysis. (By the way you can substitute other kinds of technologies for 'in-memory database' but the in-memory database seems to be the one that emerges most often.)

In fact, all modern databases manage data in memory as part of their storage hierarchy. The distinction that is being made by an in-memory DB proponent is that somehow they have reduced log I/O, or allowed multiple horizontal nodes to share data, without apparently compromising data integrity. But as we know from 9 and 8 (above), nature cannot be fooled. Something has to give. Most often that something is first the budget and, subsequently, certainty about data accuracy.

#3 There is always a single correct answer to any problem in the architectural sense

I recognize incipient architect dementia when I hear it said, as has happened far too often, that 'there is a single correct answer to this problem'. In my personal experience my one big conclusion is that this is usually a sign that whatever is that 'chosen single correct answer', and that person's view, is almost certainly the wrong solution to whatever is the problem.

By one of those strange quirks, if someone tells you this system must be distributed, you can almost bet that the reason that they are so vehemently on that side of the fence is because there is a better argument for why it should be centralized. It is as if the proponent of the 'single correct answer' knows this and thinks that the only way to obtain what he or she wants is to hector until their 'answer' is accepted. Having a meaningful discussion is disdained for imposing an orthodoxy that seems to say that 'the world should be like this and, therefore, if you do not see it this way you must be a member of an unorthodox sect (or just stupid) if you support any alternative position'.

Some symptoms of this 'condition' are easily recognized:

- **'distributed is always best'**
- **'centralized is always best'**
- **'asynchronous is always the answer'**
- **'my single threaded, single I/O database is the only practical way we can obtain the necessary throughput'.**

The moment you encounter any such orthodoxies like these you can usually bet that this is hiding or fighting against something else. You just have to work out what, and why.

#2 Physical = logical

Simple indicators of this condition include:

- **'physical equals logical, and damn the torpedoes'**
- **'my beautiful architectural picture must be implemented in real boxes'**
- **'please call off the server to my intermediary to invoke your neighbor on your server'**
- **'my logical database must be realized as a physical database'.**

I do not understand why these persist. You would think that, after all these years of education about database concepts and SQL as well as all the input from DB2, Oracle and other databases (where so many of these concepts have been abstracted), there would not still be individuals who seem able to keep hold of only one view of something at a time. They do not seem able to cope with more than one view at a time — separating abstracted views from those which relate to real implementation. For some reason there are whole bunches of people who call themselves IT architects who just do not 'get' this.

The really seductive part of their appeal is that their rigid concepts and notions will usually work on a test bench or in a pilot. It is not until you try to scale that test or pilot implementation that anybody starts to ask questions — for example, why is each enquiry or transaction costing $5 to process or taking 5 minutes to deliver a response.

But still more frustrating is, when you demonstrate that all is not well, they come back with arguments like:

- **'it is too late to change things now'**
- **'for flexibility this is better'**
- **'I don't believe the optimizer does anything useful; in fact I think it usually optimizes incorrectly' (as if they know better than the developers in Oracle, IBM, Sybase and others who have been optimizing for years).**

This attitude is, I think, a close relative of the old 'I can write any piece of code in Assembler better than you can write it in a high level language'. This may be true in a lab but it is not helpful and certainly does not often produce a scalable solution.

#1 IT is a vocation

I refer to my number one indicator of being off-kilter as 'IT is my vocation'. This may be my peculiar way of saying; 'I am a messianic visionary capable of changing the world.' The symptoms are usually expressed in a form like 'I'm not really interested in this firm's problems; it just happens to pay me to do something I am interested in'.

The arresting aspect about this — at least to me as an IT curmudgeon with long experience — is that there will be, somewhere within this group of people, one or two who genuinely have that brilliant insight into what might be. But these are the exceptions. The difficulty is trying to work out which are the diamonds amidst the dross.

The problem is too many indifferent architects all believe that that they are the next Linus Pauling and too often are of the 'Einstein was wrong' disposition. We know that 99% of the time this is not true. Winnowing out the 1% is the hard work.

In my experience the IT architects who really make a difference are remarkably humble about what they are trying to achieve. They tend not to come along and say 'my role in life is to displace everything that already exists' — to start off with (it is the followers who do this later). With the person who really makes a difference, they are usually enamored of what they are doing on a small scale with relevant ideas just seeming to emanate. It is clear that they have particular reasons for their approach — and that they do not have grand world dominance plans. It just happens to be the right thing to do at the right time — and this is why it takes off and becomes a world-beater. This is in stark contrast to more frequently encountered IT people who seem to 'know' that the way to make something happen, is to state it often and loud enough that it becomes a de facto accepted truth.

So what indicates a good IT architect?

One of the aspects that marks out a good architect comes from the realization, usually from experience, that there are multiple different answers to any given problem. Yes, there are probably fringe cases where this is not true. But, in general, there are many answers to any given question. Which one you pick depends on a host of factors, not all of which are technical.

I was involved in a recent discussion about dividing up a Grid into a Web Service front end and a request processor. The general reaction in the room was 'why the heck would you make such a split, because it will nor perform as well as a single platform?' But that is precisely the point. The reason for such a division is to obtain control, security, revenue statistics and similar advantages. But these tend not to occur to absolutists who fixate on only one dimension.

As an IT architect, for me everything requires a balance. I keep being asked by people in technology towers 'is z/Linux a good thing or bad thing?', 'when should I use Intel rather than a mainframe?'. There is no right answer. It depends. What you going to use it for? How does it fit into a general plan? Everything lives in an IT ecosystem: is it pertinent to this? Can you make a decision which will allow you to live with more than one possible technology future? What will you trade off for having that insurance?

In one particular case I can think off z/Linux was well-suited, because it enabled the rapid deployment of some test systems rather than having to wait for physical boxes. But it might turn out to be bad to run it for 20 years in the same place — because you can run it more cheaply elsewhere. So we designed the application to be portable. IT architecture is all about balance. The best IT architects I know are the ones who understand that there are few absolutes.

On the other hand, good architects are not so 'laissez faire' that they fall into the other category which permits everybody to do whatever the heck they want — and expect it will all work out for the best in the end. They realize that they have to spend time and effort to make things more coherent, extensible and flexible.

I think there is substantial 'friction' in most of IT which means work is needed to overcome its effects. If it was really easy to change things, then an architect's life would be much easier. But in making decisions good architects are aware that they are building up the legacy systems of tomorrow. This is why virtualization is so powerful; it reduces friction, making change easier, but almost always comes at some cost. Consider an example: if you wrote everything in Java and Java was absolutely portable between different platforms and you picked the wrong platform, then you would be able to make the change easily. But the reality today is that virtualized interfaces are more expensive and even Java is not completely portable. Then there is testing and impact analysis, never mind what happens when you introduce external connections (to other systems or people).

What actually happens is that, in too many cases, you wall yourself into a situation where you are tied to the decisions you already made. Then, if nothing comes along to force you to change and move onto a more correct solution,

other people start building their applications on top of what you have created, coupling them to you. External dependencies have now been introduced. Even if you want to change you cannot — because they are dependent on what you built with all its 'features'. Good architects try to decouple decisions, once again reducing IT friction.

A good IT architect needs to be able to think abstractly about problems, about what will happen in a few years time when the next architect comes along and asks 'why did what idiot choose this?'. The difficulty is that new facts arrive, which change one's analysis. The ability, therefore, to envision the big picture and to run through the stack, all the way to understanding the concerns of the person — who is going to have to implement and administer and cope with a solution at 2am in the morning when it has all gone wrong — is what differentiates the excellent IT architects from the indifferent one.

Management conclusion

The IT architect occupies a challenging role in most IT organizations, with commensurate responsibilities. As Mr. Richards points out, the best ones are quite different from the indifferent or poor ones: they have an ability to see, explain and then facilitate the delivery of increasingly complex concepts. If you have these as your IT architects you have good fortune.

Then there are the various categories (of IT architect pointers) which Mr. Richards excoriates. Unfortunately his examples and his indicators will be all too familiar to many people — in IT or general management. As such his descriptions are valuable — because they enable others, whether in IT or management roles — to recognize those who might not be delivering value but who are instead producing obstructions (however well argued) rather than ways forward.

Testing distributed IT systems

Peter Bye
Senior Systems Architect
Unisys Systems & Technology

Management introduction

As should be expected, software testing has received a massive amount of attention over the years. Much work has been done on test tools, programming languages, architectural structures and methodologies — with a view to ensuring that software systems are not only tested but are also testable in principle, in the sense that it can be shown that they have been verified. There are companies which specialize in verifying software systems of various kinds — for example, in systems used in strictly real-time environments such as the avionics and power station control systems mentioned above.

In this analysis Peter Bye takes a look at some aspects of testing, with a special emphasis on distributed systems using middleware of various kinds. He argues that the complexity of systems increases very rapidly — particularly when these involve different technologies and widespread physical distribution. He describes and suggests strategies and approaches for minimizing adverse consequences.

Setting the scene

How to test IT systems adequately has been a concern since the origins of programming. IT system failure costs huge amounts every year. Figures released during 2005 by The Standish Group, which has long taken a special interest in the consequences of IT system failure, make alarming reading (Source: The Economist quoting Standish, 11 June 2005):

- **in 2004, only 29% of projects were successful — defined as on time, within budget and meeting all requirements (in 2002, 34% were successful, so we appear to be going backwards)**
- **cost over-runs averaged 56% of the original budgets**
- **projects, on average, took 84% more time than originally scheduled.**

Not all of these failures were necessarily caused by software failure following inadequate testing. Poor and changing requirements specification, and other factors, no doubt accounted for a proportion. Yet it is safe to assume that software failure, either functional or environmental, for example performance or scalability, causes a significant proportion of the project difficulties.

The above figures are concerned with failure to meet all objectives in a software project. It is, of course, possible that failures will occur even after a project has been completed and the resulting system is in operation, even if the project was apparently one of the 29% completed successfully.

Furthermore, the consequences of failure of an operational system do not stop with financial costs, significant though these may be where a good example is the cost of downtime in financial market systems. In other environments, lives may be at stake: failures in avionics, hospitals, chemical plant and nuclear power station control software can each have catastrophic consequences. Just consider the increasing complexity of modern systems.

Levels of complexity

You can think of an IT system as comprising a minimum of two levels:

- **users, or rather their associated equipment**
- **a central system containing an application of some kind, which provides the services consumed by the users, together with storage subsystems.**

Examples include:

- **interactive systems — such as bank account management and airline reservations — where users are equipped with terminals or workstations of some kind**
- **process control systems — where 'users' can be equipment which monitors state and manage industrial processes, as well as interacting with supervisors with workstations.**

Such systems have long been built, for example using a mainframe for the application and terminals (replaced later by PCs) as the end users. Disk and tape subsystems provided the necessary storage.

The software in the central system would typically include:

- **the operating system**
- **communications software**
- **transaction monitor**
- **database manager**
- **as well as the application logic itself.**

Depending on how the application was constructed, the software would be more or less cleanly layered. A well-structured system would separate the presentation of data to end users from the business logic of the applications, making the business logic device independent and so allowing new access channels to be added more easily. Similarly, the database logic would be separated from the application. However, all would be contained within a single environment, like that shown in Figure 2.1.

The architecture shown in Figure 2.1 persisted for some time and is still to be found. Three particular factors in system architecture have resulted in increased complexity, when compared with this basic structure.

The various logical layers — presentation and so on — are now more often spread over physical layers, for example communications interface and presentation, business logic and database access. You can think of this as vertical distribution.

The infrastructure may also be horizontally distributed, often in the interests of performance and resilience. The number of servers in the presentation layer may be increased, with load balancing used to distribute the incoming traffic across servers. Similar horizontal distribution for scaling and resilience purposes may be used in the business logic layer and also the database layer, perhaps using something like Oracle's RAC. Horizontal scaling of

this kind presumes careful management of any, and all, state information.

Finally, the application may be horizontally distributed, with different sets of business logic applied in different systems collaborating to deliver services to the consumers. Such distribution may be extended to partners in other organizations. In this latter instance you can think, for example, of a company selling products on the Internet. Various internal systems might be involved, for example handing inventory, accounts and scheduling. Then there would be complementary connections to external systems, for example credit card validation and transportation of the goods for delivery to the customer.

It is highly probable that each of these systems will use different technologies from each other. In addition, they are likely to be of different ages and states of development. You can refer to this kind of horizontal distribution as a composite system, as illustrated in Figure 2.2.

What are the factors that make testing distributed systems more complicated, often more complicated than people expect? This question is examined in the next section.

The effects of distribution on testing

The following are some of the factors to be considered when testing distributed systems:

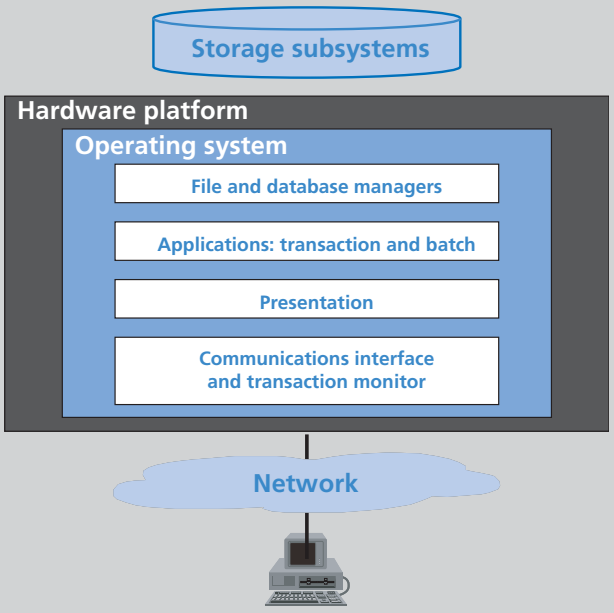
- **complexity increases rapidly as additional component systems are included, especially with horizontally distributed applications**
- **different technologies are likely to be used in the different components parts**
- **multiple organizations, either within the same overall enterprise or across others, may be involved**
- **networks link all the component parts together and therefore become a more important factor in testing than once was the case.**

Consider each of the above in turn. The configuration shown in Figure 2.2 is clearly more complex than that shown in Figure 2.1. However, it does not take many systems to result in a great deal of extra complexity. Experience shows that the increase in complexity seems to vary exponentially with the number of systems. This is particularly true of horizontally distributed applications, a point worth examining in more detail.

Horizontal distribution of applications means that services delivered to a consumer are implemented by a co-operation among a number of different application service components. What happens if one of the service components is not available?

The effect depends on the degree of importance in the delivery of the consumer service. It may be, for example,

Figure 2.1: Centralized, 2-tier application



that the unavailable component is not immediately critical and that its services may be invoked later. An example could be sending a frequent flyer number to a customer loyalty system as part of a check-in service. If all the components are essential to delivering the service, the system must be designed to minimize the probability of loss. This complicates the overall environment and the process of testing.

The above is complicated further by the use of different technologies. For example, in a vertically distributed system:

- **Windows may be used in the presentation layer with a Web interface**
- **there might be a J2EE application server with an EJB container, running under UNIX or Linux, in the business logic layer.**

Composite applications add more to this complexity. It is highly probable that different technologies will be used in the different application service components.

Where composite applications span organizational boundaries — either within the same enterprise or between different enterprises — new dimensions of complexity arise. Some of these are political, in the sense that the necessary agreements may be hard to agree (never mind set up).

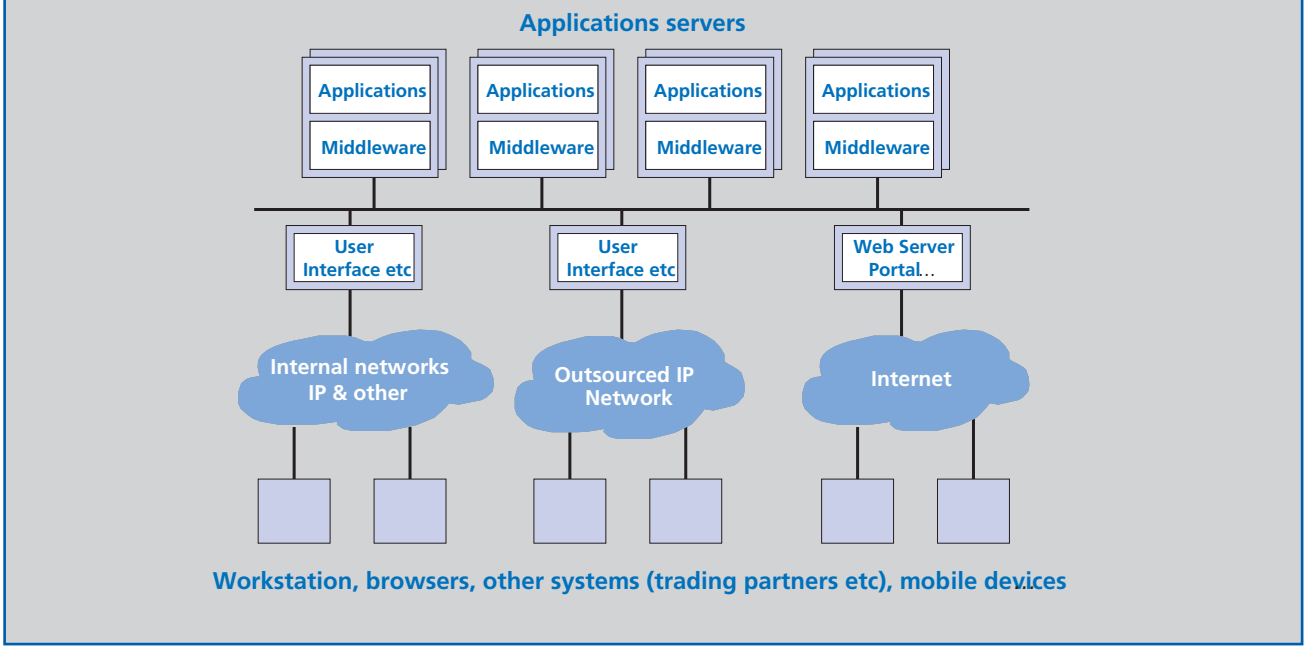
Just obtaining an agreed test facility, time and co-operating resources, for example, is not often straightforward. Testing distributed systems without having close collaboration between groups at each 'end' of a test can be intensely frustrating. Turning problems around can be protracted.

The final complication in the list above is the role of the network. In the configuration shown in Figure 2.1, the network connects the users to the system and is obviously a factor in testing end user interaction — for example, in measuring response time. However, all the other components are within the same platform, so interactions among them are internal software invocations of various kinds.

Vertical distribution introduces a network component into the application, as the layers use a network to communicate. Horizontal distribution for performance and resilience reasons adds some further complications in the form of failover and load balancing. Composite applications add still further network dependencies, possibly extending across the wide area.

Since these networks may be shared by traffic from a variety of sources, not just the collaborating components of a distributed application, there is a major potential for the network behaviour to affect factors such as performance. For example, a variable load not directly associated with a distributed application may cause performance to vary, making performance testing difficult.

Figure 2.2: Composite systems, using varities of differing technologies



The above factors combine to make the testing of distributed systems demanding. If all goes well and there are no problems, testing is always more complex because there are more cases to test.

As if this was not enough, there are almost always problems, which have to be isolated and resolved. Distributed environments, especially composite applications, pose technical problems, for example finding tools able to correlate information across different technologies. They also raise and pose organizational problems, in that the scope for blame attribution is broad.

Given that distributed systems are reality, and must therefore be tested, what strategies can be followed to ensure the best results? There are two areas where attention should be directed:

- **pre-implementation design, to minimize testing problems**
- **post-implementation tooling, to ensure that testing is as effective as possible and that problems are isolated.**

Pre-implementation: design strategies

A well-designed system, with a firm architectural foundation, is easier to test than a system assembled in an ad hoc manner. This is true for systems following the pattern shown in Figure 2.1. It is even truer for distributed environments, most especially those with composite applications.

The recommended architectural approach for designing composite applications currently centers on the notions of service orientation and service orientated architecture. The latter frequently employ the idea of an Enterprise Service Bus (ESB) linking the various components together.

The various services are regarded as loosely coupled, in the sense that the implementation details of each service are not visible externally, the services are generated independently of each other and individual services may be replaced with new versions as long as all the functions are maintained. All that is known to consumers is that the service performs certain functions, which are invoked using specified protocols. The leading industry standards are, today, built around Web Services — using SOAP, WSDL, UDDI and so on.

Figure 2.3 is a high-level schematic of this kind of architecture. The services expose an interface — for example Web Services — to enable the various functions to be invoked. The services themselves are constructed from software components of some kind. These could be many services, objects or whatever substructure is appropriate.

The purpose of an ESB is to tie all the pieces together and, in so doing, it may provide a number of features, including:

- **interfaces to a set of core, standard protocols — for example, SOAP**
- **management and monitoring capabilities**
- **adapters to work with other recognized protocols — as well as development tools for generating adapters — in order to accommodate a variety of systems, both old and new**
- **orchestration tools to enable services to be combined to execute complex business processes.**

There is considerable literature about how service-orientated architectures work and the benefits they bring. From the purely testing point of view, they permit the various services to be tested as 'black boxes', which may then be assembled or orchestrated to provide larger services in support of business processes.

This approach is really the latest in a long line of developments where software systems are built by assembling components, each of which may be independently tested and verified. The expectation is that this will ease the process of testing and verifying the resulting system.

Developing such an architecture is the first level of design. I have argued elsewhere (for example in the analysis written with Andy Roles: Designing for performance, **MIDDLEWARESPECTRA**, Volume 18, Report 3, in August 2004) that non-functional requirements should be considered at the earliest possible opportunity, when the initial architecture is being designed. Logical testability — and whatever else is required to ensure that a system can effectively be tested —, should be included with performance and other non-functional requirements, such as scalability, manageability and reliability.

Figure 2.4 shows a schematic of the flow of activity in a typical project. The activities in the shaded box — application architecture, technical architecture and integration design — follow the definition of functional and environmental requirements.

Architects must ensure that the functional and non-functional requirements are consistent with each other. There may need to be several iterations — around the architecture and design stages — before arriving at an optimum result.

Requirements may also have to be revised if the design shows that there are mutually incompatible needs. These activities take on a particular importance in composite applications, where new service implementations have to collaborate with existing systems, accessed through adapters, and possibly needing to work across organizational boundaries.

Post-implementation testing

As individual components of a system are developed, they will of course (or should be) tested at a unit level by the programmers concerned. I do not propose to discuss development tools and unit test strategies. I will focus on post-implementation testing. Testing following the unit level includes:

- **integration testing, where the various components of the systems are tested together; every system should be tested for compliance with its functional and non-functional requirements, for example performance**
- **transition to operations; tests at this level are those required to accept the system as fit for purpose, including having supporting operational and management procedures in place**
- **regression testing; systems change, with features added and deleted which means that appropriate integration testing needs to ensure that any unchanged elements continue to function: the modified system then needs to pass through the transition to operation for the new release.**

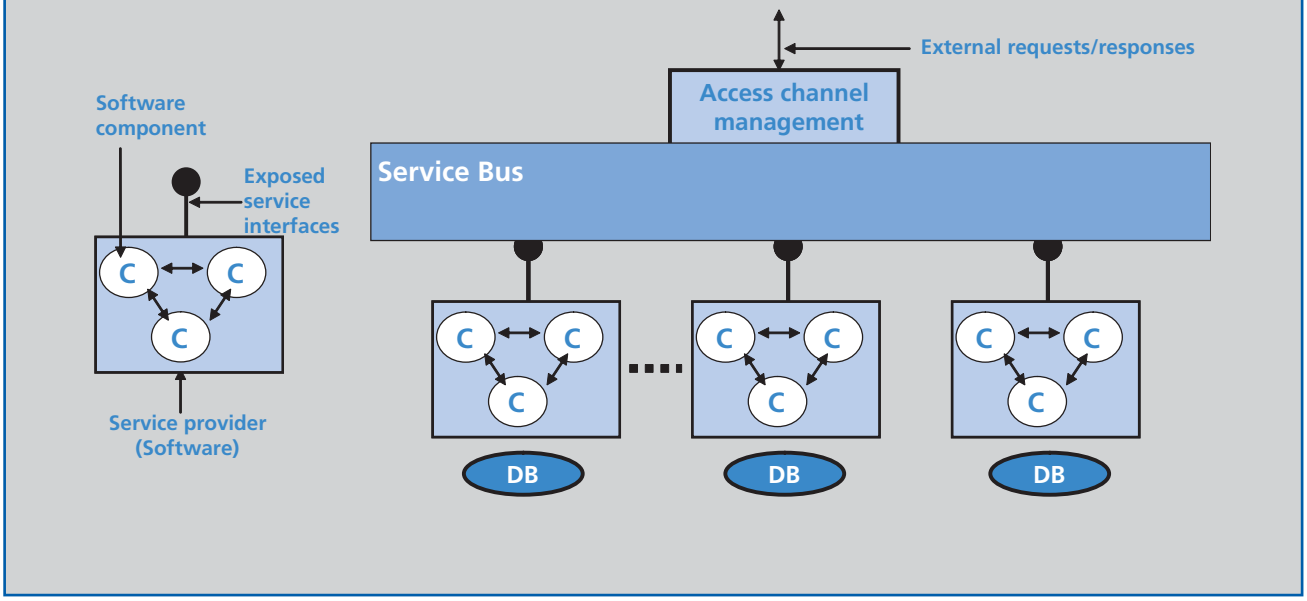
These activities can be complicated and require more work than most anticipate (or wish to anticipate). I would like to stress what I believe to be two important sets of tools for testing at these levels:

- **automation and the consequent requirement for tools to provide it**
- **the need for tools to monitor and measure what is happening in the various component parts, and to correlate information gathered across multiple component systems in a distributed environment.**

While tests can be conducted without automation — for example, by having a team of people enter requests into the system and record the results which are then compared with what is expected — automation brings many advantages:

- **automated functional tests are far quicker to execute and they are also more reliable, in that software is comparing what happens with**

Figure 2.3: Service Bus architecture



what is expected and can record all the results in a database for subsequent analysis, using a variety of analytical tools

- **labor can be used more effectively: time spent developing automated tests is an investment for the future, as the tests can be run many times (time spent executing tests manually must be spent again every time)**
- **automated tests are endlessly repeatable — this not only saves on labor, as mentioned above but repeatability is essential for technical reasons, if only to force discovery of problems**
- **performance tests for capacity and stability reasons are not really possible without automation, as least not without incurring great expense.**

There are various tools on the market, which can be used in differing combinations. What is true is that one tool is not always sufficient. The tools are sometimes part of products providing development environments, for example Eclipse and Rational. Other tools are products come from specialty vendors, such as Mercury. That said, simple test tools can sometimes be easily developed to test specific parts of a system, for example lower-level services invoked as part of a process.

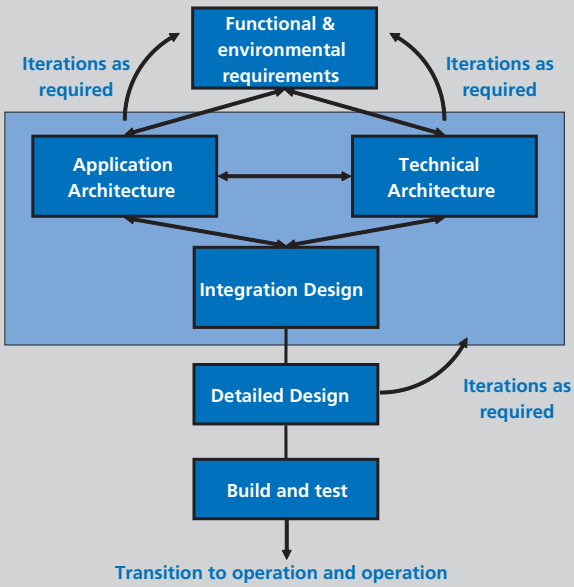
Automated testing can also be used to verify both functional and environmental aspects of a system. They may show that, in almost all cases, a system will not perform as required. For example, response times may not meet the required criteria. The average times may be wrong or, even if the averages are acceptable, peak load spikes in response time which may cause undesirable difficulties for users.

Finding the root cause of problems in a distributed system, especially those involving composite applications spread across number of systems providing different services, is almost always difficult — for the following reasons.

- **the different application technologies likely to be in use — for example, J2EE, .NET and existing wrapped applications — will be running under different operating systems and with different databases**
- **some technologies have an alarming variety of tuning options, leading to the rise of 'application tweakers' — specialists in tuning particular environments**
- **collecting the necessary information to find the cause of problems may be difficult; correlating it can be even more of a burden.**

Nevertheless, it is essential that information about the behavior of each component part be gathered. This can, in part, be statistical — in the sense that certain loads (for example, processor or memory use) can be recorded as

Figure 2.4: A typical development flow



averages over short time intervals which can then be aggregated over longer periods.

There is also a need to be able to trace all events related to processing specific requests. This requires logs containing message content, response times and — essential in distributed environments — a means of identifying messages so that they can be traced around the system.

The information gathered this way is valuable, but is complicated and time consuming to correlate. There are tools on the market which provide effective measurement and correlation. Some are aimed at specific environments — such as Java. One example is Wily Introscope.

Others are aimed at a combination of environments, including .NET, Java and various databases. An example is i3 from Symantec.

The advantage of these tools is that they provide the correlation required and help to identify the root causes of problems. They are in general not seriously intrusive, consuming only small amounts of resource in the systems being monitored.

A final word on the subject of tools. It is rare that one (tool) will be sufficient. In most cases, a combination of tools will be required to suit the particular conditions of the system under test.

An example of good practice

This example comes from an application service provider for the banking sector. The company develops the applications and operates them on behalf of a number of small and medium sized banks.

The company decided to test the viability of using J2EE as an application environment for an account management application. The major advantages expected included speed of development as well as portability across different operating systems. A proof of concept project was, therefore, initiated to develop an application and to test it in different environments.

A great deal of attention was paid to testability and tooling. Some 80,000 lines of Java were written, of which more than half of which were for test purposes. Suitable tooling was chosen to provide information about the system behavior, enabling the cause of problems to be traced and corrected. In particular, automation was heavily deployed (and the following details indicate just how extensively and effectively):

- **there were 567 functional test cases, executed by an automation tool, which checked and logged all the results; depending on the speed of the network connection to the system, the tests could be executed in as little as 3 to 4 minutes**
- **there were 21 application server test cases, 45 acceptance test cases, 10 performance test scenarios and one integrity test**
- **98% of the application business logic tests were automated.**

Management conclusion

In an analysis of this length, it is only possible for Mr. Bye to scratch the surface of a subject as extensive as testing distributed systems. The problems raised by the complexity of distributed environments, especially the increasing number of composite applications, are daunting. Multiple different environments — operating systems, application environments, databases and so on — pose significant difficulties in finding the root cause of each problem encountered during testing.

As always in IT, there is no single magic solution. However, a combination of approaches can help, both at the pre- and post-implementation stages. At the pre-implementation level, testability should be included in the requirements and considered at the design stage and the architecture should be chosen to be such that the structure is as far as possible testable. The use of service concepts can help.

At the post implementation stage, automation should be used to the greatest possible extent; it is the most effective use of labor and the only realistic way to execute performance tests (there are various suitable tools available on the market). Particular attention should be paid monitoring the system and gathering information about the behavior of each component; correlating the information for root cause analysis can be difficult and the various powerful tools that are available should be used. It is unlikely that any one tool will be sufficient; a combination is more realistic, although the number should be kept to the minimum consistent with quality results.

The example of the banking application shows what can be achieved, at least in that specific environment. The resulting application met expectations, in large part because of the attention paid to the points above.

The Windows Communication Foundation (alias Indigo)

Tom Welsh
Consultant

Management Introduction

At last Windows Vista (the operating system previously known as Longhorn) has reached beta, and we can look forward to a general rollout by late 2006. One of the biggest and most challenging projects Microsoft has ever undertaken, Vista promises to bring radical changes to the way software is developed, managed and used. This is particularly true of one its major subsystems, the Windows Communication Foundation (WCF), long familiar to developers as project 'Indigo'.

In WCF Microsoft has rationalized .NET's support for Web Services, message queuing, COM+ and much more — replacing five different communication/middleware models with a single integrated one. In addition, a coherent, integrated set of Web Service standards has been implemented, making interoperation between Windows and other platforms much easier as well as more future-proof.

In this analysis, Tom Welsh:

- ***offers a brief introduction to WCF's purpose, structure and features***
- ***considers whether Vista is an Enterprise Service Bus, or its equivalent.***

Longhorn, Indigo, Avalon and Whidbey

Longhorn — the next in a long line of past Windows versions — has been distinguished by:

- **its sheer ambition**
- **the time it is taking to deliver.**

The timescale for its delivery has been a story of continual slippage. Indeed, such have been the many delays, and so many of the components have been dropped or postponed, that some sections of the media began ironically calling it 'Shorthorn'.

In October 2003 Eric Rudder, Microsoft's senior vice president of Servers and Tools, told Computerworld that "[we] absolutely will deliver Yukon, the next version of SQL Server, and Whidbey, the next version of Visual Studio, shortly". Those two releases are currently scheduled for the end of 2005, over two years after Rudder made that commitment. But it looks like being worth the wait.

Even the product's name is a source of confusion. Since its inception, it was known by the codename 'Longhorn'. Now two separate beta versions are available:

- **the client operating system has been formally dubbed 'Windows Vista'**
- **the server operating system is still called 'Microsoft Windows Server Longhorn'.**

So whether you talk about Vista or Longhorn depends on whether you are referring to the client or the server. For simplicity's sake I shall stick to Vista in this article, to cover both.

Vista and accompanying products and interfaces

Vista is accompanied by a number of important new products and application programmer interfaces (APIs), including:

- **Windows Presentation Foundation (WPF) (alias Avalon)**
- **Windows Communication Foundation (WCF) (alias Indigo)**
- **Windows Storage Foundation (WSF) (alias WinFS)**
- **Windows Workflow Foundation (WWF)**
- **Windows .NET Framework Extension (WinFX)**
- **Visual Studio 2005 (alias Whidbey)**
- **SQL Server 2005 (alias Yukon).**

Windows XP already comprises around 40 million lines of code. Vista — which is even bigger — necessitated a complete rewrite.

Between them WPF, WCF, WSF and WWF impose drastic changes in some of the operating system's most important functions, such as the user interface, storage, and middleware. Microsoft has taken the opportunity to tidy up and improve these subsystems, building in new features and support for some of the latest industry standards.

WinFX is the new API for Vista, superseding both the .NET Framework and the older Win32 interface. Unlike the .NET Framework, it is an all-managed API — meaning that all code using it will run under the control of the .NET Common Language Runtime (CLR).

Visual Studio 2005 can be used to build applications that invoke the .NET Framework. But, under Windows Vista, developers will have to migrate to WinFX. As Visual Studio 2005 is not forwards compatible, applications built with it will not run on older versions of the .NET Framework (1.0 and 1.1).

Thus the move to Windows Vista will be one-way. It may also be quite expensive, as users will have to adopt the new operating system, WinFX, Visual Studio 2005, and everything that goes with them.

What does WCF do?

WCF's goals are ambitious and far-reaching. According to Microsoft it will radically simplify the way in which distributed systems are built, moving:

- **away from the traditional distinction between clients and servers**
- **towards peer to peer networking, where any computer can request services of any other.**

WCF has three architectural design goals:

- **built-in support for a broad set of Web Services protocols**
- **implicit use of SOA (Service Oriented Architecture) and support for, principles**
- **a single API for building connected systems.**

Microsoft's official definition of WCF is rather lengthy, but only because it is so rich in new features. "[WCF] is Microsoft's unified programming model for building service-oriented applications with managed code. It extends the .NET Framework to enable developers to build secure,

reliable, transacted Web Services that integrate across platforms and interoperate with existing investments. [WCF] combines and extends the capabilities of existing Microsoft distributed systems technologies, including Enterprise Services, System.Messaging, .NET Remoting, ASMX, and WSE to deliver a unified development experience across multiple axes, including distance (cross-process, cross-machine, cross-subnet, cross-intranet, cross-Internet), topologies (farms, fire-walled, content-routed, dynamic), hosts (ASP.NET, EXE, Avalon, Windows Forms, NT Service, COM+), protocols (TCP, HTTP, cross-process, custom), and security models (SAML, Kerberos, X509, username/password, custom)."

The reference to "secure, reliable, transacted Web Services" derives from the title of a joint IBM-Microsoft paper of that name, which was published in October 2003. WCF implements the WS-* series of specifications in order to provide end to end security (even across the trust boundaries of separate organizations) and quality of service (QoS). The latter refers to the more intangible qualities of a network link, such as bandwidth, latency, reliability, and — just as critically — the bounds within which these are likely to vary.

Although its main aim is to deliver "secure, reliable, transacted" Web Services, much more is expected of WCF. As well as unifying the programming model that developers use for writing distributed applications, it will also unify the approach that operators and administrators take to deployment, configuration, monitoring and troubleshooting.

Microsoft has shrewdly taken the opportunity to build in 'deep support' for service instrumentation and configuration, much as a well-organized municipal authority might be ready to install sensors and piping when a road is dug up. This should help operations staff to keep a finger on the pulse of WCF applications, checking for:

- **QoS indicators**
- **incipient problems**
- **traffic levels.**

WCF extends the .NET Framework 2.0 (currently in beta), and is said to be the first programming model built from the ground up to provide implicit SOA development. It is available only as an extension to .NET, and consists mainly of a set of classes that run under the control of the .NET Common Language Runtime (CLR).

Microsoft explains that WCF merges the features and strengths of no fewer than five of today's Windows 'technologies':

- **ASP.NET Web Service (ASMX)**
- **Web Service Enhancements (WSE)**
- **.NET Enterprise Services**
- **.NET Remoting**
- **System.Messaging.**

ASMX

ASMX is Microsoft's abbreviation for an ASP.NET Web Service. Until now, ASMX has been the obvious default choice for building Web services that must interoperate with other systems that are not built on .NET. This is because it does not rely on any proprietary Microsoft techniques, and does not incorporate any of the more advanced WS-* specifications.

WSE

Provided as an optional supplement to .NET, WSE has been Microsoft's way of letting developers try out early implementations of those WS-* specifications that could not be fitted into .NET itself. In this respect it plays a similar role to the software packages that IBM offers through its alphaWorks Web site.

WSE 2.0 supports WS-Addressing, WS-Security and most of the other specifications layered on the latter. Like IBM's alphaWorks packages, though, WSE was essentially a tactical stopgap but Microsoft has committed to making WCF compatible with it, so developers can safely go on using it until they are ready to cut across to WCF itself.

.NET Enterprise Services

.NET Enterprise Services is the region of .NET where COM+ lives on under the covers, permitting its powerful capabilities to be employed by any .NET application. In the temporary absence of any Web Service specifications for distributed transactions and other related QoS features, Enterprise Services was the natural (and almost the only) way of adding distributed transaction capability to ASMX Web Services. As well as transactions, an impressive range of benefits accrued:

- **automatic host start-up**
- **idle time management**
- **instance management and scalability**
- **concurrency management**
- **granular security**
- **loosely coupled events**
- **many useful administrative functions.**

Unfortunately, because of its dependency on COM+, this solution was only feasible within the .NET environment, so interoperability with other types of system were ruled out. (Another limitation was that developers could not add their own custom services and filters).

.NET Remoting

.NET Remoting is a relatively obscure communications technique which combines some of the strengths of Web Services and COM+. Compared to Web Services, it offers faster and more efficient operation through optional features such as binary encoding and the use of raw TCP/IP instead of HTTP.

There are also extensive opportunities to customize the default 'pipeline' by adding extra filters. On the other hand, these features are not (yet) interoperable beyond the world of .NET — and they entail extra programming complexity. Perhaps Microsoft has said so little about .NET Remoting because it did not want to distract attention from its increasing support for Web Services.

System.Messaging

System.Messaging is the rather technical-sounding name that denotes one of the many .NET 'namespaces' and the functionality which it provides. Like many other such terms, it has become the short hand way of referring to a subset of .NET features — in this case queued messaging (along the lines of Microsoft Message Queuing (MSMQ) and IBM's WebSphere MQ).

In fact, System.Messaging can map to MSMQ or to asynchronous, one-way Web Services. MSMQ has many advantages — for instance performance, scalability, reliability and security — but it is not directly interoperable with non-Microsoft systems (although gateways are available).

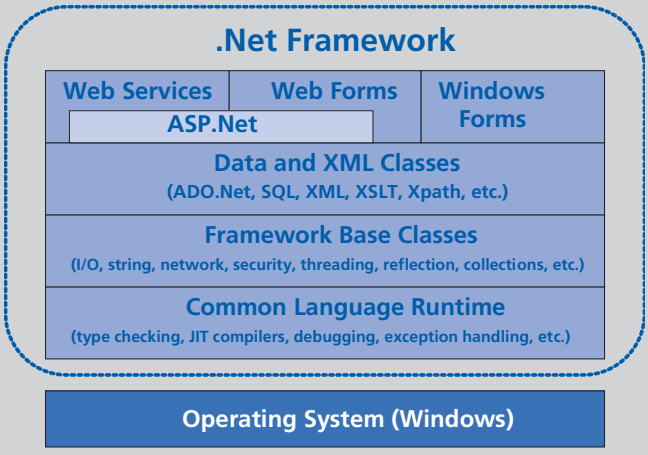
WCF — advantage or drawback?

Whereas each of these five alternative approaches has its own advantages and drawbacks, WCF combines the advantages while eliminating most of the drawbacks. In a classic example of what programmers call 'refactoring', Microsoft has reviewed the experience gained in the first five years of Web Services deployment.

In the light of that experience, it has redesigned the part of .NET that handles Web Services to make it cleaner, better integrated and easier to use. Above all, WCF's facilities are far more orthogonal than those of .NET; that is, it becomes more nearly possible to 'use everything with everything'. Another benefit is that WCF reflects and implements the many far-reaching changes in Web Service specifications that have taken place since 2000.

In a nutshell, WCF should:

Figure 3.1: DCF — bindings decision tree (Source: Microsoft)



- **be simpler to learn, be easier to develop with and be interoperable with any other applications that support the WS-I profiles, like ASMX**
- **provide an open, extensible, highly customizable architecture, like that found in .NET Remoting**
- **support distributed transactions, granular security, concurrency management, asynchronous disconnected calls, reliable messaging, publish/subscribe, scalability and management — like that found in Enterprise Services**
- **implement a coherent set of compatible yet orthogonal Web Service specifications, including many of the latest updates and additions —for example, WSE**
- **offer queued (asynchronous) messaging as an alternative to synchronous Web Service invocation, in System.Messaging.**

Choice and simplifying the developer's task

Without having to look beyond WCF, developers will be able to create applications that interoperate within the same computer, across computers within .NET or between .NET and any other platforms that support Web Services. Out of the box, WCF offers a choice of invocation models, including:

- **conventional RPC, with blocking calls using lists of typed parameters**
- **asynchronous RPC, with non-blocking calls using lists of typed parameters**
- **conventional messaging, with non-blocking calls using a single message parameter**
- **message-based RPC, with blocking calls using a single message parameter.**

To simplify the programmer's task still further, WCF comes with a set of off-the-shelf bindings (Figure 3.2), each of which is configured with a frequently demanded options:

- **BasicProfileHttpBinding: this conforms to the WS-I Basic Profile 1.0, which specifies SOAP over HTTP (this has an end point's default binding if none is explicitly specified)**
- **BasicProfileHttpsBinding: this conforms to the WS-I Basic Security Profile 1.0, which specifies SOAP over HTTPS**
- **WsHttpBinding: this supports reliable message transfer with WS-Reliable Messaging, security with WS-Security and transactions with WS-Atomic Transaction; the binding enables interoperability with other Web Services implementations that also support these specifications**
- **WsDualHttpBinding: this is like WsHttpBinding but also supports interaction using duplex contracts; using this binding, both services and clients can receive and send messages**
- **NetTcpBinding; this sends binary-encoded SOAP, including support for reliable message transfer, security, and transactions, directly over TCP; this binding can only be used for WCF-to-WCF communication (because binary-encoded SOAP is still not widely supported by the industry at large)**
- **NetNamedPipeBinding: this sends binary-encoded SOAP over named pipes; again this binding is only usable for WCF-to-WCF communication between processes on the same Windows machine (because it relies on optimized local Inter-Process Communication)**
- **NetMsmqBinding: this sends binary-encoded SOAP over MSMQ; the binding can only be used for WCF-to-WCF communication (because MSMQ is a proprietary Microsoft product — although gateways are available to other products, such as IBM's WebSphere MQ).**

Web Services support

As for Web Service specifications, WCF's first release will support:

- **WS-Addressing**
- **WS-Policy**
- **WS-Metadata Exchange**
- **WS-Reliable Messaging**
- **WS-Security**
- **WS-Trust**
- **WS-Secure Conversation**
- **WS-Atomic Transaction**
- **WS-Eventing**
- **the SOAP Message Transmission Optimization Mechanism (MTOM).**

With the exception of the two last named, which did not appear until later, these are very much the same specifications that are cited in the "Secure, Reliable, Transacted Web Services" declaration. In other words, WCF is a first step towards delivering on those promises.

The structure of WCF

As WCF consists of a set of changes and extensions to the

.NET Framework, it makes sense to begin with a brief overview of the latter. Conceptually, as Figure 3.1 illustrates, the .NET Framework is quite simple — the hallmark of nearly all good design. It comprises just two fundamental parts:

- **the Common Language Runtime (CLR)**
- **the .NET Framework Class Library.**

Other aspects of .NET — such as ADO.NET, ASP.NET, Windows Forms, Web Forms, Remoting and interoperation with COM+ — are implemented by the Class Library. The latter is truly gigantic, and contains re-usable code for so many standard system and application functions that the source code of .NET applications is often surprisingly compact.

The CLR is the engine for preparing, running and managing .NET programs. Everything else in Figure 3.2 belongs to the Class Libraries. These unify and supersede predecessors such as Microsoft Foundation Classes (MFC) and Windows Foundation Classes (WFC), and are available to programs written in any supported .NET language.

WCF has relatively little impact on the lower two layers of Figure 3.3 (the CLR and the Framework Base Classes). Instead, it mainly affects the upper two levels — especially the top one. Figure 3.3, which pulls together some of WCF's most important features, should not be compared directly with Figure 3.2, as it represents a different abstraction of the .NET Framework.

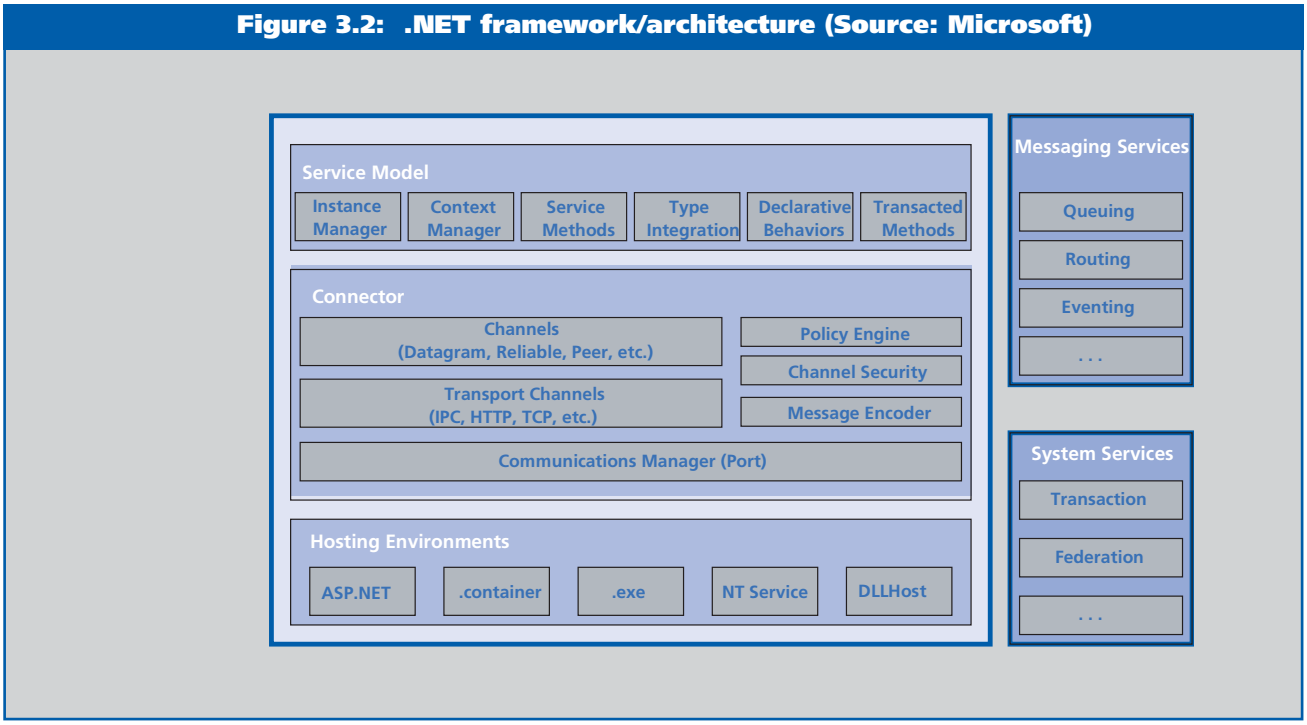
The major subsystems of WCF (the main block in Figure 3.2) are:

- **the service model**
- **the connector**
- **the hosting environments**
- **the associated messaging and system services.**

The purpose of the service model is to facilitate SOA development in any .NET language — whether C#, C++, Visual Basic, Fujitsu NetCOBOL or any of 20 or so others. Incoming messages are assigned to ordinary code routines or methods through the [ServiceMethod] attribute. Instance and context management are also provided for each service. Declarative behaviors and transacted methods automate security, reliability and transactional characteristics as methods enter and leave a service.

The WCF connector is a layered input-output framework based on the SOAP standard, which helps developers to build messaging applications independent of transport, target platform or language. The two main entities within the connector are ports and channels:

Figure 3.2: .NET framework/architecture (Source: Microsoft)



- **a port represents a service end point as a network location (for instance, as a URL)**
- **a channel is a symbolic device that services use to send and receive messages.**

There are also:

- **a message encoder, that allows special XML message encodings to be added to the system**
- **a policy engine**
- **support for secure messaging.**

WCF can be run from any of the hosting environments shown in the bottom layer. For instance, it can be run in a Web context through ASP.NET, as an NT service, or as a DLL host in COM+, as an executable or as a container.

The two boxes at the right-hand side of Figure 3.2 represent the system and messaging services that can be accessed through WCF. These include identity federation (optionally by means of WS-Federation) and transactions (optionally through WS-AtomicTransaction). WCF also provides a set of messaging abstractions such as queues, events and content-based routing.

Is WCF an Enterprise Service Bus?

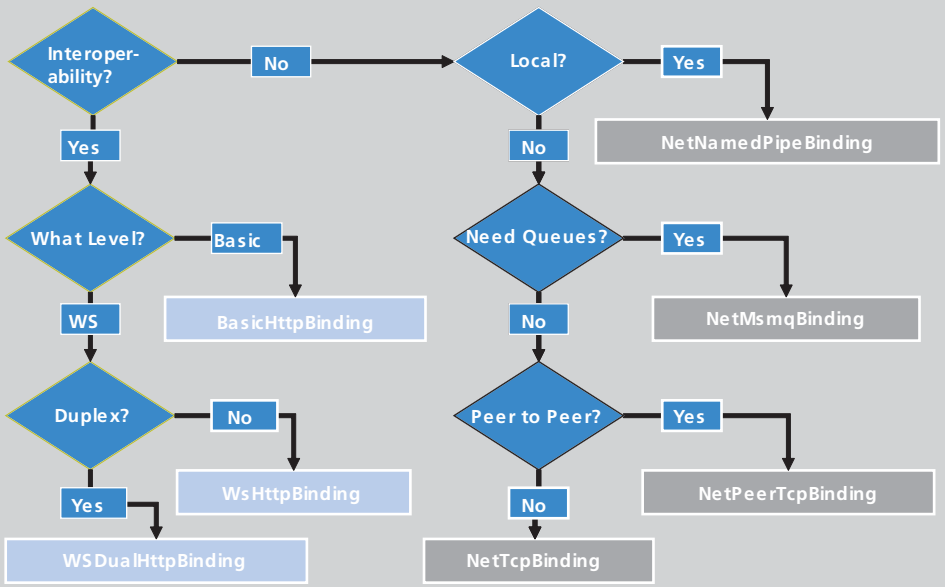
Gartner Group has suggested that WCF amounts to an ESB, since it supports:

- **Web Services**
- **message oriented middleware (MOM)**
- **intelligent routing**
- **a distributed peer to peer architecture (rather than one built round a hub).**

But Ronald Schmelzer of ZapThink, an analyst firm specializing in SOA, disagrees flatly. In his opinion: "...the ESB term is being used in increasingly more vague and unhelpful ways. For example, Microsoft's Indigo has been described as an ESB. However, Indigo, which forms the basis of Microsoft's next-generation development environment combining SOA as well as advanced code management and code portability capabilities, is not by itself any more an ESB than any other development or runtime environment".

Sonic Software's David Chappell, a leading authority on ESBs, also dismisses any similarity between WCF and an ESB. "What they are building is a message bus with Web Services extensions," he explains. "This is very different from an Enterprise Service Bus".

Figure 3.3: WCF architecture (Source: Microsoft)



What is the difference, then? According to Chappell, a message bus requires the writing of low-level code, whereas "the mantra of the ESB is configuration rather than coding". He does concede that Microsoft's BizTalk Server, combined with WCF, could approach the status of an ESB, but maintains that "BizTalk is still a hub and spoke integration broker".

Management conclusion

Like the rest of the software enhancements that are set to accompany Windows Vista, WCF looks like being a valuable step forward. Windows itself, the .NET Framework and Visual Studio have all been redesigned and rewritten — making them more efficient and easier to use, while updating them to comply with the latest standards. The only pitfall left appears to be delivery: even the best ideas can fail if poorly implemented. However, that seems unlikely, as Microsoft has thrown its immense resources — including the personal attention of chairman Bill Gates — behind the Vista program.

WCF cannot be properly evaluated in isolation. It is a major component of the new WinFX API, which will become the standard interface for programming Windows as soon as Vista ships. In its own right, however, it represents a big improvement in Microsoft's support for building efficient, reliable distributed systems.

WCF merges five different programming models into one. It encourages developers to conform to accepted SOA principles and supports ten of the best-established industry standards for Web Services. It also simplifies the creation of true peer to peer networks, and makes it easier for Windows applications to interoperate with those built on other platforms, such as Java.

It may not quite fit the technical definition of many of the self-proclaimed 'interested parties' of an ESB. But it is a pretty safe bet that there will be many developers who will be using WCF to deliver ESB-type function long, long after they have forgotten that there was ever anything called an ESB.

Linux server virtualization: implications for middleware

Mark Lillycrop
Principal Analyst
Arcati

Management introduction

From humble origins in 1991, as Linus Torvalds' personal UNIX implementation, Linux has taken the IT world by storm. This open source operating system accounts for a very large proportion of the Web server and hosting market, and is now experiencing huge growth in print/file server applications, e-mail and firewall systems as well as numerous distributed office roles. Figures from analysts IDC late last year showed that Linux server revenues had passed the $1B, following nine consecutive quarter of double-digit revenue growth.

In this analysis, Mark Lillycrop examines what Linux is achieving. He then moves on to consider what are the implications for middleware as more and more Linux server virtualization is introduced.

Linux success in the enterprise

There are many factors driving the success of Linux in the server environment. These vary from near-universal support for the operating system to the wide availability of mainstream applications. All have contributed to its success within the corporate environment in recent years.

There are also 'political' attractions, as Linux (ostensibly at least) avoids the lock-in involved with more proprietary solutions, particularly Windows. As a result many regional and national government departments in the USA, UK, France, Germany, Finland and others have gone for Linux in a big way, as have a number of Mid-European and Far Eastern authorities (which in many cases are utilizing the operating system to support green-field applications).

From a financial point of view, businesses have rapidly latched onto the cost benefits of Linux server applications. Though far from 'free', Linux does provide many opportunities to rationalize and streamline software costs within the enterprise, and the growing competitiveness of the Open Source marketplace is ensuring that these costs stay relatively low.

Managing Linux in the data center

For all its popularity, there are a number of issues with Linux (and open source software in general). These have often acted to restrict its infiltration into the enterprise data center. Despite its rapid progress, Linux still has some ground to make up in terms of overall performance and availability, especially compared with the more robust UNIX implementations (in particular, Solaris, AIX, HP-UX, etc.). Additionally, Linux still lacks scalability in terms of the number of processors it can support, although the situation in this respect constantly continues to improve.

Perhaps the most significant issue, however, is manageability. We often talk, in the IT industry, of pendulum swings between centralized and distributed technologies. Linux has experienced its own swing in microcosm over the last few years. In a great many cases, Linux systems have been acquired around the periphery of the organization, often to support new Web-based applications with little or no direct involvement from central IT.

Only recently InformationWeek editor Larry Greenemeier provided an analysis of some recent research into the way that Open Source software is used within the enterprise. He concluded:

"Most large, multibillion-dollar companies don't know how much open source they're actually using. It's introduced into the IT environment by developers looking to build the best applications in the shortest amount of time possible.

"Most companies don't have a budget, per se, for open source. Open source is often used to help launch side projects that otherwise would stay on the shelf because there isn't enough IT money to go around.

"Perhaps the greatest driver of open source adoption [seems to be] that programmers like it."

This is a scary scenario. It suggests that one of the biggest growth trends within corporate IT is taking place outside the formal organizational and financial structure of the business (but that also happened with the PC and, arguably, the mini-computer). Like the previous wave of Windows-based client/server systems, Linux applications that have been acquired and implemented around the enterprise now need to be located, analyzed, consolidated and managed centrally.

Most CIOs are, of course, well aware of this problem. Consequently there is a growing enthusiasm for the various Linux server consolidation offerings on the market which, to a greater or lesser extent, enable virtualization and tight control of software resources.

Linux is not the only player

Linux may be leading the way in this area but it is only one of several operating platforms that are moving inexorably towards virtualization as users begin to appreciate the technical and cost advantages of maintaining a pool of compute power and storage that can be applied to individual applications as required. This is, after all, simply a different manifestation of the same trend that has led to grid computing and other on-demand facilities.

Like grids, server consolidation (be it on Linux, Windows, z/VM or whatever) has considerable implications for the way that companies will implement middleware to support the new environment. The way that applications interact with each other, access data, and communicate with other networked resources is gradually evolving to meet the needs of a virtualized world.

Approaches to Linux server consolidation

The 'heavyweight' solution to Linux consolidation is undoubtedly the mainframe. IBM's VM hypervisor operating system, originally introduced way back in 1972, has

been given a new lease of life as a way of consolidating multiple Linux workloads alongside traditional mainframe applications, sharing data and system resources as required. z/VM, as it is now known, can technically host thousands of individual Linux server applications as system images, although the practical limit is more likely to be in the hundreds.

The great advantage of the VM environment is its mature management capabilities. The hypervisor has proved extremely adept at managing the middleware, data sharing and security requirements of hosted Linux images.

As shown in Figure 4.1, there are several distinct ways to configure Linux applications on the mainframe. The first two (top left and center) show the software running on the bare metal:

- **the first without logical partitions (never a very bright idea), is no longer possible on the latest models**
- **the second with a separate copy of Linux running in each LPAR (Logical PARtioning facility, the capability offered by IBM).**

IBM's latest z9 hardware will support up to 60 LPARs. But this is still not an ideal solution for truly large-scale Linux consolidation.

z/VM also provides the opportunity to support huge numbers of Linux images, either in a dedicated configuration (top right) or as part of a mixed OS environment, sharing system resources with traditional mainframe workloads running under z/OS and OS/390. (Note that, in the lower configuration, the Linux and z/OS workloads are running in separate LPARs, but each z/VM LPAR can support hundreds of Linux images.)

The IBM middleware perspective

From a middleware perspective, z/VM provides a unique environment:

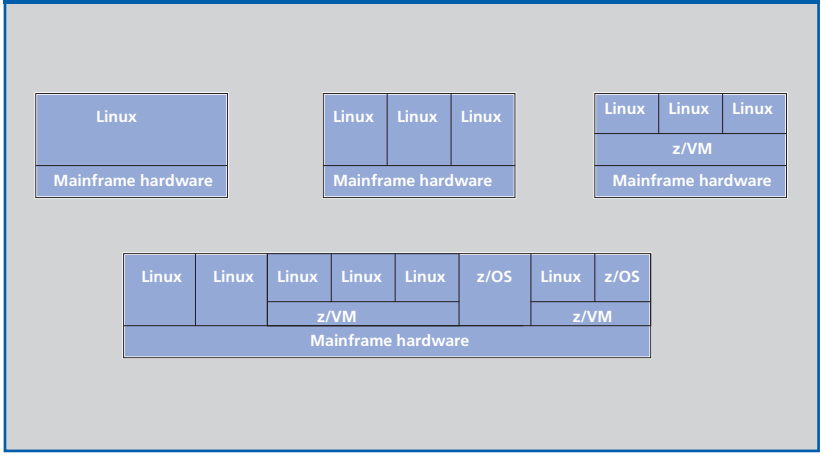
- **processing, communications, I/O, and memory are all shared and tightly managed**
- **the high-speed access to mainframe sub-systems — such as CICS, DB2, IMS, etc. — are equally available to Linux and to other operating systems running on the shared processor.**

Most users working in such an environment will have a mature combination of middleware in place, with products such as WebSphere MQ and WebSphere Application Server. These provide an integration backbone between the mainframe and others distributed platforms.

Such a backbone allows consolidated Linux applications fully to exploit a broad range of industry-standard interfaces, like the MQI, SOAP and JMS. They are also likely to be exploiting CICS and IMS Gateways, DB2 Connect, and other such products which can, thereby, enable Linux application to interact with traditional z/OS-based systems and applications — at near-memory speeds.

z/VM also provides a wide choice of security options. Passwords and authentication can be centrally managed via the hypervisor itself, although many users opt for a Linux-based solution using standards such as the Secure Shell. This automates the management of security across the multiple images.

Figure 4.1: z/VM virtualization



Intel-based server virtualization

Although the conventional mainframe may offer a Rolls Royce solution for Linux server consolidation, there are many reasons why this route will not appeal to everyone. Unless organizations have an existing investment in S/390 applications and staff skill-sets, they are more likely to consider virtualization on an Intel-based platform. This is a much more diverse marketplace and, though far less scalable at the top end, it offers users a greater range of options which are often significantly less expensive.

The most mature offerings on the Intel platform are:

- **the VMware server products**
- **Virtuozzo from SW Soft**
- **Microsoft's Virtual Server.**

VMware has the largest market share within the corporate virtualization market space. SW Soft's business has grown out of the commercial hosting and service provider sector. But its focus on performance optimization has attracted the attention of many internal data centers.

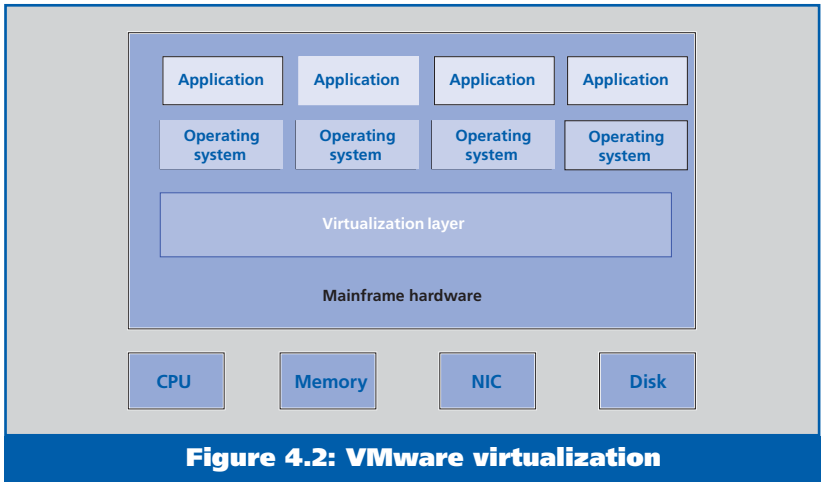


Figure 4.2: VMware virtualization

As shown in Figures 4.2 and 4.3, these two products take different approaches to virtualization. The VMware (and the Microsoft one is similar) products take the virtualization down to the hardware level, supporting multiple operating system instances on one server.

In some ways, this approach is similar to the mainframe LPAR, but it lacks the sophistication of the mainframe resource sharing and performance optimization. To support multiple OSs on the Intel chips, the VMware products have to perform hardware trapping, which has an inevitable effect on performance.

SW Soft's sales proposition with Virtuozzo is its 'virtual private server', based on a level of virtualization software that sits on top of the operating system. SW Soft claims that its solution is significantly more scalable and resource-efficient than other Intel-based solutions, as it reduces the amount of hardware access and uses a single copy of the operating system.

It also boasts 'smart configurations', which allow administrators to:

- **support (potentially) hundreds of virtual servers**
- **move these around between physical servers depending on the relative criticality of different applications.**

One of the main problems is that the x86 chip-set is simply not designed to be virtualized. This stands in contrast to IBM's:

- **modern mainframe hardware architecture — which has been designed from the outset to accommodate multiple operating systems and mixed workloads**
- **POWER processor, which is rapidly evolving to support virtualization.**

VMware, SW Soft, and Virtual Server have adopted various approaches to compensate for the performance overhead imposed by the Intel and AMID platforms. Inevitably there are trade-offs and drawbacks to each of these.

Overcoming Intel limitations and other options

As usual, when a technical limitation of this kind is encountered, the industry has risen to the occasion and come up with multiple other solutions. One that is currently attracting a great deal of attention is the Open Source Xen hypervisor from Cambridge University. This offers 'paravirtualization' for the Intel platform.

Xen claims to be more efficient than existing offerings, and manages the Intel processor in a way much closer to the z/VM model. The downside is that it requires the guest operating system to be modified and ported to Xen — which involves considerably more effort up-front.

This is no problem for vendors like IBM, HP, Novell and even Sun, who are lining up behind this approach. But it does currently restrict the technology to Linux and UNIX as Windows cannot be ported in the same way. To support an unmodified OS requires a change in the hardware architecture, which is coming in due course with VT-enabled processors. In the meantime, though, Xen offers an extremely promising virtualization offering for Linux applications.

There are other options for those considering virtualization. Look at Cassatt and Virtual Iron (the latter running over multiple processors).

What becomes apparent, though, is that this is still a relatively immature market, with many vendors vying to provide the most elegant solution but all wrestling with technical problems that are likely drastically to reduce the benefit of consolidating the distributed applications in the first place.

'It's all in the middleware'

There are, thus, clearly many choices available to customers who seek the financial and architectural benefits of Linux application consolidation. But, as with similar developments in grid and on-demand computing, one size will never fit all.

Depending on the performance requirements of individual applications, the way they access and share data and the characteristics of other applications 'in the mix', the suitability of VM, VMware, Xen or any other solution will vary considerably. In other words, application virtualization ultimately needs to be managed at a higher level, by the middleware that connects the whole environment together.

Over the next few years, users need (and probably will) see a new generation of middleware emerging, one that has the ability to:

- **manage application processes on the fly**
- **route calls and services between participating applications and processors depending on specific requirements.**

This middleware will address issues such as:

- **allocation and process management**
- **security management**
- **data management**
- **systems management.**

Allocation and process management is concerned with:

- **how 'processes' are defined, in terms of the start and finish of individual business functions**
- **how the parts of the process are allocated to processors, virtual processors or other hardware/OS sub-divisions**
- **how the relationships between these constituent parts are managed**
- **how resources are allocated (according to business rules and service level requirements, application priority and capacity availability).**

Security management needs to consider at what level should security be managed (at the hypervisor level or by hosted OSs or by applications, for example). Then there will be the issues of how authentication problems or alerts be handled and how security will be monitored around the periphery of the virtualized environment.

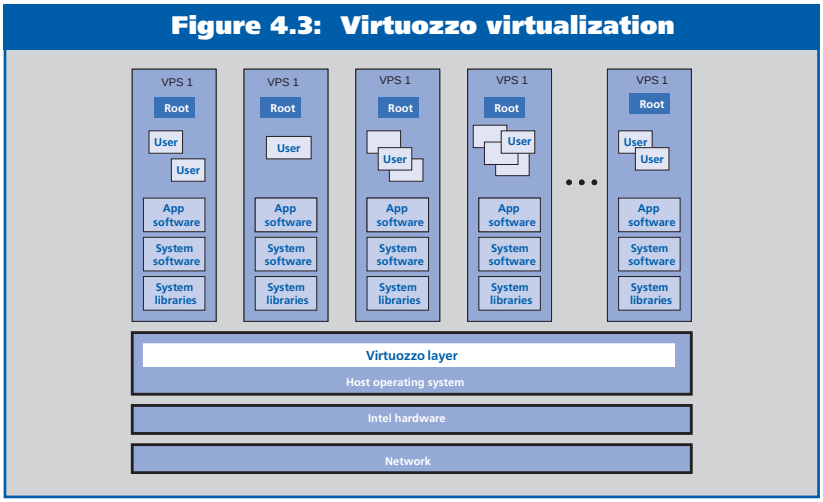
Data management will need to address:

- **how database access is to be managed**
- **how data integrity will be guaranteed**
- **how commits are managed across participating processors and applications**
- **what criteria should be used to determine the optimal storage location of data, in terms of cost and performance.**

System management will involve examination of:

- **how much of the management of this environment will be automated**
- **where and when should operator intervention be involved.**

Figure 4.3: Virtuozzo virtualization



All of these issues, and more, will need to be addressed in a fully-functional virtualized IT environment. The required middleware components will be critical and come from a number of sources.

The position today

Some of these functions are already performed within traditional IT environments, and the management tools surrounding products such as Tivoli, WebSphere, DB2 and Oracle. These are already capable of a

large amount of on-the-fly coordination within a distributed environment.

Linux server consolidation, however, occupies a middleground between enterprise IT management and Grid management. The middleware needed to manage the virtualized environment will need to incorporate elements of high-performance centralized system management on the one hand with specialized Grid middleware technology on the other.

For this to happen, there will need to be new standards that allow management products on both sides of the wall to work together. Grid technologies are notoriously immature in areas such as transaction processing and commercial database management. The industry has been waiting for the experience and expertise gained in scientific projects by organizations — such as GLOBUS — to be used and proven in more mainstream deployments.

One ray of hope in this respect is Univa, the organization that emerged out of GLOBUS at the end of last year. This has the express intention of developing, packaging and supporting Grid-based technologies (effectively middleware) for supporting mission-critical business applications.

The recently announced alliance between Univa and IBM is evidence of the pivotal role that organizations of this kind are likely to play in the not too distant future. Though still some way from recognizing and optimizing interoperability between every flavor of server virtualization on the market, Univa is clearly occupying a strategic position in bringing together the essential middleware parts of the whole equation.

Management conclusion

Clearly Linux server virtualization is just one component of a much broader on-demand IT environment that is emerging within and between today's large and medium-sized enterprises. This style of dynamic resource allocation is a far cry from the way that applications and transactions have been handled in most businesses in the past. The drivers for this style of IT provisioning may come from changes in the way that businesses choose to acquire their computing power rather than from any innate enthusiasm for on-demand facilities from within the industry itself.

There can be little doubt, though, that server virtualization is here to stay and that its emergence has profound implications for middleware. Indeed, the very diversity of standards and optimization techniques that are to be found among the virtualization providers today underlines the essential role that middleware will continue to play, even in a world of increased server virtualization. And traditional middleware, as Mr. Lillycrop describes, does not disappear either, for it will play a key role in connecting all those interconnecting all those virtualized systems.

Service-oriented Integration — SOI

Dr Keith Jones
IBM Software Solutions Worldwide

Management introduction

Integration continues to be a major expense item for many enterprises as companies are acquired, application packages are purchased and new software is developed. The topic is made all the more challenging and complex by the size and heterogeneity of modern IT systems.

Well formed methodologies and technologies have evolved over time to provide structure to integration projects and proven deployment strategies. The advent of the service-oriented approach to building systems is enabling architects to devise new ways of constructing software and integrating existing systems.

In this analysis Keith Jones:

- ***reviews the application of service-oriented thinking to integration in enterprise scenarios***
- ***describes the most common solution patterns that might be used***
- ***identifies the value of support for SOI in emerging middleware tools and technologies.***

The problem

Integration is the work that must be done to enable applications to work together. In most cases applications that have been integrated were, nevertheless, not originally designed to work with each other and many of these were written by different software companies. Delivering integration work of this kind is an enormous item in most enterprise IT budgets — second only perhaps to the ongoing cost of application maintenance.

The need for integration comes from many and varied quarters:

- **as enterprises grow through acquisition and mergers, applications must be integrated into existing IT systems**
- **as enterprises decide to build or buy to satisfy business needs, applications must be integrated into existing IT systems**
- **as purchased application suites and infrastructures evolve, new components must be integrated using appropriate new technology options as these become available.**

The trend in building modern IT systems to use heterogeneous platforms and application components is undeniable. Yet the underlying principles for integration are simple enough:

- **the re-use of existing business logic/data wherever they meet business goals**
- **the introduction of integration technologies that are both cost-effective and provide the best strategic solution.**

There are many points of integration that may be considered for typical applications. Each has its advantages and disadvantages. For example (Figure 5.1), integration:

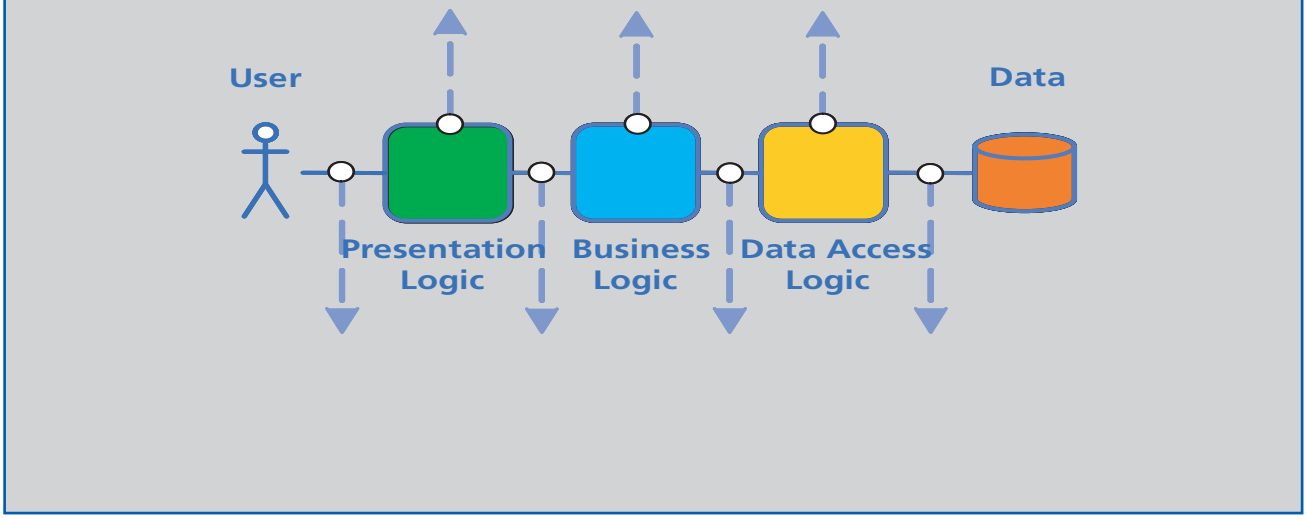
- **at the user interface using browser or portal technologies can provide a consistent look and feel but may not hide the lack of integration of the data sources**
- **integration of data sources may provide consistent access to statement-of-record data but expose issues in business processes that are irreconcilable.**

The reality is that integration projects usually involve long lists of interactions, each with their own functional and non-functional requirements and status in the strategic outlook. And it is the long list of costly work items that makes integration such an expensive proposition that leads architects to search for better alternatives.

Solution patterns

The first simplifying assumption is that most integration

Figure 5.1: Points of integration



projects can be broken down into the work needed to enable a number of pair-wise interactions between applications. A well integrated solution is characterized by applications exchanging information between interacting 'end points' in a relatively seamless and efficient manner (Figure 5.2). The challenge is then to find the best technology solution for each of those interactions.

As a device for use in the analysis of integration requirements it is possible to think of all the pair-wise interactions in a matrix (Figure 5.2) from integration point to integration point. Additionally, the requirement for integration at each of the end points may be captured separately as a set of:

- **functional characteristics: for example, business operation, information model, protocol, etc.**
- **non-functional characteristics: typically this includes availability, performance, scaling, security, integrity, usability and manageability.**

A pattern may then be identified for each and every intersection in the matrix that requires an integration solution. That pattern must potentially satisfy all the functional and non-functional requirements for integration at that intersection.

For some large integration projects this step greatly reduces the complexity of the work needed. A relatively small number of patterns are often found to satisfy a broad range of the integration capabilities needed.

The patterns that might apply to enterprise integration scenarios are numerous. However, the most commonly used patterns fall into four categories or families:

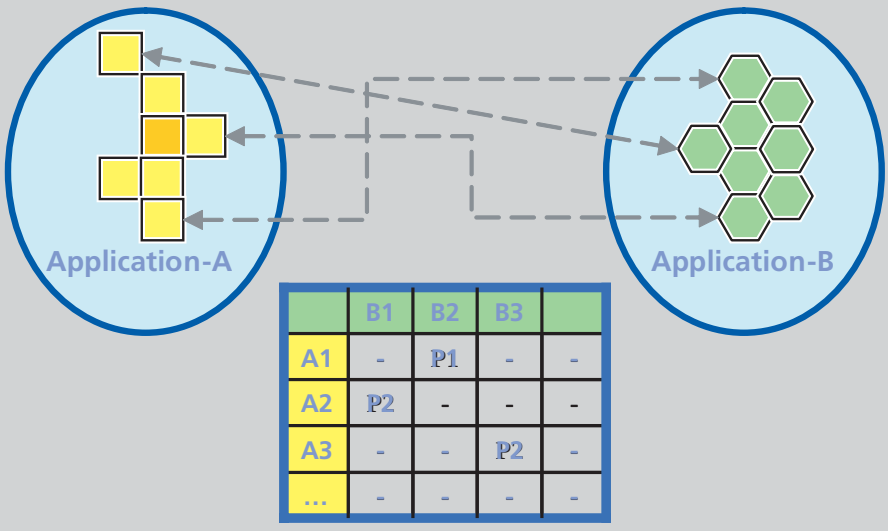
- **'no' connection patterns (for example, extract-transform-load)**
- **'direct' connection patterns (TCP/IP sockets)**
- **'mediated' connection patterns (for example, adapter, gateway, message broker, service bus)**
- **'symbolic' connection patterns (such as publish and subscribe, or pub/sub).**

The best middleware infrastructure products support a wide range of these patterns. They are 'better' because they provide multiple options for realization and deployment

Choosing a solution

The 'stack' of related integration patterns (Figure 5.3) that has been developed over many years to solve an ever

Figure 5.2: Interacting integration end points



widening range of integration problems often presents multiple different, possible solutions for specific use in application interactions. For example, a TCP/IP direct connection, a MOM mediated-connection and an event-based symbolic connection might all satisfy specific requirements. The challenge is to choose the most appropriate solution for each situation.

Many factors come into play when making such a choice. Re-use, or the extension of, existing middleware infrastructure is often a dominant factor — not least because as skills, investments and operational procedures are already established. But there are other factors that must also be considered, depending upon whether the end points to be integrated are new applications or existing ones.

One way to assess the advantages and disadvantages of a particular pattern and its realizations when using available technologies is to assess the changes needed (if any) to application logic to effect the integration. The more changes that are needed, the more costly the integration may become to implement, test and deploy.

The integration pattern stack (Figure 5.3) suggests that there is both a vertical and a horizontal dimension to the way in which integration points interact. The fundamental purpose of each interaction is the exchange of data whether it be command data, business data, event data or some combination of different types.

No-connection patterns

At the lowest level data is exchanged between application components using files or memory buffers. This is often the best performing option because it involves:

- **no data transformation**
- **a simple exchange protocol**
- **local operating system services.**

This is, however, the least flexible of all the integration patterns. The applications are extremely tightly coupled and it is difficult to reconfigure these applications or to further re-use their business value.

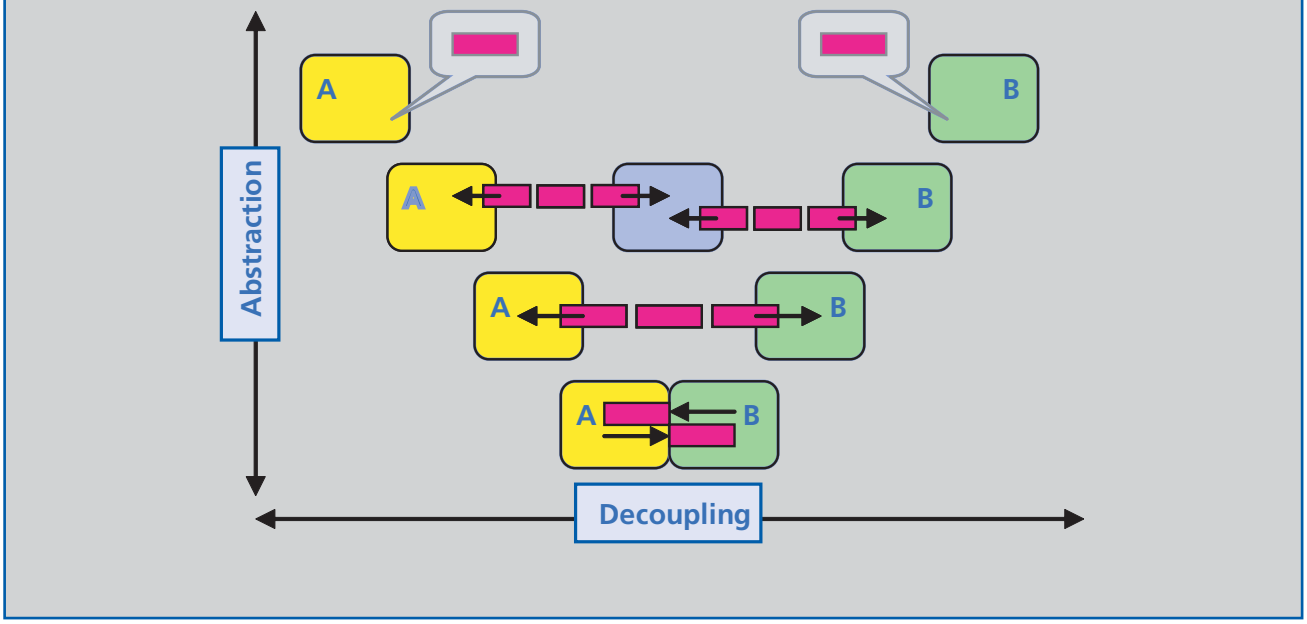
Direct-connection patterns

At the next level in the stack, applications exchange data using a direct connection. This allows for integration points to reside on different servers when standard networking protocols are used — such as TCP/IP or FTP.

The application servers may be co-located in a data center or they may be connected across enterprise and geographical boundaries. However, both applications must be:

- **available at the same time**
- **capable of handling the same formats in order to exchange data.**

Figure 5.3: Integration pattern 'stack'



Mediated-connection patterns

The mediated-connection pattern at level 3 in the stack represents the richest set of options (Figure 5.4 shows some common variations). The adapter pattern is most often applicable when integration points have different characteristics that cannot be changed economically.

In a representative adapter scenario, packaged application suites — such as Siebel Systems' CRM — offer interfaces for data access. Take, then, an existing application which must be integrated even though it was designed for (or with) a different interface.

An adapter can be used to map data requests from one application [A] into compatible requests to be handled by the target application [B] and map data responses returned to complete interaction cycles. When the exchange protocol is simple the adapter is a relatively simple pattern to apply.

When the application protocol is sophisticated:

- **data transformation is required**
- **security protocols must be mapped**
- **transaction boundaries must be enforced.**

Inevitably the adapter becomes much more complex. However, it is sometimes useful to think about complex-adapters as though they are chains or flows of simpler adapters which each provide an aspect of the integration work needed.

Adapters that comply with the Java Connector Architecture (JCA) — such as SAP XI — are fast becoming the most common in the marketplace for integrating new business logic with existing information systems.

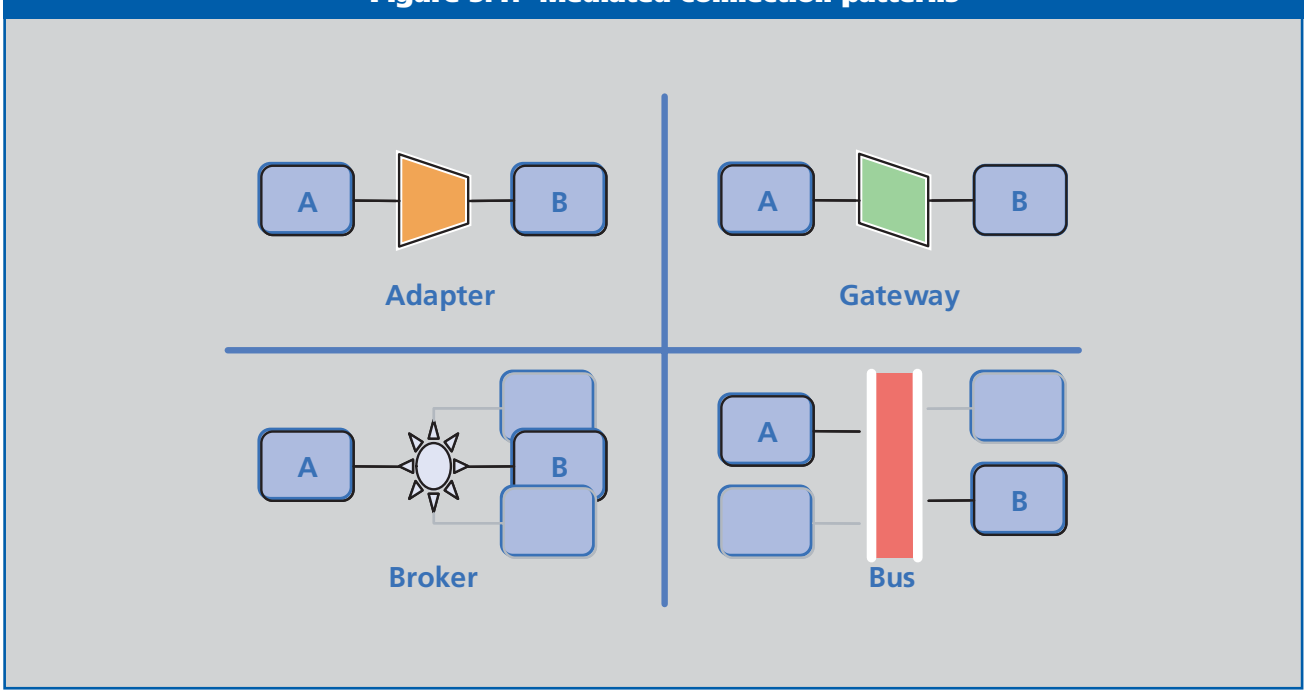
The gateway pattern

When the integration points exist in different organizations it is often advantageous to use the gateway pattern to provide the mediation necessary. The gateway between applications offers a control point for applying corporate policy standards, security guarantees and safeguards as well as the required data and protocol transformations.

The gateway pattern is:

- **often used to mask the existence of some application end points to un-authorized external users (as in the case of B2B scenarios)**
- **may also be used to concentrate the internal use and re-use of application end points for control and accounting purposes.**

Figure 5.4: Mediated-connection patterns



The broker pattern

When integration is best achieved by routing data to many different application end points within an enterprise, the broker pattern is often applied. This pattern simplifies business logic by removing knowledge of location (addressing) for particular end points — and may also be used for workload distribution.

The broker receives data requests from one application and routes them transparently to another. Which application handles a particular data request depends upon configuration data or upon the content of the request.

The broker pattern is often used when a large number of similar end points must be connected. In this case the broker acts as:

- **a hub for communications**
- **a control point for applying management policies**
- **a buffer against failures.**

The bus pattern

The most powerful mediated-connection pattern is the bus. Applications connect to the bus using standard interfaces — with an adapter or gateway, as necessary — to gain access to any other application similarly connected.

Logically the bus provides a sequence of mediated-connections in a chain between any two interacting application end points. These mediated-connections provide several degrees of isolation between the end points, which minimizes the need for costly change to business logic.

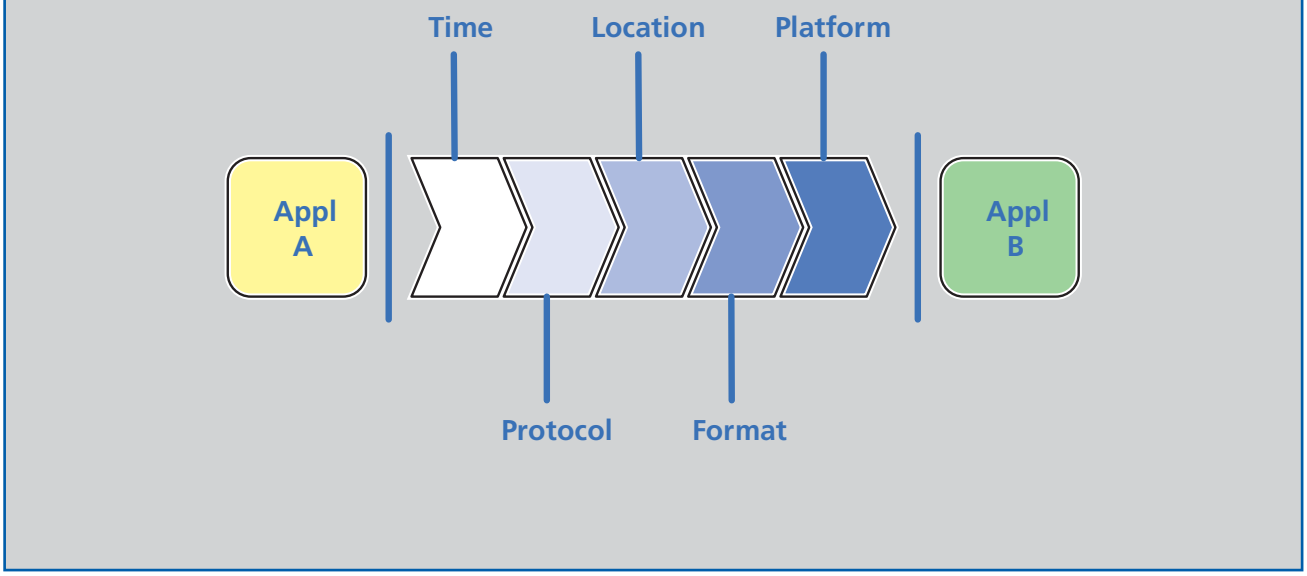
That isolation also provides flexibility to reconfigure an IT system for operability and optimization. But it may add to application performance costs. These costs can be mitigated in turn by caching and other techniques that are transparent to business logic.

Symbolic-connection patterns

Ultimately the purpose of integration is the sharing and processing of data. The symbolic connection pattern allows applications to publish data to symbolic destinations that are either queues or topics. Applications that need access to published data may subscribe to receive notifications when these become available.

The end points that use the symbolic connection pattern are not aware of each other and share only the semantics of the data published in the context of an enterprise scenario. However, the business logic at these end points must be constructed to publish and subscribe to business process (context) data as well as to data about business entities (content) in order to enforce guarantees of integrity.

Figure 5.5: Affinities between end points



Loose-coupling

This pattern offers additional degrees of separation between the application end points. This separation decouples the application logic being integrated by:

- **encapsulating the target end point**
- **removing built-in awareness of the realization or deployment details.**

Several dimensions of that coupling are illustrated in Figure 5.5. In general the more decoupled are the application end points, the greater is the potential for flexibility in the integrated solution. Each pattern offers some decoupling in one or more of these dimensions. The effects are cumulative (although not necessarily in the order shown).

For example, some direct connection patterns can be used to implement platform decoupling but cannot be used to implement the other dimensions. In contrast, the bus mediated-connection pattern can be used to implement decoupling in some or all dimensions to meet requirements.

Integration using services

The decoupling of application end points is a significant component in the horizontal dimension of the integration pattern stack (Figure 5.3). The corresponding vertical dimension of abstraction and virtualization is also important when choosing a solution for integration.

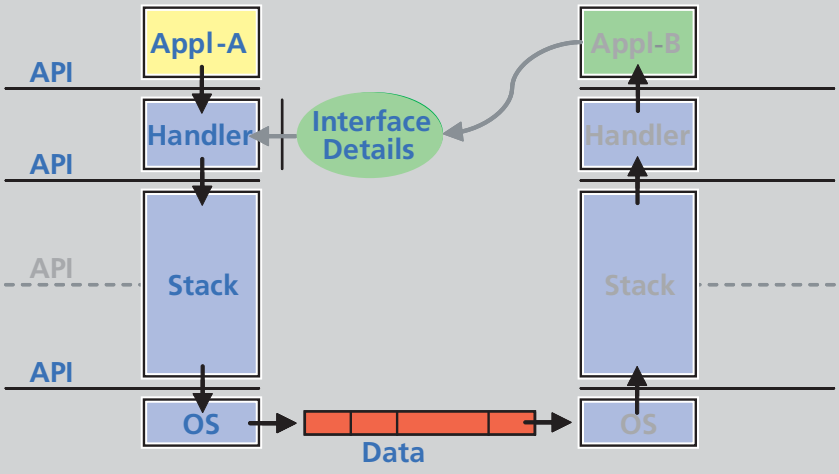
A number of significant middleware developments in recent years have enriched the options available for integration in this dimension. At each integration point there is a vertical stack (Figure 5.6) that, in most cases, corresponds at a minimum to some basic networking capability — for example, TCP/IP, IIOP, DCOM, etc. — for supporting the transfer of data between applications.

In more sophisticated scenarios a messaging capability (for example, as available with WebSphere MQ, the renamed MQSeries) is also included in the stack supporting the exchange of data as messages. At the top of the stack in the most advanced form of middleware (WebSphere Message Broker or Process Server are examples) there is also an option to include a service capability which supports the exchange of data between applications as service requests and responses.

The choice of API (and corresponding formats) to be used at each integration point is significant — because, at each layer below the chosen API, the middleware is providing functional value that:

- **results in less complicated business logic**
- **facilitates greater decoupling.**

Figure 5.6: The vertical stack



When an application uses a particular API, data and contextual information is passed to the stack for processing. The (prior) choice of platform, data format, location of the target end point, synchronicity and exchange protocol is usually:

- **most specific and concrete for APIs at the lower levels in the stack**
- **more virtual and abstract for APIs at higher levels.**

Various recent developments in the middleware technology stack have made it possible to simplify application logic further — by making it configurable with concrete details of integration end points at deployment time or even dynamically at run-time. Service APIs (JAXrpc), for example, may now be configured with end point location and protocol details at deployment or run time (as illustrated in Figure 5.6).

It is the handler component, as shown, which dynamically retrieves configuration details from the execution environment. This enriches an otherwise simplified API call made by the application logic [A] to exchange data.

At the target end point the corresponding handler component may dynamically enrich the contextual information passed to the application logic [B] together with the data. It may also provide correlation and other functions (depending upon which API has been chosen).

It is also worth noting that service APIs and interfaces may now be deployed for:

- **direct-connection integration patterns**
- **mediated-connection patterns**
- **symbolic-connection patterns (coming soon).**

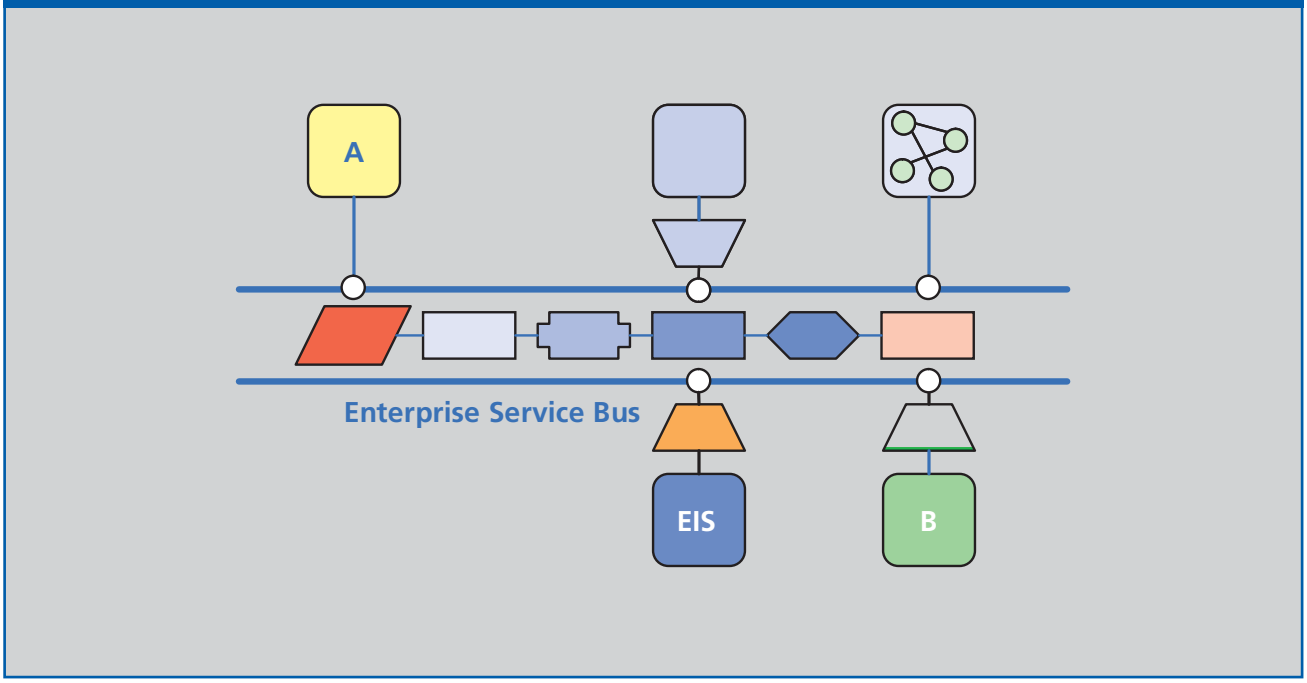
SOI

Service-oriented integration (SOI) is, therefore, not about one protocol or one pattern or one middleware technology. It embraces a much wider range of integration capabilities.

The essence of SOI lies in the choice that is made to encapsulate integration end points behind published open standard interfaces and to use standard service APIs to access those endpoints:

- **either explicitly in application logic (for the direct-connection and symbolic-connection patterns)**
- **or in adapter logic (for the mediated-connection patterns).**

Figure 5.7: Integration using as ESB



In the latter case the adapter presents a service interface on one side for integration purposes and a legacy interface on the other. An example of this — in the Java world — is the use of a service interface in front of a JCA adaptor that in turn encapsulates an Enterprise Information System (EIS). Application logic that must be integrated with such an EIS can now be designed to interact using the service interface.

Developing an SOI Solution

Once patterns have been chosen for interacting end points in an integration project, the technologies for realizing them must be selected to meet specific functional and non-functional requirements. The advantages of current middleware technologies over previous generations are evident in their support for the decoupling and abstraction of integration end points in application logic, as outlined in this analysis.

The advantages of the supporting tools, such as WebSphere Integration Developer, are evident in their support for open standards (such as WSDL) and service lifecycles. This includes the generation of end point configuration metadata and dynamic end point proxy handler code.

An additional advantage of some middleware technologies is their evolution to include service bus capabilities. This evolution has enabled integration projects to choose between file-based, simple transport-based, message-based, event-based and service-oriented capabilities for the patterns chosen.

The service bus pattern deserves special consideration (Figure 5.7) if the number of service end points is expected to grow in an enterprise integration scenario. The bus allows for large numbers of application components to be connected as service providers and consumers either directly or using appropriate adaptors. Service requests and responses are automatically routed by the bus based on end point locations and based on supported protocols registered in a service directory. The bus also provides a number of mediations that may filter, transform or enrich service requests and responses as required by corporate standard policies or the requirements of integration end points.

The service bus also provides a common infrastructure for new business logic as well as integration of existing application logic using service interfaces. The decoupling provided between end points — whether they are newly developed or adapted from existing applications — provides operational flexibility for the future and a framework for re-use of valuable software assets.

Management conclusion

Integration is the work that must be done to enable applications to work together. In most cases this means integrating existing applications that cannot be changed economically. Many patterns have been developed and proven over time to support a wide variety of enterprise integration needs.

The good news is that middleware infrastructures have evolved over time to provide realizations for those patterns based on a number of complementary technologies; files, databases, networks, messages, events and most recently services.

To some, SOI is about the top-down vision of building an enterprise that responds to change on-demand — by developing IT systems that are based on a service model that is evolving over time with business needs. But SOI can also be seen to apply bottom-up as integration projects are tackled and application end points are exposed as services — to be accessed using gateways or an Enterprise Service Bus. The same patterns may be used to effect integration of applications that is more loosely-coupled and more virtualized using service APIs and interfaces.

New middleware tools and technologies are emerging that extend existing infrastructures to support services for integration as well as top-down development of services for new application systems. Packaged applications are similarly evolving to provide service interfaces for key business functions.

Convergence upon the use of service technologies in these different domains is bringing new focus to integration projects. Critical existing application capabilities are being exposed as services and made available for reuse in wider enterprise SOA initiatives. As Dr. Jones describes, this raises the value of the work produced in integration projects and accelerates the development of the enterprise service model.

Thoughts on approaches for SOA/ESB applications: part II

Nick Denning, Chief Technology Officer
Strategic Thought

Management introduction

Service Oriented Architectures (SOA) — and the implementation of SOA technologies to create an Enterprise Service Bus (ESB) — promise much. With these it should be possible to:

- ***simplify the complexity and reduce the costs of development, operational management and overall ownership of a common infrastructure and the systems it supports***
- ***wrap existing business logic into common services which can be integrated into flexible processes, thereby improving the performance of the business and maximizing the return from existing assets***
- ***enable organizations to integrate — or detach — business units quicker than has hitherto been possible, particularly in the context of merger and acquisition***
- ***support integration with partners using electronic straight through processing (STP), increasing the ability of an organization to trade at increased levels with its customers, to be able to meet unexpected increases in orders from customers and in turn to trade with suppliers to meet customer demands***
- ***facilitate closer involvement of business users in the design, deployment, monitoring and re-engineer business processes***
- ***provide data to support Return on Investment (RoI) evaluation and business performance monitoring using run time metrics captured from electronic business processes; these metrics can subsequently be used to justify future and even current investments.***

However, SOA might deliver none of these things. It may simply become the next technology fad that consumes significant expenditure and only delivers the same applications back, with similar problems to those of the old applications. In this analysis, the second in a series, Nick Denning examines some of the common pitfalls and problems that he and Strategic Thought have encountered over the years (including in delivering SOA/ESB solutions to clients). He also adds comments about why an SOA or ESB is relevant to solving these problems. He outlines briefly the required SOA approach and in a subsequent MIDDLEWARESPECTRA he will describe in more detail a common software architecture that can exploit a consistent model, address relevant patterns and map applications styles to an architecture as part of a road map for embarking on SOA adoption into an enterprise.

Classic problems with traditional applications

The design and implementation of applications based on traditional methods pose a number of challenges which are often difficult to deal with. Too often we ignore these difficulties, and hope for the best.

Some of the common challenges are:

- **immature locking strategies, including poor handling of deadlocking when conflicts arise that permit data corruption through lost updates, or else block the system because locks are held for too long**
- **failing to bound transaction sizes, resulting in transactions being of indeterminate size and potentially becoming so large that row locks escalate to table lock and even block all other transactions (resulting in the system apparently 'hanging')**
- **refreshing screens via polling to maintain up to date data on user screens, which introduces excessive loads on systems even when no changes to the database are taking place**
- **generating dynamic SQL in an uncontrolled fashion, potentially generating excessive loads and blocking locks**
- **using a menu-based approaches which result in unauthorized or un-tested paths through code**
- **paying insufficient consideration to the impact that arises when increasing the number of users or the volume of transactions have to be processed (a system fails to scale effectively as the load on the system increases)**
- **insufficient consideration of operational management**
- **limiting the interfaces to screen designs and the database designs and failing to consider in sufficient detail internal sub-system designs and interfaces that can encapsulate each component of logic; a component change generates knock-on changes to dependent components so that component re-use between systems cannot be presumed to be viable**
- **failing to design interfaces at system boundaries; this causes stovepipe applications and creates difficulties building adapters to the system later in the lifecycle**
- **implementing poor testing strategies for proving individual components, for carrying out integration, system and acceptance testing, often with no automated regression testing harnesses; too often this results in testing being excessively expensive, time consuming, late, poorly executed and unable to catch bugs which are then put into production only to produce high support costs and reputation risk for the supplier.**

These problems have arisen largely because of the absolute focus on delivery on time. This means that few projects are given sufficient resources for project teams to:

- **design the architecture, standards and common software components**
- **include the features necessary to address these generic challenges**
- **implement the business logic as an independent part of the whole.**

To deliver a SOA/ESB approach it is essential that activities are divided between:

- **an architecture group that identifies the business case for technology deployment and use, that defines the technology stack that will support that architecture together with the standards and low level architecture in the form of common components and then establishes the programme phases for individual projects**
- **the individual project teams that will construct and deliver the individual business deliverables.**

Too often the architect has no or too little authority. He or she is relegated to the role of adviser, and then that advice

is ignored or discounted. Too often the value of the architect's work is limited. In these situations the role of architect rarely lasts long.

Essential elements of an SOA approach

The key features of an SOA are as follows:

- **instead of a design approach based on screens, code and a database leading to stove pipe construction, SOA applications are based on work flows to provide state management of the process — routing each stage of the process through user screens and back end services until that process is complete (a 'middle-out' method)**
- **there is a separation between services and screen interaction, thereby enabling a service to be invoked either by straight through processing (STP) or by a user function; if a service is capable of participating in an STP then all the data required to perform the operation must be marshalled in the message prior to service execution**
- **execution of a service is triggered by a persistent message, so nothing can be lost; once started the process must complete, or be in error and then be passed to exception handling.**

Indeed the logical conclusion is that SOA mandates a design pattern combining STP and exception handling. Failures are passed to an exception handling engine which either performs a 'fix up', enabling the process to be re-started, or re-routes the exception (changing the business process) or backs out the business process (Figure 7.1).

Analysis of problems

Below I will analyse some typical problems that occur with traditional applications. I will also explain how an SOA approach assists you to address them.

These problems are latent in most systems. When you adopt an SOA and start to deliver an STP model which justifies the expense of SOA development, I believe these problems will attack applications with a vengeance and can compromise whole programmes.

Indeed, these poor techniques are ingrained into most developers' as 'acceptable compromises'. This is due to the demand from business units that focuses on functionality. Unless there is a change to this design approach, these problems will be highlighted in an ESB solution.

For example, applications which could deadlock 'escape' the problem because transactions are triggered by user screen activity. The gaps between users pressing 'execute' are interspersed with think-time which provides the transaction, once started, the opportunity to complete. It also reduces the chance of two transactions running at precisely the same time on the same resource and deadlocking. However, if the same business logic is driven by a multi-threaded agent running flat out while messages exist in a queue, then there will be no think-time between transactions and the chance of deadlock increases dramatically.

This type of argument applies to other problem scenarios. If they can happen then they are far more likely to when driven by messages — if there is any possibility of bursts of message traffic being delivered to a service.

Locking, deadlocking and unbounded transactions

An application needs to lock records to guarantee that the records remain consistent while business rules are applied. Poor locking strategies are probably the single most significant aspect of design when implementing a scalable solution that will be subjected to significant loads.

There are fundamentally two locking strategies implemented at the database level:

- **strong locking or 'read then write' — holding locks between the read and subsequent writes**
- **weak locking; this involves 'read, read, write' where the data is initially read, locks are discarded, think-time occurs, then records are re-read to check that no interim change has since occurred and the updates are performed.**

There are two extremes of locking scope: brute force and delicate. A brute force strategy locks all the resources that might be needed. The delicate strategy locks only the minimum number of resources. Unfortunately 'read, read, write' is often poorly implemented. Sometimes there is no re-read checking, leaving the database vulnerable to lost updates and data loss.

Designers may introduce measures to minimize deadlocks. Tables can be updated tables in the same order and bottleneck locks can be taken to prevent 'collisions'.

If brute force minimizes the chance of processes running

concurrently, delicate locking allows locks to be acquired during the transaction — which might result in deadlock with:

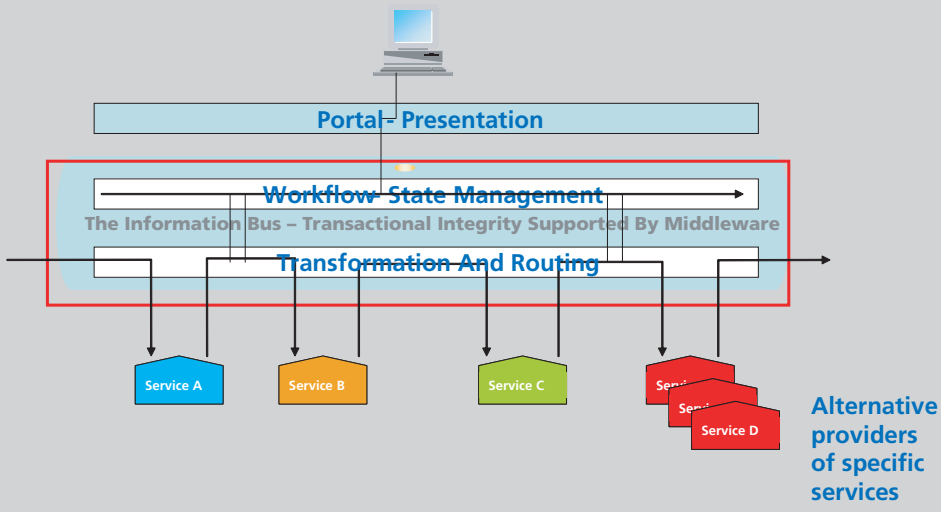
- **read-write holding locks for an unacceptable duration**
- **if there is think-time, 'read, read, write' may requiring complex checking (to ensure no change) this is often not performed.**

Deadlocks are a major problem in solutions with large numbers of concurrent users. Often they are not seen during testing but then occur when the system is under load with high volumes of business. The difficulty is that this tends to have a major impact on business performance. Poor deadlock handling can also impact the user interface with applications hanging and users idle waiting for a response. Large transactions can similarly hang the system by blocking all other transactions.

SOA encourages a design approach which will address these locking issues. Provided that transactions are properly designed so that either order does not matter or order is explicitly handled (see below in designing transactions) an SOA approach enables the solution to:

- **reduce a large un-bounded transaction to an unbounded set of bounded transactions which can run in parallel without blocking other transactions and hanging the system while using the message system to prevent any transaction becoming 'lost'**
- **using compensation to back out any transaction that has been broken down into a set of transactions, in the event that one transaction in the group fails**
- **building a co-operative lock to prevent other transactions starting when a large business transaction that requires a consistent view of the data is running**
- **implementing a re-try mechanism with messaging; a deadlock backs out the transaction, pushes the message back onto the queue and thereby automatically generates a retry**
- **routing a message, if the back out count exceeds a specific level, to an exception handler (this prevents infinite re-tries)**
- **providing an implicit throttle capability via single threading specific components**
- **having each individual transaction participate in an STP transaction; when it starts it must have all the data needed for the transaction, so there is no think-time and it can follow a strong/delicate locking model (there is no need for complex re-read checking and the likelihood of deadlocking is minimized).**

Figure 7.1: Elements of SOA/ESB



At a high level an ESB/SOA delivers a straight though process, with exception handling passed to users, probably running a portal

Polling

Many applications must present up-to-date information on a screen. Examples exist where changes in the database must be presented on screen as quickly as possible.

In the finance sector where users compete against each other fair delivery of data at the same time is important. A typical model for addressing this is that the application:

- **times-out after some pre-determined period**
- **executes each time a refresh query to re-populate the screen showing any data changes.**

This can impose major load on a database slowing the response of other transactions. Excessive deadlocks can arise when users attempt updates on the database dominated by 'refresh, query, read' locks.

To prevent an system from overloading, the refresh interval must be increased. But the gap between refreshes means that updates can be delivered unfairly.

A publishing model which broadcasts to users details of the records that have changed can resolve this problem. However, an alert to multiple users that instantly triggers hundreds of queries can also overload the database when a change occurs can be overwhelming.

One solution is to implement a publish and subscribe model (pub/sub). When the data changes in the database, applications are sent messages informing them of the change and including the full message payload (so there is no requirement for a refresh query). Ideally the subscription mechanism is sufficient to identify precisely the set of records required by an application.

Where this is not possible, use of a course grained subscription model works, with a fine grained filter in the subscribing application. Thus the load on the system is consistent, minimized and configurable: under heavy load the number of subscribers can be reduced, focusing on only the most important applications.

SOA supports pub/sub. In a subsequent analysis, I will describe how SOA applications can be significantly enhanced by using the pub/sub approach. Of particular note here is that we believe, at Strategic Thought, there is no place for polling in an SOA application.

Designing transactions

There are identifiable problems with transaction design approaches which permit data inconsistencies to arise. When data is presented on a screen, some errors may be identified, checked and corrected.

If transaction levels through STP solutions rise, and there is a commensurate increase in system load, should an old style failure occur resulting in transactions being rejected to the exception handling interface, an organization may not have sufficient staff to handle the errors. If the system becomes overloaded it may be difficult to reduce the load on the system to enable it to restart. Major business disruption may ensue.

Good design of transactions within an SOA is, therefore, vital to:

- **minimize the risk of failure**
- **provide a failsafe approach to business process delivery.**

The characteristics of a 'good' application are:

- **code follows the required locking and concurrency strategy to minimize the time taken to execute a transaction and the risk of blocking other transactions**
- **the time to complete a transaction is further minimized by addressing other factors — such as client/server communications ('chat') and including use of RDBMS stored procedures and placing co-operating objects in the same application server**
- **database code should optimize the use of high performance database features to minimize the load on the database and the time taken to execute a transaction**
- **the use of external resources (that might incur overhead) should be minimized, such as the use of large numbers of separate calls into an application server infrastructure (when calls could be made directly between objects in the same application server)**
- **right-sizing a transaction so that no transactions will take an excessive time to run, but the resources required by the transaction code nevertheless represent the majority of the overall work undertaken in the transaction, (in so doing, avoiding excessive load on the underlying infrastructure)**
- **applications are properly instrumented, thereby enabling them to be monitored so that any problems can be identified, isolated and resolved.**

An STP process must invoke underlying transactions with messages containing all necessary data which cannot become out of date during the period since the transaction was queued. This needs an alternative design approach from that used by conventional client/server applications.

SOA mandates a change in design approach and provides the mechanism to implement good applications. However, programming standards, appropriate designs and regular reviews of code produced remain essential.

Generating dynamic SQL

This flags a particular problem regarding designing good transactions. The JDBC API can encourage the use of dynamic SQL. Dynamic SQL is dangerous. It should be used sparingly because:

- **dynamic SQL is more difficult to test reliably because of the large number of test cases on the screen and in the database required to fully test applications**
- **the effectiveness of RDBMS query optimizers is limited if the query is different every time**
- **it is difficult to predict and control the level of load that a user can place on the system when using an application that supports dynamic SQL creation to vary the number of records accessed.**

The use of dynamic SQL is prevalent when supporting generic screens which are designed to respond in different ways depending on data entered onto the screen. The inefficiency results from the need to retrieve defined data that requires the joining of multiple tables with a dynamic 'WHERE' clause.

Designers are encouraged to implement specific transactions and to remove generic dynamic SQL transactions. The SOA approach encourages the construction of specific transactions to support well defined work flows. The benefits are:

- **consistent behavior of transactions**
- **better use of the database performance features**
- **well defined scenarios that can be tested properly**
- **an improved quality of the solution.**

Menu-based approach

Historically green screen applications had panels of menus which users would navigate to execute their business processes, often having to write down prime keys of records before they moved from one screen to the next. Such applications were often highly complex and needed a significant level of training before a user could operate the application.

This meant that:

- **different business processes occurred in separate offices where people found different ways to achieve the objective**
- **there were many potentially 'illegal' paths through the system that needed to be tested**
- **it was possible in some cases to forget to complete a business transaction where there was no work flow schedule of outstanding activity for completion; instead users had to run a query or report to find the work outstanding and then enter the keys, record by record.**

Using the process components of an ESB/SOA can easily address these problems:

- **tasks exist as messages in the work flow and they cannot be forgotten if the user is distracted or the system is interrupted**
- **work flows built 'above' the application can automatically step a user through the screen, following a business process and carry references to the underlying records**
- **by applying business rules to the work flow simple processes can be routed to less skilled operators while difficult tasks can be passed to highly skilled staff, thereby minimizing per transaction staff costs and optimizing use of scarce, highly trained people**
- **the number of paths through the code is defined, and a complete test plan can be constructed**
- **time is not wasted navigating between menu screens; this improves productivity, and the information needed for the operation is provided by the system (minimizing the risk of re-keying error)**
- **if there is a delay processing a component of work the user can process other transactions, improving productivity**
- **work flows can be instrumented to that user performance data can be monitored, which can trigger re-training, re-engineering of the solution or such other actions as are necessary to pin-point problems and then address them.**

Performance and scalability

Designers seldom include performance and scalability as an integral part of the design. The following need to be addressed — how to:

- **contain the cost of queries as the size of the underlying data set increases**
- **ensure that data required by transactions is spread out so that as the number of concurrent transactions increases the transactions do not lock each other out**
- **ensure that the commonly accessed data is most likely to be held in cache from previous queries limiting the I/O load**
- **determine what additional disks can be added to the system and I/Os spread across those disks**
- **ensure that the solution is multi-threaded and can utilize additional CPUs if they are added to the system**
- **determine ways in which a system can be partitioned across multiple computers to provide horizontal scalability without incurring excessive inter-machine coordination costs (such as a distributed lock manager overhead).**

To explore these issues would take a separate analysis. But they must be catered for and addressed by the designer.

Performance and scalability are hugely enhanced by using an SOA. The message based approach enables organizations to:

- **prioritize work for expensive human intervention**
- **run throughput in parallel optimising use of resources**
- **enable additional hardware to be introduced and fully exploited as a system takes on more load**
- **monitor throughput and identify bottlenecks and faults that impact performance and scalability so that these can be addressed**
- **introduce 'throttling' and 'holding pens' to manage deadlocks (as previously discussed)**
- **identify and address the scalability and throughput limits before they are reached in the production environment.**

Operational management

Applications tend not to consider the issues of operational management and the mechanisms by which the system can continue in the event of a failure of part of the system. Typical, if inadequate, approaches include:

- **running a solution on multiple platforms in a mode that enables business to continue in the event of the loss of one platform**
- **shutdown of an installation**
- **bringing on line for limited periods additional engines to provide additional capacity and then shut this down when a peak is passed**
- **running multiple versions of system and application software in the same production environment during a phased release regime, as new software releases are brought into limited production, certified for all traffic and then released for general use**
- **routing identifiable classes of traffic down particular engines when an engine is in limited production.**

Though the use of a common infrastructure layer supporting SOA a number of operational management capabilities are included:

- **tools to manage the infrastructure are extensible and applications can be instrumented to enable them to be managed by the underlying management technology**
- **perhaps 30% of an application is associated with management functions; deploying an SOA application reduces the complexity of any application because it can utilize the standard approach**
- **once a common enterprise scale management infrastructure is deployed, the added cost of introducing new applications is minimal, so all subsequent solutions can be delivered to enterprise levels of manageablilty**
- **a common infrastructure should enable a common BC/DR approach to be adopted which should be less complex, more reliable to implement and less costly to test**
- **the cost of maintaining skills of operational staff will be higher initially but the use of a single infrastructure willultimately minimize the number of separate operational skills that need to be maintained, and their associated costs.**

Integration

Typically integration between applications is extremely difficult because separate applications have different data

models. It is also difficult to track transactions through an integrated system because many data models do not support transaction labeling of data between input and output adapters.

The concept of integration between stovepipes is fundamentally changed by use of an SOA. Focusing on a 'middle-out' approach — where each solution starts with a work flow to which are attached a set of services and exception handling screens — removes the need for stovepipe development.

Existing integration approaches are essential to SOA because of the need to enrich content and transform formats between services to ensure that the outputs from one service can be modified to match the inputs of the next service. At Strategic Thought we support the generic business object (GBO) and application specific object (ASBO), and conversion between GBO and ASBO through the work flow.

Testing

It is relatively infrequent that automated testing suites are developed for solutions. Even when they are, too frequently they are only for the framework. In addition, test cases are not maintained in line with the code. Then test harnesses fall into disuse. As a consequence bugs creep into applications over time, performance and scalability are not monitored and unexpected problems can occur in production when the system becomes loaded.

Building test harnesses around user interfaces is particularly prone to problems because of the impact of changes to the GUI. Building regression test suites around messaging systems is far less vulnerable to this because it is more straightforward to:

- **manage changes to messaging interfaces and their associated test frameworks**
- **collect and compare the results from messaging based systems.**

Moving to an ESB should facilitate improved regression testing — if only because it is so oriented to messaging. Adoption of an SOA can be expected to greatly enhance the ability to test enterprise applications for both functionality and performance because of the message-based approach that underlies it:

- **individual objects can be tested exhaustively driven by large sets of test data prepared as messages**
- **outputs are also messages and the results of separate tests can be compared to prove correctness**
- **because each service has a well defined interface which should only change under change control, test data can be prepared and re-used many times reducing the cost of maintaining regression test suites**
- **driving solutions with large numbers of pre-prepared test messages will enable the upper limits of performance and scalability to be identified and addressed prior to the throughput levels being reached in production**
- **performance testing also generates failure conditions — such as deadlock and re-try which can then be isolated and 'designed out'; these are scenarios that are often difficult to identify in a functional test environment**
- **a message-based approach enables organizations to capture production data for re-running in a test environment, either as representative data or for business analysis purposes such as fraud investigation or business planning.**

Classic 'mistakes'

There are typically four reasons why functionally 'correct' applications that fully comply with their specifications fail at the 11th hour to go into production and end up being scrapped. It is critical to consider the impact of these problems when considering any new SOA/ESB application design (and development patterns) to ensure that the delivered solution can be successfully deployed into production.

- **the GUI interface is not intuitive and the application, while functionally correct, is unworkable from an ergonomic perspective**
- **there is no migration path from the data structures in the existing system to the data structures in the new system and a migration of existing business is not possible**
- **the business processes from the old system cannot be mapped to the business processes of the new system and as a consequence existing contractual obligations cannot be fulfilled on the new system**
- **the new application does not correctly model the business processes and in particular does not take account of time.**

A specific example of this latter one can occur where the system is required to implement certain business rules. These are implemented in the system on the primary data

entry screen. However a common error is to assume all the data is available when the record is first created. In fact, often the business process is followed by progressively adding data to the customer record during the course of sales process and it is only possible to apply the business rules at the final business operation.

Flawed business requirements and functional design have been known to produce systems that cannot be operated in practice because the application of business rules is inappropriate, making the system unworkable.

Implementing an SOA is not going to remove the possibility of all 'cock-ups'. An SOA solution, however, is work flow focused and, therefore, introduces the concept of time plus a design approach that:

- **is business process oriented**
- **takes into account time through the business process and the separate actions required to complete the process (breaking down the business transaction into a series of separate design transactions)**
- **should highlight scenarios where information is not available at the start of the process and is only discovered during the course of the work flow, and then deal with these stages**
- **only deliver tasks to users trained to execute them**
- **provide a mechanism whereby existing applications can be re-faced with SOA services — minimizing the requirements to migrate data from old to new data models.**

All of these factors inherent in an SOA approach should improve the analysis of carried out in the early stages of a project. In turn this will reduce the number of system deployments pulled at the 11th hour to the embarrassment of all.

Management conclusion

In this analysis Nick Denning examines some of the more common problems that he and his company have encountered in client situations. He relates these to SOAs and ESBs and shows how these might assist in the delivery of improved solutions and applications.

*In a subsequent **MIDDLEWARESPECTRA** he will take the next step — and describe a common software architecture that can:*

- ***exploit a consistent model***
- ***address relevant patterns***
- ***map applications styles to an architecture***

all as part of a road map for embarking on SOA/ESB adoption.

MIDDLEWARE**SPECTRA**
is published and distributed worldwide by:

Subscription Center
19 St. Michael's Road
Winchester SO23 9JE
England
Telephone: +44 1962 878333
Fax: +44 1962 878333

Research and Editorial Office
19 St. Michael's Road
Winchester SO23 9JE
England
Telephone: +44 1962 878333
Fax: +44 1962 878333

Email and Internet
Email:
spectrum@middlewarespectra.com

World Wide Web:
www.middlewarespectra.com

ISSN 1356-9570

[incorporating FINANCIAL MIDDLEWARESPECTRA ISSN 1460-7220]