Architecting e-business Solutions with IBM’s Business Transformation & Integration Middleware

Keith Edwards
Senior Consultant Architect
Transformation & Integration Middleware
IBM Software Group

Introduction

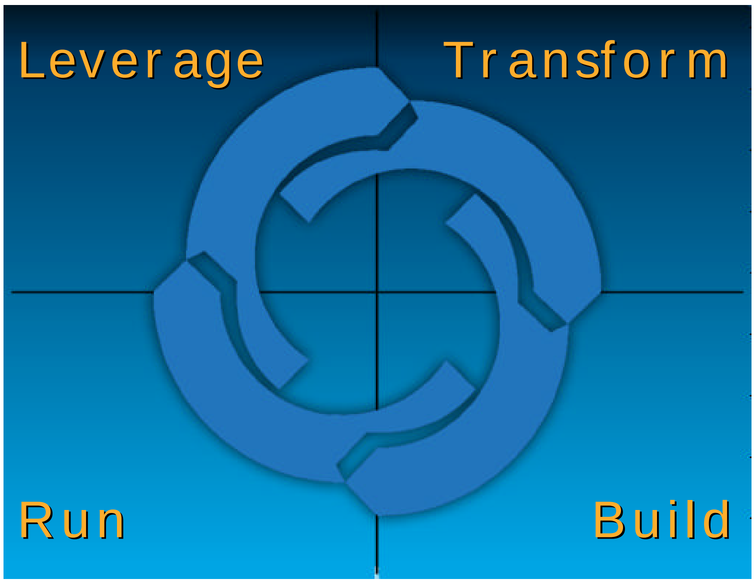
It's all about business -- or should that be e-business.

This phrase, or any number of derivatives, is extremely popular with the IT industry, highlighting a desire to provide solutions to business problems rather than just technology. The requirement to understand the business models, and their associated processes, required to compete in whatever industry you are in is of paramount importance.

This is particularly true when considering the use of the Internet as a delivery channel for goods or services. Of course the primary business driver is to understand if you have a successful business model, and at this point at the beginning of the twenty-first century (OK, I didn't need to put that phrase in, but I always wanted to write it, and it is my document!) the jury is still out on some of the new Internet based business models.

One of the most interesting aspects of architecting e-business solutions is that fact that to do it successfully you really do need to have an understanding of the business processes that need to be implemented -- although you don't need to know whether the actual business model works, just what it is supposed to do!!

IBM's high level approach to providing a way of thinking about e-business is by use of the e-business cycle, illustrated below



This encourages the process to begin with what business process you wish to implement and in particular how they might be affected, or **Transform**ed, by delivery via an Internet channel.

For example, a bank's loan process usually involves an assessment by a loan officer of a customer credit risk by the examination of things such as, the state of the customer's accounts, a credit report about the customer, the employment record of the customer and of course the amount of money required. The traditional approach to completing this process may take many minutes if not hours, and yet if deployment is required via the Internet, where the time for response would be of the order of one minute, the whole way the credit risk assessment is done has to be reexamined closely.

Understanding the business process is very important because when we come to the next step, namely, **Build**ing applications to support the process, we need to have the business process to refer to in order to successfully architect a solution. In the e-business cycle the build part refers to all of the software technology that goes in to the creation and deployment of those applications. The **Run** part of the cycle is used to highlight the middleware requirements and hardware requirements that might be needed based of course on the business requirements. Finally the
Leverage part of the cycle is where you gather the information you collect while running the applications and figure out better ways (well hopefully better!) of doing business, which then feeds the beginning of the cycle again.

This document will focus of the software aspects of the **Build** and **Run** parts of the e-business cycle, making the assumption that the business process required has been determined.

I believe that all application designers would agree (yes I know that is quite an assumption on my part, but this next bit is really common sense!) that the ideal approach to building an application is to use the following steps:

1) Choose/design the application architecture to support your business process.
2) Choose the middleware software to implement the application architecture.
3) Choose the tools to build the applications to the middleware specified.
4) Deploy the application to the hardware/operating system platform that makes the most business sense.

You should realize that there is a step zero, which is to understand the business process that we will design the application architecture for. In addition, those of you with a development management background will probably be thinking that this sounds all very well in theory, but when it comes to actually running a project you are basically measured on three things; the creation of the application to a desired functional specification (which of course you are allowed to change!); the expenditure of a certain amount of resource, people, machines, money, etc. (which you can also change, the most common change being to increase this expenditure!); and finally you have to deliver on a set date -- and that you may not change (well okay you can bring the date forward, but it is a rare project that achieves that).

As a consequence of the date pressure, there is usually a temptation to get started as quickly as possible, and the fastest way to start is to pick a development tool and start writing code. This usually leads to a predetermined set of middleware (or even the creation of your own middleware) and evolves into an architecture that may or may not allow the deployment to the best business choice platform.

Essentially, this reverses the steps three, two and one, and I describe this 3-2-1 process as development using the process of I hope I get lucky.

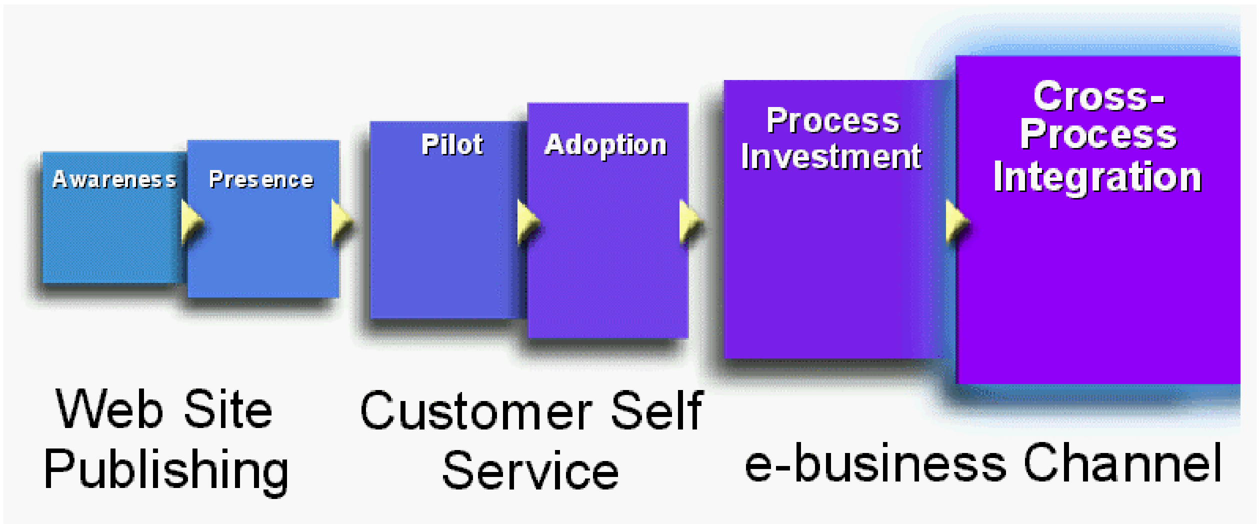
However, development (as with life) is never usually this clear cut. I could also describe the 3-2-1 process as one with significant risk in how the application can be deployed, or RUN. The longer the application will have to be supported in RUN, the greater the risk, and by association you can definitely assume the greater the support cost.

If my application is not going to be needed for a long period of time, for example an application to support a six month marketing campaign, perhaps I can afford to take the risk on deployment.

However, if I am implementing a core business process, then I hopefully will want to have that deployed, with the capability to change, for a long period of time and the deployment risk of 3-2-1 would be great.

Note that this choice is driven by how important is the business process and reinforces the requirement to understand what the business process is for when designing applications.

Finally, think about the stages that most businesses go through with their use of web technology. These stages are depicted below:



Each of these stages affects the business to a differing degree.

The first stage, which involves little more than establishing a web site, can be thought of as required in order to play in this space. No real business benefit occurs, rather you avoid the detriment of not appearing when someone types www.*your_company_name*.com in the browser and nothing appears -- a result that, to many people, indicates you are no longer in business!

The second stage is where things start to get interesting when you allow the end (web) user to get access to information about themselves. This is where you begin to modify the behavior of that user in their interaction with your company, and the importance of that interaction begins to increase. It is usually one of several ways in which that information can be gained and as a consequence the impact of its going away is limited -- it is annoying to the end user, but it does not stop them from conducting business.

In the third stage, the business has adapted the processes to make the web the delivery channel for the business process allowing a full interchange of information with the end user. In this case the end user's behavior has been modified completely to rely on this channel. If the channel goes away it is a major problem to the end user, resulting in no business being conducted. Consider the online share trading companies as an example of this business critical environment. If one of the trading sites goes down, it makes CNN headline news, and the CEO of the company has to explain what happened. The brand image of the company is staked on the success or failure of the web channel.

So why did I go through this discussion? Well, it is to point out that decisions made in the earlier two stages can impact the final stage if the architecture of the solution is not well understood. That impact is not usually for the better! So even at the beginning the ideal should be to think about architecture -- or at least be aware that you did not if you then need to take the environment forward into the higher levels of e-business.

This introduction should have impressed upon you that the application architecture is something really important, which is why this paper is about architecting. So let’s architect!

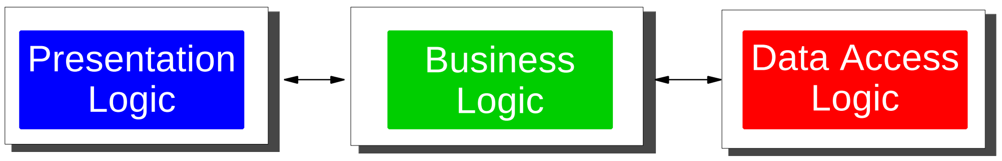
Designing an Application Architecture for e-business

The goal in designing an application architecture is to determine the capabilities required to support the business process and then provide for the those capabilities in the form of features and functions that define an architecture. The features and functions should ideally be expressed in non product specific terms and ideally without implying any specific programming model.

The design foundation for an e-business application architecture should be three logical tiers.

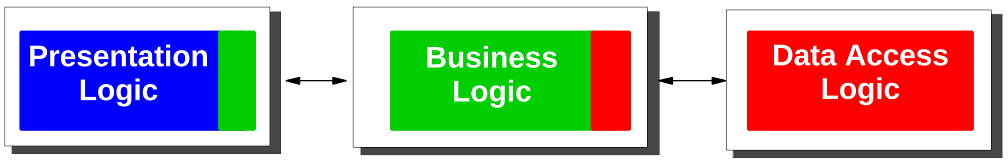
The first tier handling the presentation capabilities, or how the end users will interact with the application; the second tier handles the business logic, or what process is supposed to happen; and the third tier handles the information access that the business logic will need to manipulate.

This is most commonly displayed in the following pictorial form:



This foundation implies the separation of these elements of an application, but it should be emphasized that this is a logical way to build these elements separately. When it comes to deploying the pieces that have been built, the deployment could well be on one physical machine or multiple machines. Ideally we would like the flexibility to choose where to put the elements based on business requirements, perhaps associated with scale or type of end user device.

In addition the previous diagram is a little simplistic in that we need some way to connect each of the boxes together. This means that we need some business logic connector (or client) for the presentation box and a data store connector/client for the business logic box. This adaptation is shown below.



In general the IT industry agrees with this approach as the foundation for e-business applications. Of course there is a lot of disagreement when it comes time to select products to implement, but that discussion is for a later chapter.

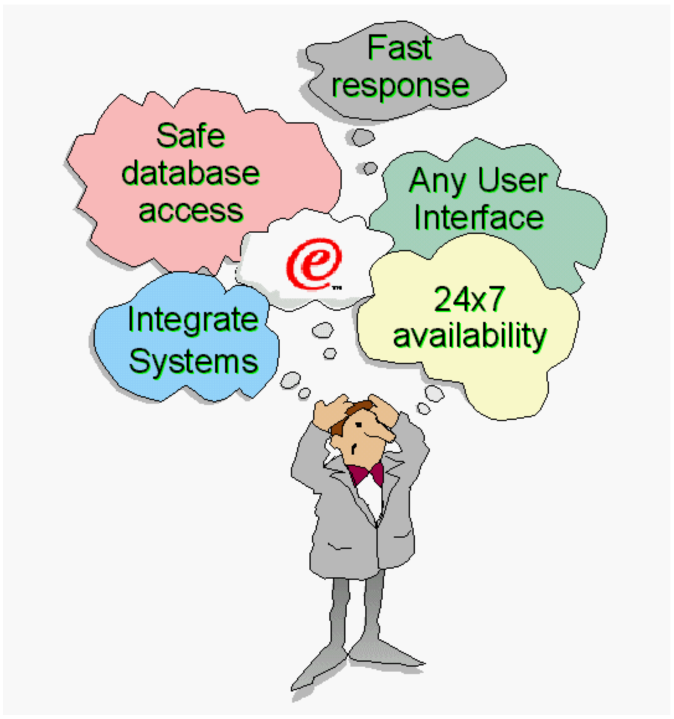
In some ways you can regard this as the end result in the debate that occurred in the industry over three tier versus two tier client server design. The key point being that two tier client server systems have limits to the number of end users that can be supported - quite a drawback if your user population is the World Wide Web!

For those interested in the discussion on two tier versus three tier systems, I have appended a section on this at the back of this document.

However, I will state that logical three tier is the foundation and go on from here.

Desirable Capabilities for e-business

I said that the goal in defining an application architecture is to determine the capabilities required to support the business and then design the features and functions needed to provide the capability. Clearly there are many things that business processes might need and it is beyond the scope of this document to cover every possible capability. So instead I will concentrate on the key ones that are needed for most (if not all) e-business applications. These are shown pictorially below and explained in the subsequent text.



First -- make sure you have fast response time. Response time is a key measure of end user satisfaction. This can be illustrated by asking yourself the following series of questions:

Have you used a web browser in your home to buy something over the Internet? (When I ask that question of audiences I present to I usually get about an 80 to 90% positive response).
Then ask yourself, -- after you entered your credit card details and then clicked the submit icon on the screen -- how long did you feel comfortable waiting for the response to come back -- (when I ask that question the answer usually is between 2 to 10 seconds; and in the first half of 1999 it was 5 to 15 seconds -- so people are expecting faster response!)

The key point is everyone has a period of time they are prepared to wait -- if you exceed that time they become uncomfortable (or dissatisfied) with your company -- what they choose to do with that feeling depends on the relationship they have with your company -- if this is the first time into the web site they might just go back to the search engine and go find another company -- if they have used your online services before -- they probably will continue to do business with you -- but you are giving them a reason to go look elsewhere -- which is really easy on the Internet.

It is also highly desirable to provide multiple options for a user interface. Particularly where the application is accessible across an Enterprise as well as the Internet. For some tasks, for example an application for a well trained call center representative, a character based user interface will be more productive than a graphical user interface. If the call center has a large turnover in staff, then ease of use may be more important, in which case a graphical user interface may be better. Note that the decision here is based on the business characteristics of the call center, leading to a choice in end user device. This choice is particularly relevant when you consider the impact of pervasive computing, where the use of small devices to drive applications will become more common -- for example using your mobile phone to submit orders.

Once you have an effective application that provides real value to the end users, you will find that they will want to be able to use those services at times they find convenient, particularly if your company has a global presence -- it's always part of the business day some where on the planet! As a consequence the requirement to run the application 24 hours a day, 7 days a week, 52 weeks a year becomes very strong.

You are also likely to be faced with the Integration of different business systems. There are two areas to consider -- the integration of multiple business systems to provide a uniform front end interface and then the integration of the business systems themselves so they can intercommunicate and complete an entire business process.

Finally (at least in this subset of capability) the moment you allow an end user to use an application to change the state of information held in the Enterprise you must make sure that this can be done safely -- this is not a statement about security, although that is important, rather it is about making sure that all updates an application makes either happen or don't happen. This is the concept of transaction processing called unit of work management -- where an application server ensures that updates to datastores are coordinated and either all happen - are committed - or all don't happen - are rolled back.

When multiple data updates are made the requirement for two phase commitment processing will come into the picture.

So we have a series of capabilities and my intent here is to start the thought process about what might be needed to support your specific business processes.

We will now look at the various aspects of a three tier application architecture and look at the implications of the choices to be made.

Choices in Three Tier Application Design

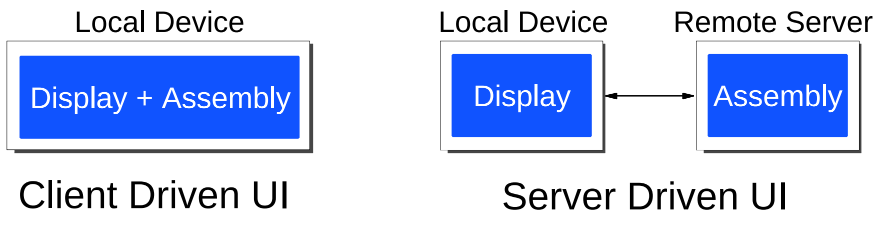
I stated that use of a three tier foundation was the best way to design e-business systems. There are choices to be made in how we design our three tier system, and those choices will affect how we can deploy our application.

We must look at the type of presentation devices we wish to support in the presentation tier and how that presentation tier is to communicate with the business logic tier.

We must also look at the requirements in the business logic tier, particularly with regards to the updating of data and how we will access sources of data, where those sources might be database systems or other business systems, either in our enterprise or outside it.

Design Considerations for the Presentation Tier

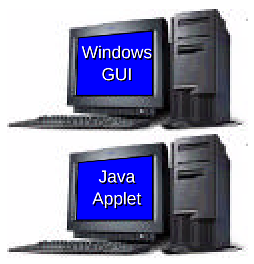
There are two things that must be accomplished in the presentation tier. The output to the end user device must be assembled and then that output must be displayed. Clearly the display of the output must happen on the end user's device. The assembly of that output can either be done on the device, which is called a Client Driven user interface (UI); or can be done remotely, usually on some sort of server, hence this is called a Server Driven user interface(UI). These two choices are shown pictorially below.



Both of these options provide a way for an end user to interact with some sort of business function and each has benefits and drawbacks. Note that in each case the assembly logic can be regarded as presentation side business logic.

Client Driven User Interfaces

An example of a Client Driven User Interface is a full function PC client, perhaps in a two tier, windows based, client/server application. I will refer to this as a traditional PC client. In this case the application has been installed, either by the end user or by some network management system. A Java applet running in a web browser can also be considered a Client Driven UI.



Client Driven User Intefaces

The key is that all the presentation creation and display is on the local machine.

This provides some key benefits.

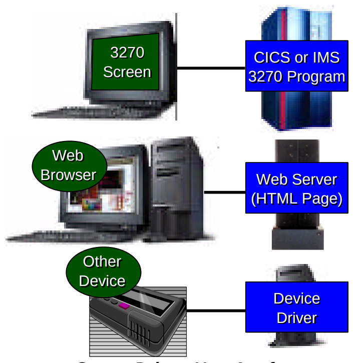
The application does not have to send a request over the network to perform tasks like field validation. This allows the local device to perform much of the end user interaction and limit the requests heading over the network This can improve the response time of applications that need to communicate over slow networks versus a server based user interface, a particular benefit for web based applications using Java applets. It also reduces the burden placed on any server to which this Client Driven User Interface will be connected, and if combined with some business logic and data storage capability, also allows the application to operate when disconnected from the network.

There are of course some drawbacks.

The Client Driven user interface typically requires a reasonably high function end user device. This tends to have a higher cost than with a Server Driven user interface. In addition the application software running on the client has to be deployed, which in traditional PC clients introduces a significant cost for management. This cost is less where the application can be deployed automatically, for example in the case of a Java applet downloaded to a workstation's web browser. In the case of this approach the application downloaded is usually dependent on a specific level of operating system on the client. This can lead to additional cost in deploying new releases of an application, where the new release requires additional function that may not be present in the end users operating system, requiring an upgrade to the end user device. For example, Java applets are dependent on the Java Development Kit (JDK) level that the web browser is at. Successful deployment depends on being at the correct level. This fact is an inhibitor to the deployment of applets in Internet based applications, where the supplier of the applet does not control which web browser the end user has.

Server Driven User Interfaces

An example of a server driven user interface is a traditional mainframe, green screen, application, perhaps using CICS or IMS to create a 3270 screen for display on a 3270 device. A web browser displaying HTML pages, served from a web server is also an example of a server driven user interface, as is any device that can receive information assembled remotely, for example a pager.



Server Driven User Intefaces

The key element is that the assembly of that which is to be displayed is done remotely from the device.

This provides some key benefits.

The end user device only has to be able to display the information received. This allows for relatively low function devices, compared to the client side user interface case, to be deployed. resulting in a lower cost solution. There is also no client side code to manage or download which leads to much lower management costs. The device still has to be able to display the information received, but given the lower functional requirements, it is more likely that the end user device will remain a viable display platform compared to the client driven user interface. Of course sometimes, upgrades to the device are required. For example, if you wanted to be able to send pages of information to web browsers in XML rather than HTML, you would need the end users browser to support this.

There are of course some drawbacks.

In order to be able to function the end user device must be connected. Therefore, this is not a viable option for applications that require the end user to be able to operate disconnected from the network. In addition each user interaction must be passed over the network to the remote server. If we have an application that requires may interactions, for example, editing (or field validation) of the end user input, where the network capacity between the device and the remote server is low, we may significantly increase the time taken for an end user to perform their task compared to a client driven user interface.

Size of the User Community - Scale of the Presentation Tier

Ideally we want our presentation tier to be able to support any number of users, particularly if we are building an Internet based application, where our potential audience is numbered in the millions (well in the wildest dreams of most marketing organizations it is!).

Please note that at this point we are only going to consider the question to scale for the user interface tier and not the business logic tier that it will be connected to - that discussion happens later on.

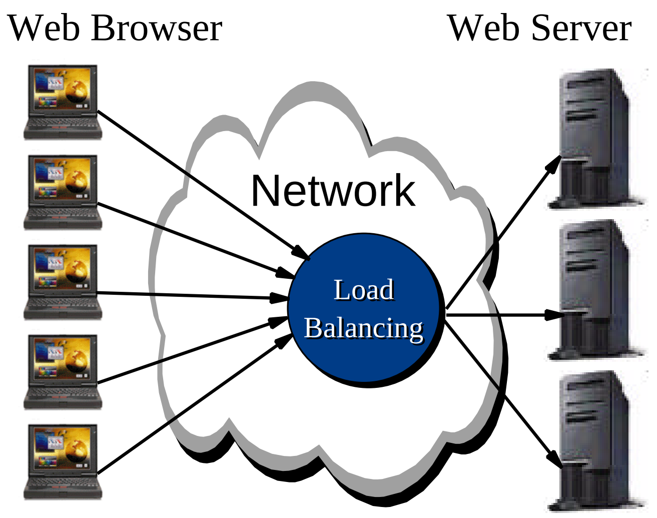
In the client driven user interface case we have a one-to-one relationship between the device our end user has for display and for assembly. In this case we can support any number of end users as long as we can supply them the device. We just have to worry about how much the device costs and the management of the software on that device.

In the simplest server driven user interface case we have a many-to-one relationship. Many display devices can be connected to one server. However, there is a limit to the number of devices that can be connected to any one server. That number varies depending on the type of end user device and type of server, but the key principal is that there is always a limit. If we want to scale to any number of end users we must allow for more than one server, and therefore support a many-to-many relationship. Scale in this case is a question of how many users can one server support and then how many servers do I want to have. If one server can support 1000 end users then two servers should be able to support 2000 and so on in a linear progression. This is referred to as linear scaleability - and the practical limit is how many servers do you want to buy!

Once we have more than one server to assemble the end users presentation output, we also have to determine which server our display device should connect to - after all if all the devices tried to connect to the same server, that would defeat the whole rationale of providing multiple servers.

This is done by load balancing the requests over all of the servers that could handle the request. This load balancing is most commonly achieved somewhere in the network between the end user device and the presentation server. The primary reason for this is that we want our end user device to be as low in function as possible and we also do not want to reveal the server topology to the end user.

If we take a web browser, web server combination as an example then we would have a high level design as shown below.



Network based load balancing for server driven user interface

In the simplest case the load balancing alternates requests across all of the servers, a process referred to as round-robin load balancing. However, we would ideally like to have some greater capability than this to improve the operational characteristics of the system.

For some requests we may want a user to connect to a server, and then have all subsequent requests go to the same sever, rather than have each request go to a random server. This is usually done to avoid the connection costs normally associated with first time connection to a server for every request.

All requests are not equal in the amount of processing they may require, and as a result a simple round robin load balancing approach may result in an unequal distribution of work amongst the servers. Ideally we would want the load balancing to take into account the load on the server when making the decision about where to send the request.

If one of the servers should fail we would also like the load balancing to detect this and prevent requests being routed to the failed server. This capability allows us to provide a high availability solution. The capability to add and subtract servers from the load balancing pool is also desirable. This allows us to add capacity dynamically if we are expecting an increased load, or to restore a failed server to the pool.

If we are going to provide a high availability solution then we must ensure that our load balancing does not become a single point of failure for the system, so there must be redundancy in the load balancing mechanism, and ideally multiple network paths for the requests to be routed along.

Key Principal Reminder

The key principal to remember is that both client driven and server driven user interface can achieve the same result - namely the display of information to an end user.

If you were to look at the problem from just a “BUILD” perspective you would conclude that it did not matter which one was selected. However, when the “RUN” characteristics of each choice are considered, the deployment requirements, which are based on the business requirement of who is the end user, indicate that the choice does matter - hence reinforcing my point made in the introduction about the value of considering the architectural implications before building your application.

Design Considerations for the Business Logic Tier

The business logic tier’s function is to execute the business process requested by the presentation tier. This can range from a simple single look up in a database, to a complex chain of multiple business processes.

The main areas of consideration when architecting this layer are, minimizing the response time, determining whether the business logic updates information, supporting the appropriate number of requests and maintaining availability.

Minimize Response Time

A key measurements of end user satisfaction is that the applications they use have fast response times. As a consequence of this performance requirement, the design of what is to be executed, and in what order, needs to be carefully considered. The best approach for this consideration is to define the minimum business function that needs to be executed before a response can be sent back to the end user and ensure that is what is implemented. Any additional work can be completed after that response has gone back. Of course in order to be able to do this, an understanding of the business process and its minimum requirements is needed.

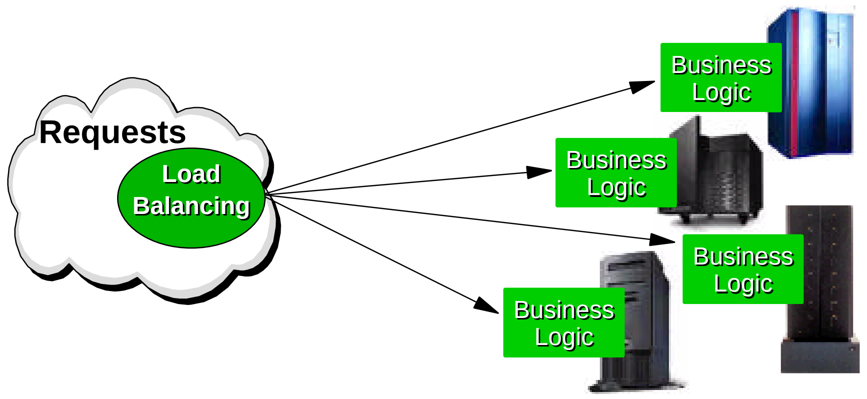
Read Only or Update?

The business logic also needs the capability to access data, either from data storage systems, such as a relational database or via other business processes. To meet this requirement, the business logic tier must be able to find and connect to the source of data; request data from the source; send data back to the source (assuming that we want to update some of that data); and indicate that we have finished using the source of data if we are updating it. For update of data there needs to be transactional control over the updates. In the case of a single data source, the data source itself can provide this capability, however, if we need to update multiple sources of data the business logic tier will need to coordinate the transaction. Appendix II provides an explanation of transactions.

How Many Requests are Needed?

When designing an application architecture it is desirable to design in such as way as to be able to support an unlimited number of requests - assuming that you have an unlimited amount of hardware to run them on. Of course when it comes to deploying the application, practical limits associated with how much money do you have to spend on hardware come into play - but this should be the limit - not the application architecture.

The hardware platform selected for deployment will determine how much one physical machine can handle. If more than one machines capacity is needed then the business logic must be made available on multiple machines, where those machines could be of differing types, as shown in the diagram below.



It is necessary to provide a load balancing capability for the requests over all of the servers. This is exactly the same principle discussed in the presentation tier for server driven user interfaces.

However, whereas the load balancing capability could be placed in the network as with the server driven user interface, it could also be deployed at the requesting device, or client. This is discussed further in the section *Connecting the Presentation Tier to the Business Logic Tier*.

The load balancing capability should be capable of distinguishing between servers of differing capacity and route requests accordingly - rather than provide simple round robin load balancing. There should also be the capability to add or remove servers to or from the load balancing scheme without bringing the system down.

Maintaining Availability

If this load balancing capability is in place and it can also detect failed servers, then fail over of requests can be achieved to provide a high availability solution. In the case of fail over it is desirable that there be the capability to automatically fail over without having the requester resubmit the request.

The Role of an Application Server

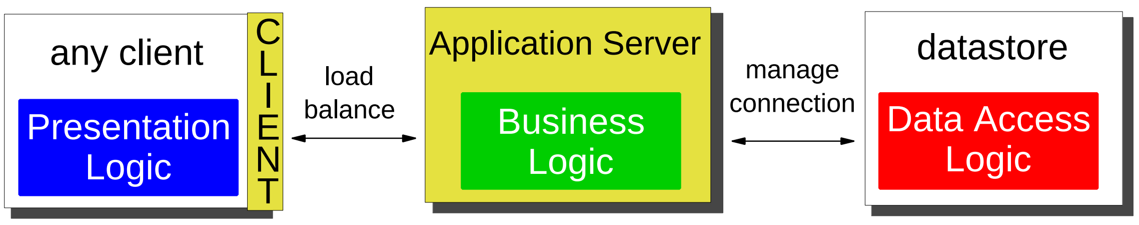
The capabilities previously discussed are usually provided by an application server of some kind.

From a simple viewpoint an application server should provide a run time environment, or container, for the business logic in a logical three tier application architecture with the capability to support the requirements previously discussed.

The application server should also provide a client, a small piece of software, that is used in the presentation layer to connect it to the application server - remember that the presentation layer may be on the same machine as the application server or on some other machine. This client should provide some way of load balancing across multiple servers.

The application servers also provide a choice - some more than others - in the programming model that can be used to create the application business logic - we will look more closely at these choices later. Note that the programming model is not a determining factor in the architecture - well at least it should not be!

The server should also provide standards based access to the data access logic or datastore - the server should also be responsible for managing the connection to this datastore and be responsible for coordinating the commitment of any updates to the datastore. The application server's management of the connection to the data source is needed to provide optimal performance, where the connection can be done at application server start up rather than at each execution of the business logic contained in the application server.



The coordination of updates is critical if the business logic needs to update information and that information is held in more than one datastore that the business logic accesses directly. If only one datastore is required then this requirement is less as the datastore can take responsibility for coordinating its own updates.

Connecting the Presentation Tier to the Business Logic Tier

In the section on *"How Many Requests are Needed"* I indicated that there was a choice in how load balancing of requests from the presentation tier to the business logic tier could be achieved.

There are two ways in which this connection can be achieved:

- By direct connection of the presentation tier to the business logic tier
- By routed connection of the presentation tier to the business logic tier

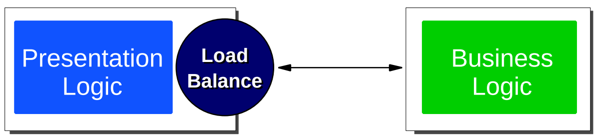
Note that from an application developers viewpoint this choice should be transparent and be part of the deployment or RUN part of the development cycle.

Direct Connection of the Presentation Tier to the Business Logic Tier

The first case is where the presentation tier is directly connected to the application server supporting the business logic tier. This minimizes the latency between the presentation tier and application server, providing for optimal response time.

In order to support the load balancing required to scale to any number of requests it is necessary that the client have awareness of all of the application servers that could satisfy any request and have the intelligence to perform the load balancing. Note that this does not mean that the application programmer at the client side should be aware of this - rather that the middleware client used to connect to the application server should be aware.

The key consideration of when to use this approach is that the "client" has to know about the server topology in order to support the load balancing for linear scalability. In this case it is usual that the client be under the control of the business logic provider and relatively close to the business logic, as any changes to the topology will have to be shared. This makes this approach most applicable for the server driven user interface approach design for the presentation layer.

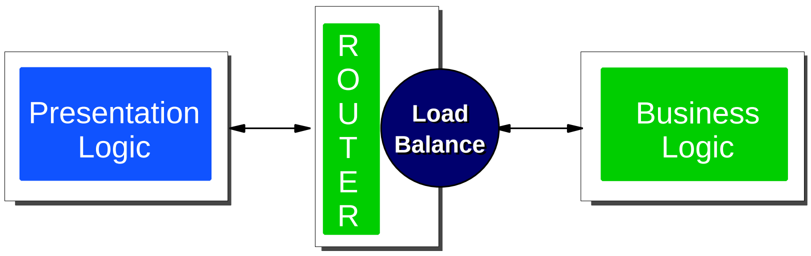


Direct connect to application server - load balancing on client

Routed Connection of the Presentation Tier to the Business Logic Tier

The second case is where the presentation tier connects to a router which then passes the request onto the business logic tier. This introduces an additional "hop" in processing the request and hence additional latency in the connection.

In this case the presentation tier needs to have no knowledge of the application server topology that is supporting the business logic tier. It depends on the router for this function, hence the router can be regarded as the point of load balancing. It should be noted that for a high availability solution there should be at least two such routers and that this implies that there be at least simple round robin network routing between the two with fail over capability should one of the routers fail. This makes this choice a more complex configuration to manage versus the direct connection approach.



Routed connect to application server - load balancing on router

These considerations lead to the conclusion that this choice is best suited to the client driven user interface design approach for the presentation layer, where the individual clients do not need to have the server topology exposed to them, thus reducing the management complexity.

However, it should be noted that for a small number of client, the burden of the additional router may actually increase the complexity versus a direct connection approach.

This is an indication that the client driven user interface approach becomes inherently more complex than the server driven user interface approach as the number of users increases.

The Effect of Latency in the Presentation to Business Logic Connection

Fast response time is a key requirement for all applications. The response time that an end user will see it comprised of:

- Speed at which the presentation layer can display the output
- Latency of the connection of the presentation layer to the application server
- Speed of the application server to execute the business logic
- Latency of the connection between the application server and the data source
- Speed at which the data source can process the request

In most deployments the application server and the data sources are usually under the control of the application provider and are usually tuned to execute very quickly and have minimum latency between the application server and data source.

The presentation layer may or may not be within the control of the provider. For example. in a business to consumer e-business application, the end users device is typically not under the application provider's control. This can result in a disproportionate amount of time being spent in the connection between the presentation layer and the application server - this is made worse as the amount of data passed increases. It is therefore advisable to design applications with the minimum amount of data passing between the presentation layer and the application server, and to try and make just one request to drive the required business process. The goal being to limit the impact of the network latency of the presentation layer to the application server layer - which is usually the link with the least capacity and speed - particularly in Internet based applications.

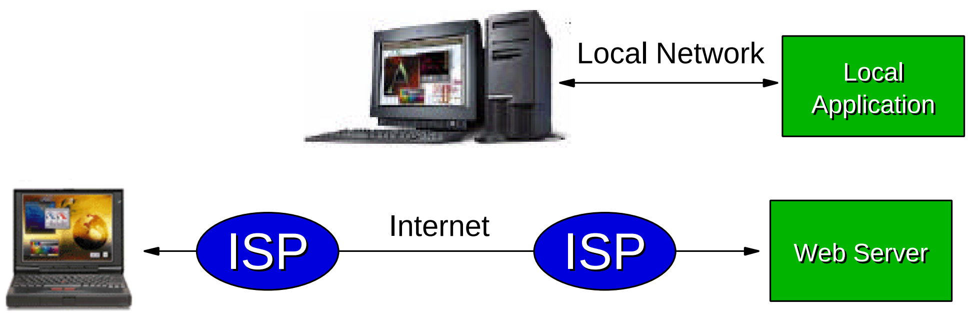
The network latency issue is also a factor in considering what is required of the final response back from the application server layer to the presentation layer.

The final reply to the presentation layer will always be outside of the transactional boundary of the business process that executed on the application server.

The final piece of information is exposed in that if we loose communication between the application server and the presentation layer just as we send the final response, the presentation layer will never receive that response - it will be lost.

In Intranet applications this is not usually a big deal as the probability of loosing a local connection from a client to a server in a local area network is very small, and the latency between the layers is also very small.

In Internet based applications, the probability of loosing the connection is much higher and the latency is also higher.



Much lower connection latency with local applications vs Internet applications

For example, an Internet user will have connected via their Internet Service Provider (ISP) which will connect via several routers to your ISP and then to your web servers. There is plenty of scope for a lost connection. This means that you can not guarantee to get the final screen to the end user, and you must consider the implications of this to your business model. The most common way this problem is over come, is to send an additional notice via e-mail of that which the end user has done. For example, if you ordered an item from an online store, that store will probably send you an e-mail confirming the order details.

Application Server Affinity

Once we have managed to connect our user to a business process, and sent back our response, the user may make a subsequent request. From a performance viewpoint, as well as not requiring the user to rekey data, it is desirable that we have some way to remember what the user did last time.

We can characterize the type of information being held in two ways.

1. Information we hold specific to the user, but not specific to the application - for example the fact that the user has signed on. I will call this **user context** information.
2. Information specific to the user and specific to the application - for example a shopping cart holding a customer order in a e-commerce application. I will call this **application context** information

These two types of information should be considered separately because of the scope in which they may be used and the duration for which the information is held.

User context can be used by any application the user makes a request of and is usually used to determine if the user is authorized to use the application. The duration for which user context is retained is for the duration a user is active on the system - the definition of what active means is usually defined in the security policy for a system.

Application context is associated with a users use of a specific application or group of applications. The duration that application context information is retained varies widely depending on what the application is designed to do.

This "memory" of what has gone on before clearly requires that we store information somewhere, and the easiest place to store the information is on the server we connected to. We have also determined that for scalability, we would like to be able to load balance requests across all the servers we have that could satisfy the request. This includes subsequent requests from the same user.

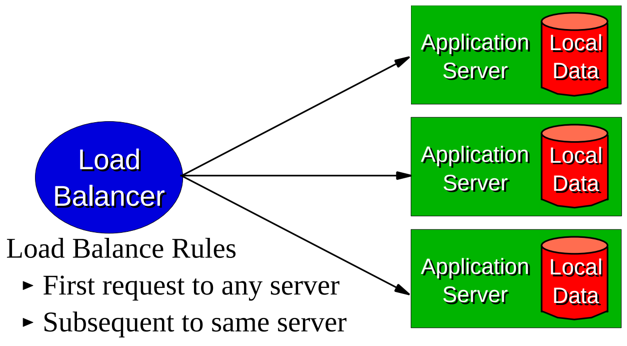
If we store the information we want to "remember" in the server that handles the request, we create an affinity for that user in that specific server. The reason being that our other servers do not know about this stored data and may not be able to complete a subsequent request- this can also be referred to as data affinity.

In the case where we want to store some information we have two design points to consider.

1. Where the data is stored where only the initial server can access it.
2. Where the data is stored in such a way that all servers can access it.

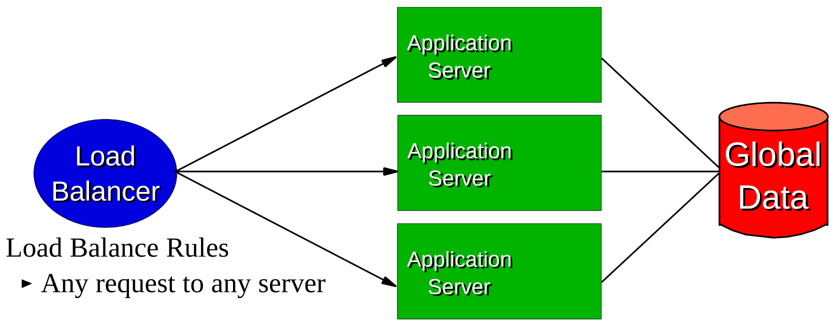
In the case of local storage on only the initial server, the usual benefit is that subsequent requests can be handled with the fastest possible performance, as this stored information could be held in the server's memory.

The drawback is that we must send all subsequent requests to the same server, which impacts our capability to load balance the workload. This also means that our load balancing must have the capability to detect subsequent requests and direct them to the same server.



Optimal Performance - data local to application server

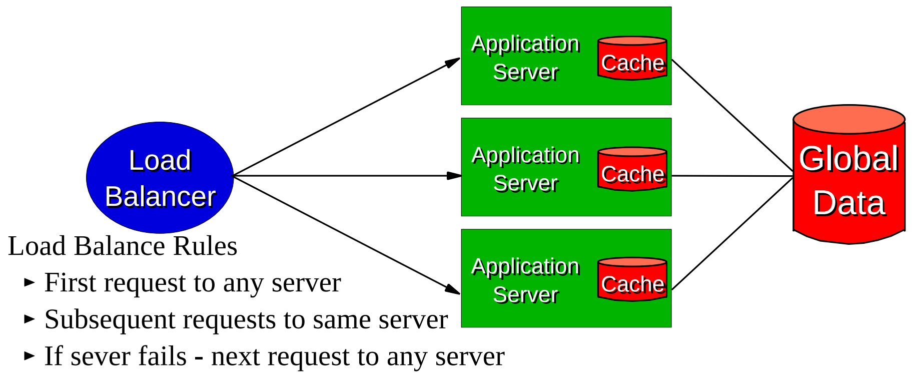
In the case of information stored so all servers can access it. The usual benefit is that we can direct requests to any of the available servers, thus maximizing the scalability. However, the trade off is that we have to get the information from the storage area, in most solutions this would be a relational database, which will perform more slowly than if the information was held locally.



Optimal Scalability - data available to all application servers

When availability is considered, the option of holding information only in the application server is not viable. If the server fails that information is lost, the end user must resupply the information - which is not a desirable characteristic! However, if we just do the global data approach, the performance of the application will suffer and may not be adequate.

To address these considerations it is possible to do a combination approach, were there is a central repository with a cached copy of the relevant data on the application server. In the case were everything is working, the first request is routed to any server, information is stored in the cache and in the global repository. Subsequent requests are routed to the same server, which uses the cached copy of the data, any changes are passed to the global repository. If the application server fails, the application server that gets the next request, retrieves the stored information from the global store and then caches it for subsequent requests.



Load Balance Rules
- First request to any server
- Subsequent requests to same server
- If sever fails - next request to any server

Optimal Performance and Availability - local cache with global data

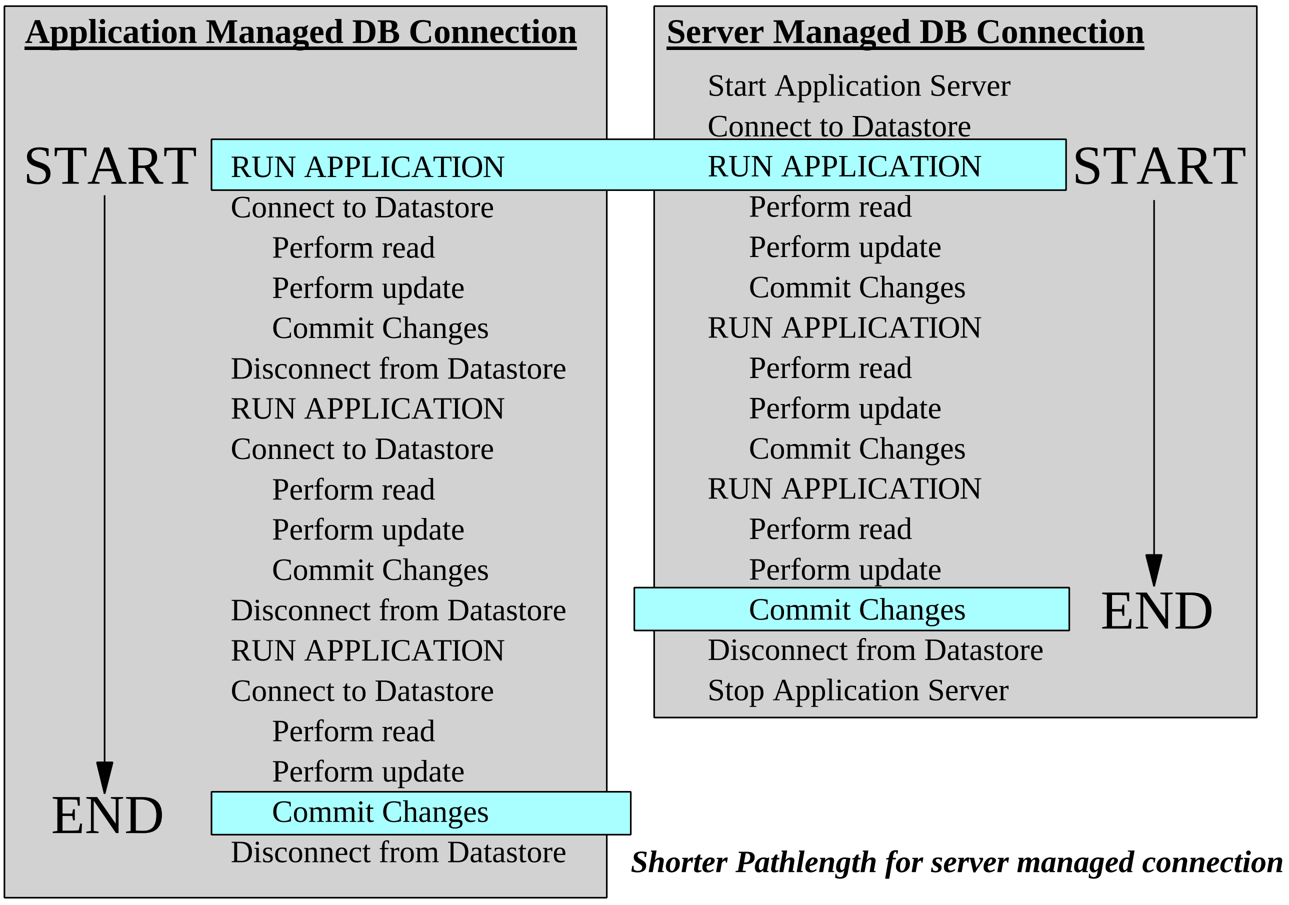
Note that there is still a trade off here. The linear scaling is compromised by the fact that we send subsequent requests back to the same server. However, this impact is likely to be low if the applications provided on the servers are all the same, or are of a similar workload. The greater the mix of workload, the greater the prospect for an imbalance in the load balancing.

Connecting the Business Logic to the Data

There are two possible ways for business logic to gain access to data. The first is that it can directly access the data storage system, for example a relational database. The second is that it can make a request of another business systems, with its own business logic, to get data on its behalf. The second of these falls into the area which is popularly referred to as Application Integration and will be dealt with in a separate section. This section will focus on the case where the business logic is connecting to the data source directly.

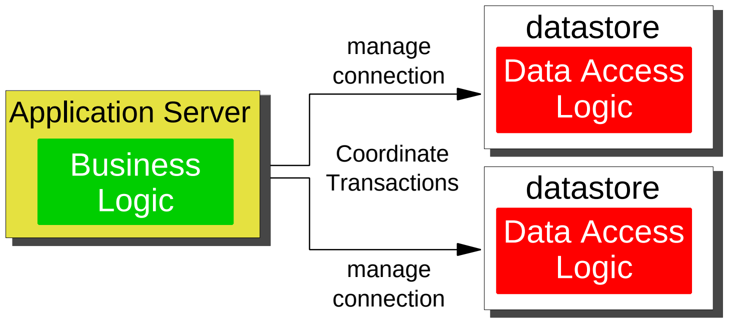
The section on the role of an application server (page 17
) indicated that
the application server should manage the connection to the datastore as well as provide a coordination capability to support update of data in the datastore.

The connection to the datastore is required to optimize the performance of the application by reducing the number of times a connection has to be established. Consider the time required to execute an application three times. A summary of the steps for application managed database connection versus application server managed connection is shown below:



The server managed case does not have to perform the connection an disconnection steps, hence has a shorter path, hence executes in a shorter period of time.

The coordination capability is very important where there is more than one datastore being accessed by the application server. The application server acts as a transaction coordinator. This coordination ensures that all of the datastores participating in the transaction commit all of the updates, or in the event of a failure, back out any updates. This is a local transaction as there is only one transaction coordinator.



Local Transaction Coordination of Two Datastores

The connection of the business logic tier to the datastore tier is usually accomplished using the client of the datastore, and ideally should utilize a standard protocol that supports transactional coordination. The most common standard for this is an XA connection although it is also possible for application server and database server providers to supply more proprietary connection protocols.

If the application server does not have transaction coordination capability, then the system will depend on the data store for the transaction coordination, or another application server that does have the capability. These non transactional or simple application servers should only access one other data store or application server if update of data is required.

Physical Placement of the Datastore

There are two choices in where to deploy the datastore.

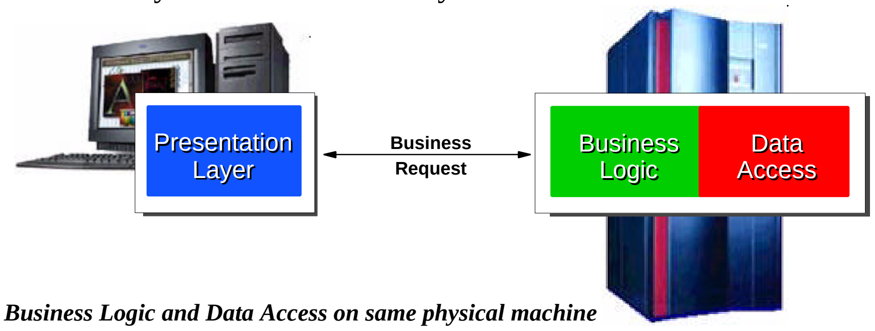
1. Deploy on the same machine as the application server
2. Deploy on a separate machine from the application server

These two choices should not effect the building of the application, that is the design and development, so long as the interface selected between the application server and the datastore is the same regardless of the choice.

Where the choice matters is in the deployment, or the running, of the application, where a trade off between the total possible user community and the performance of the application will be made.

In the case where the application server and datastore are located on the same machine, the access of the data can be achieved using cross memory techniques, rather than cross network. This provides the fastest possible access to the data.

However, the trade off is that we are limited in the number of users we can connect. This limit is the number of users that can be connected to one machine. We are also limited in the number of business logic processes we can have executing at one time. This again limits the number of users that can concurrently access our business system.

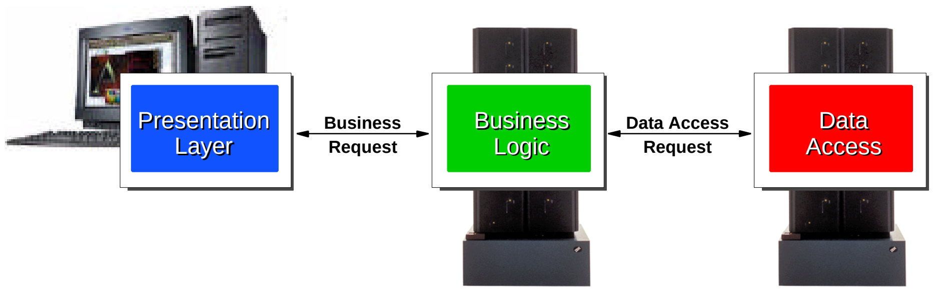


Business Logic and Data Access on same physical machine

In the case of the business logic and the data access on separate machines the connection between the application server and the datastore is now a network connection of some type. This is inherently slower than a cross memory request and thus the performance of our application will be degraded versus the same machine approach.

However, we can have the same number of application servers as we had connected users in single machine case, and each of these application servers can support many users (the number depending on the server capability), where the total number could be in the millions.

Thus the separate machine approach dramatically improves the scalability of the system while degrading the performance. However, by using the fastest interconnection possible between the machines the degradation can be quite small.



Business Logic and Data Access on separate physical machines

In general the same machine approach should only be considered where the size of the user community is well understood - where the size is unknown, for example in Internet base applications, the separate server approach is mandatory!

The Ultimate Limitation of Three Tier Application Architecture

In a three tier application architecture the limit of the scalability is of course to datastore used to hold the business information where this is a single datastore. The single datastore is desirable as it is the easiest to manage and does not require data synchronization with replicated copies.

However, whereas the business logic can be replicated over as many hardware servers as required, essentially providing infinite scalability, the data store can not be scaled infinitely without either partitioning of the data or replicating the data over multiple stores - you can build large sites without getting to this point - but for high volume, large number of users, with change of state of the data applications, this requirement will eventually have to be faced. It will also have to be faced should a disaster recovery strategy call for two geographically dispersed sites with take over capability. Ideally the application should run the datastores on the highest performing platform before taking one of the two actions mentioned.

At present the largest database you can get is DB2 running on OS/390. (The idea of this entire chapter was to not mention any product names -- but I just could not resist in this one case!)

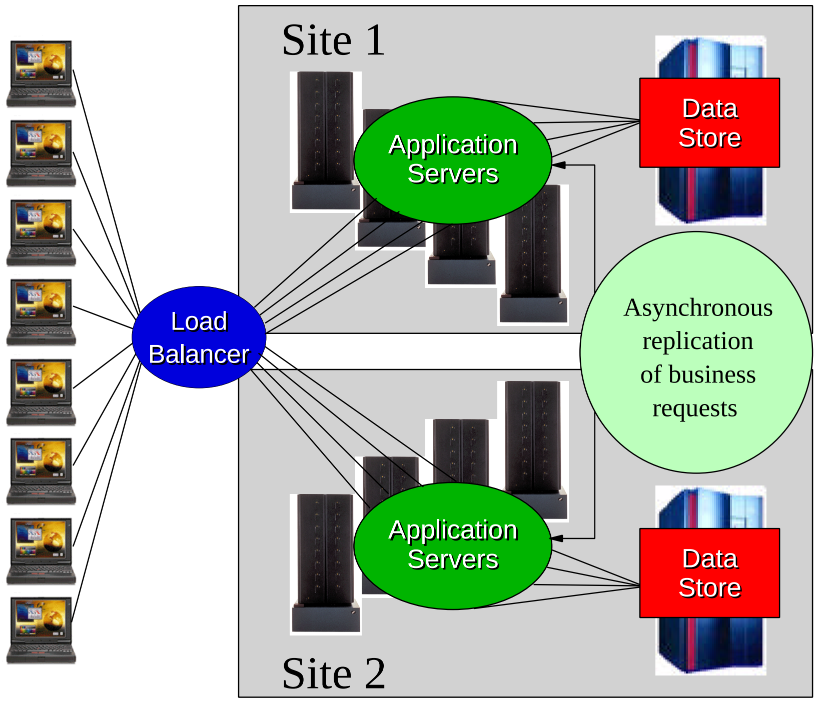
The choice of replication versus partition of data is one of a trade off between optimal performance of the datastore versus ease of management of the data. Partitioning gives the highest performance, but increases the management challenge. Replication eases the management, but decreases the performance as well as requiring synchronization of data to be managed.

If duplicate dispersed sites are required, for example to cater for almost immediate disaster recovery, then replication of the data will be required. In this case one may be tempted to choose replication by itself, and if adequate performance can be achieved then that is all that is required. With this option, the replication of the data can be achieved in one of two ways:

i) Synchronous update - ensures synchronization of data but incurs a severe performance penalty as we have to effectively run a duplicate data operation.

ii) Asynchronous update - the data stores will not be synchronized when the initial update occurs, thus synchronization must be managed and although the replicated update can be achieved very quickly afterwards, the system design must cater for a longer delay. The mainline update is minimally impacted from a performance viewpoint.

The optimal approach is the asynchronous approach, unless the business rules are such that it is not allowed for the two data stores to be out of step at any point in time - in reality very few business systems have this requirement. The asynchronous update can be achieved by either the replication of the datastore as updates occur, using the datastores capability (assuming it has one) to do this, or by replicating the business request and flowing it to the alternate data store and application server to handle the request. The later approach will provide better performance characteristics under load as the data stores just have to be concerned with single updates and an asynchronous messaging subsystem can be used to replicate the business request and hence the data. This also provides a capability to provide an audit trail for the updates should this be necessary.

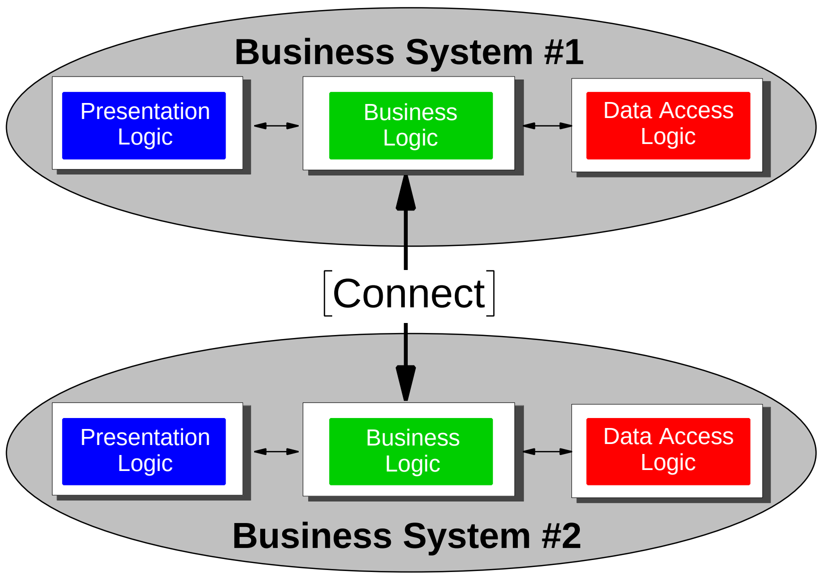


Replication of a three tier application architecture

This section highlighted the benefit of using an asynchronous approach to the replication of data stores. In the example given, business requests were replicated from one application server to another. While in this case this is to facilitate data replication, it is also an example of the integration of application servers and is the basis for Business Integration.

Connecting Application Servers - Business Integration

In a three tier application architecture, the business logic resides in an application server of some type. It is the business logic that defines the business process for the specific application and therefore when we talk about integrating business processes, we are really talking about connecting application servers together - where the application server can be of any type.



Integration of Business Systems Requires Connection of Application Servers

Application servers can be connected using either synchronous or asynchronous communication methods.

Synchronous Communication

Synchronous communication is like two people having a telephone conversation, both sides are available to communicate (talk) and do so until a satisfactory conclusion is reached (OK I know that some telephone conversations don't have a satisfactory conclusion - but then that case is analogous to where one of the side's of the application fails while we are communicating!).

Systems that communicate synchronously have two modes for the communication. The first is like the telephone conversation, one side will call the other and then wait for a reply and then each side takes it in turn to communicate - in communications this is called half-duplex. The other mode, called full-duplex, is where both sides communicate simultaneously, something not possible for human phone calls! From the perspective a single application process it is also not possible to communicate in full-duplex. Systems that achieve this have a pair of application processes functioning in half-duplex mode. So our synchronous communication is actually made up of one to many request and wait for reply pairs.

Asynchronous Communication

Asynchronous communication is like the case of calling someone on the telephone and getting their voice mail system. In this case you have to leave a message if you want to communicate and you make an assumption that the owner of the voice mail will actually check that they have a message and then answer it. There is also the assumption that we have provided enough information to answer it!

While my analogy of asynchronous communication does describe a model of asynchronous communication, it is not quite good enough for robust communication. The reason is that in order to leave the message I have to be able to talk to the voice mail system, but what if that system is unavailable?

Ideally asynchronous communication should provide a way for me to leave the message in my own system and have that message forwarded when the other system is available. This capability is called Store and Forward without having to wait for a reply.

Choosing Between Synchronous and Asynchronous Communication

In connecting application servers together the Store and Forward capability is **ALWAYS** required unless the systems being integrated have identical availability characteristics. So in my phone example, store and forward (voice mail) is required unless the two parties are always available to take calls with one another at exactly the same times of the day. In practice this is not possible unless unrealistic restrictive arrangements are made. From that last sentence you should have inferred that I believe that, store and forward is always required when connecting application servers of different business systems. For application servers in the same business system, where there is one tightly controlling organization, then it may be possible to do without store and forward - maybe!

So synchronous communication is request and wait for reply and asynchronous communication is store and forward without having to wait for a reply.

Suppose, that in my store and forward case the next thing I do is actually wait for a reply from the opposite systems store and forward mechanism. What is the difference between that and synchronous communication? From a programming style point of view there is no difference as this provides the request and wait for a reply model.

So why do we need synchronous communication at all? Good question!

There is one fundamental difference between the two models and that difference is all to do with transactional integrity during the update of data. You should therefore note that in applications that query information with no updates, there is no difference between synchronous and asynchronous for request and wait for reply requirements.

So what about update of information?

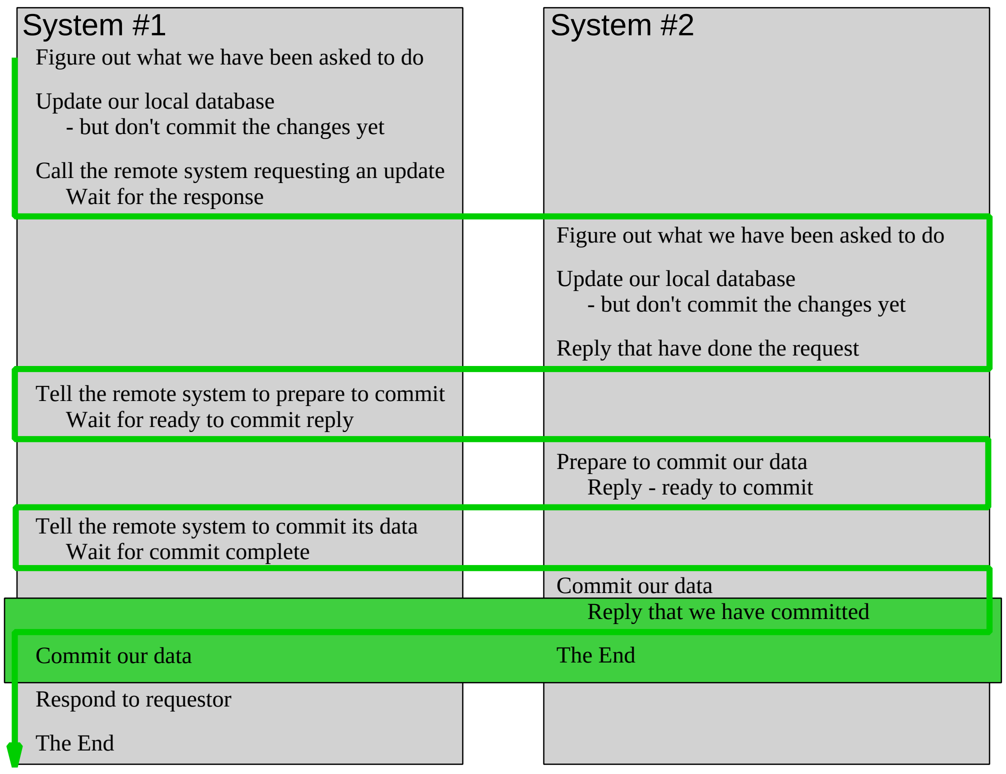
Comparison of Synchronous and Asynchronous in Two System Update

Let’s use an example of communicating between two systems, where the objective is to update a database in each of the systems. We will make one further requirement that both systems must be updated or both systems must not be updated.

For those of you who read though the appendix on Transactions this probably sounds like a requirement for a distributed transaction - but just hold that thought for the moment!

To achieve our goal. The first system must receive a request of some sort. It will interpret the request and update a local database as well as sending a request to the second system asking for it to update its database. We need some way to make sure that both updates have occurred and then we will respond to the requester that we have finished.

In the case of pure synchronous communication the most robust way of achieving our goal is to use a distributed transaction with a two phase commit of the two databases. This will produce a flow of communication that would be as follows (assuming that we are successful in our updates):

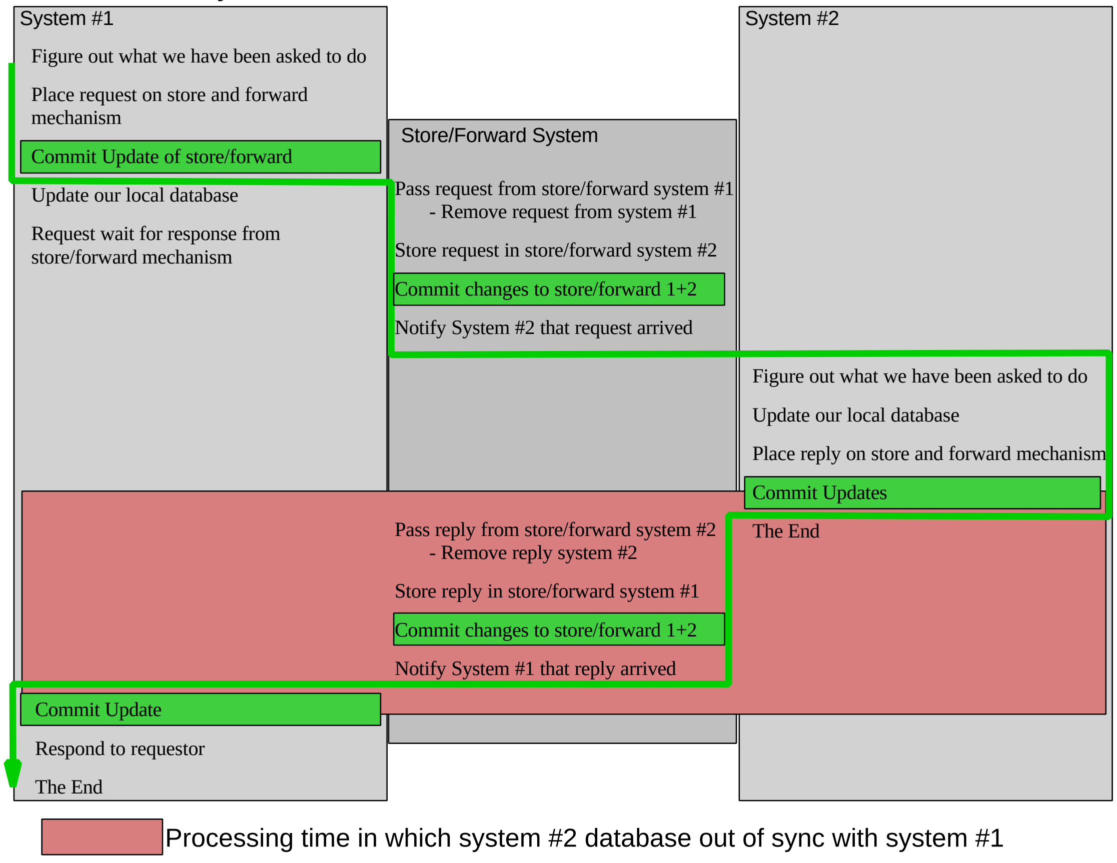

System #1
Figure out what we have been asked to do
Update our local database
- but don't commit the changes yet
Call the remote system requesting an update
Wait for the response
System #2
Figure out what we have been asked to do
Update our local database
- but don't commit the changes yet
Reply that have done the request
Tell the remote system to prepare to commit
Wait for ready to commit reply
Prepare to commit our data
Reply - ready to commit
Tell the remote system to commit its data
Wait for commit complete
Commit our data
Reply that we have committed
Commit our data
The End
Respond to requestor
The End

The key thing about this flow is that it is all one transaction and that the actual commitment of the data occurs right at the end. Up to the point where our processing enters the shaded box, if we had a failure of some kind all of the updates can be backed out. The shaded box represents the window of uncertainty that always exists when you update two separate systems. This approach really minimizes the duration of this window - but due to the fact that communication has to obey the laws of physics, in particular that communication is not instantaneous due to the maximum speed being that of light. Hence there must be a delay between the actual commitment of the update in system #2 and the acknowledgment of that commitment reaching system #1. In that delay period, a failure in the network may mean that system #2 has actually committed the data, but system #1 has not seen the reply. System #1 now has a problem in how to decide whether to commit or rollback its own updates.

In an ideal system, system #1 would wait, holding all the pending updates, for the network to come back and then decide what to do. However, there is no way for system #1 one to know when this might happen and it is usually not practical to hold database update locks indefinitely as other applications will need the data. As a result, system #1 is usually designed to take either the commit or the rollback option when the failure is detected. The most common choice is to roll back the updates, this is referred to as a presumed abort commitment strategy. Of course system #2 may have committed in which case we have inconsistent data, namely system #2 updated, system #1 not updated. A very unlikely outcome, but possible.

So with synchronous communication we ensure that both systems are updated by doing all the update commitment as part of one transaction. In the event of failure our transaction manager, provided by our application server, can issue a rollback request to all the datastores involved.

So what about the asynchronous alternative?


System #1
Figure out what we have been asked to do
Place request on store and forward mechanism
Commit Update of store/forward
Update our local database
Request wait for response from store/forward mechanism
Commit Update
Respond to requestor
The End
Store/Forward System
Pass request from store/forward system #1 - Remove request from system #1
Store request in store/forward system #2
Commit changes to store/forward 1+2
Notify System #2 that request arrived
Pass reply from store/forward system #2 - Remove reply system #2
Store reply in store/forward system #1
Commit changes to store/forward 1+2
Notify System #1 that reply arrived
System #2
Figure out what we have been asked to do
Update our local database
Place reply on store and forward mechanism
Commit Updates
The End
🟥 Processing time in which system #2 database out of sync with system #1

In the above diagram33
33
 the asynchronous flow is shown and we have an additional system to think about, namely the store and forward system. While the end result of the flow is the same as in the synchronous case, the state of the data along the way is very different.

In order to ensure that we will perform the update on the second system, the store and forward mechanism must make sure that the information is passed from system #1 to system #2 and is only passed once. We do not want to do two updates! To do this the store and forward system must have a transactional capability for passing the request from one system to the other, so that it is either on the first system or on the second system. What we can not allow is for the request to end up on both systems or neither of the systems.

The flow shown makes an asynchronous request of system #2 before updating its own database. This allows us a hold an update lock on system #1 while the system #2 update is being performed. This is to minimize the time that the two systems will be out of synchronization. We could do the update before making the request. This could be done where the design of the application is such that the time between the updates of the two systems is not critical - as long as both eventually happen!

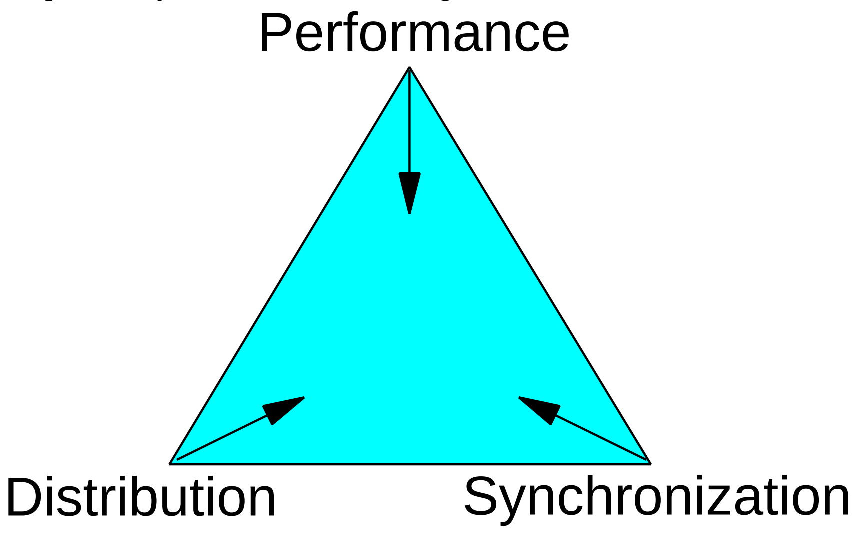
It should also be noted that five transactions are executed in this process versus the one transaction that we had in the purely synchronous case. Therefore one might conclude that we will take more processing time with this asynchronous alternative. This is true in the case where we are trying to limit the time difference in updating the two systems. If we are not concerned about that, then from System #1's point of view the asynchronous approach is actually faster as we don't have to wait for System #2 to complete its update.

So the trade off in asynchronous versus synchronous update is as follows:

Asynchronous
- Complete separation of the availability profiles of the two systems
- Faster update of first system if not concerned about time difference in the two updates
- Systems will be out of synchronization for small period of time

Synchronous
- Requires identical availability profiles of the two systems
- Faster update of both systems where application is time sensitive to synchronization
- Systems not out of synchronization at any time, but possible in doubt situation

I can represent this pictorially in the form of a triangle, shown below:



This is a general principle in the design of distributed systems. You can optimize your system anywhere within the triangle of, Performance (how fast does your system run), Distribution (how far apart are you systems), and Synchronization (how close together in time are updates), the closer you get to each apex the more optimal each aspect. In general you can optimize for any two of the three, but not all three. I have never found a system that optimizes for poor performance - so you get to choose distribution versus synchronization. Business to Business communication is inherently distributed - so there goes synchronization!

So if we really want to design our applications such that the updates of the two systems do not have to be time synchronized, there is absolutely no requirement for synchronous update. The time synchronization requirement must come from the business requirements of the application. In my experience there are very few (if any!) business processes that have an absolute time synchronization requirement.

My conclusion is that asynchronous intercommunication should always used for integrating business systems (application servers).

This does apply a greater design burden on the application developer as they have to consider the impact of time synchronization in the asynchronous case. However, the run time benefits are considerable as we can now operate our systems completely independently if required. This independence is almost always mandatory in business to business integration between to separate businesses.

So to integrate business systems we need a store and forward capability to pass requests between the systems to be integrated. However, we have the question of how do we connect the business system to the store and forward mechanism. This is particularly important where we want to integrate a system that we have purchased and do not have any source code for it!

Connecting Application Servers to a Store and Forward Mechanism

There are two ways of connecting an application server, with its associated applications, to the store and forward mechanism. Invasive insertion of the chosen mechanism into the application and passive adaptation of an existing application interface

Invasive Insertion of the Chosen Communication Interface

This approach requires that you have access to the source code of the application, hence is generally not applicable to cases where a packaged application is purchased. It is also time consuming, as we must retest the application to ensure that the changes we have made do not introduce errors into the application.

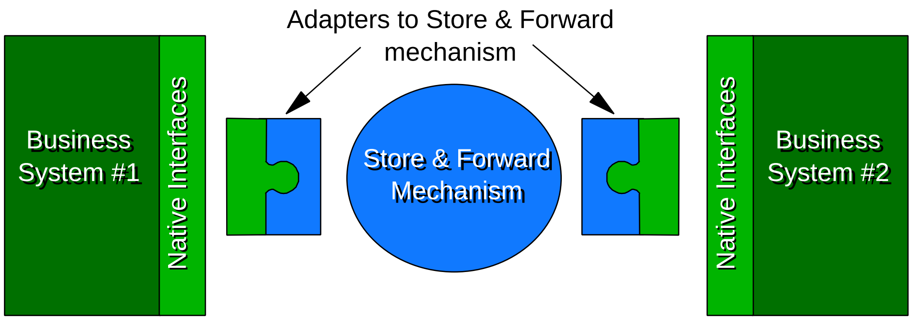
In general this is used in cases where there is an established integration mechanism and a new application is being built that wants to participate in the integration.

Passive Adaptation of Existing Application Interface

This is achieved through the use of an adapter of some type which makes use of a existing interface to the application and then bridges that interface to the chosen communication interface.
If the application already supports the chosen communication interface, then the adapter is very straight forward, and really just an extension of the application. However, adapters can also be very complex, particularly if the only way to access the application is through some sort of screen interface, where we might have to build a "screen scraping" adapter.

In general this approach is faster to build than the invasive one, as we only have to test the adapter and as a consequence is the preferred approach. In fact it can be argued that the invasive approach is really the creation of an adapter that is being inserted into the application code. The trade off being that passive technique is usually simpler to build, whereas the invasive approach may well provide for more efficient performance.

So the conclusion is that adapters, either passive or invasive are required to connect our applications to the chosen integration communication mechanism.
This can be represented as shown in the diagram below.



Once we have our adapter then comes the question of where to place it.

We could in theory place the adapter on a system remote to the application, if we have a native interface that can operate remotely. The benefit of this is that our store and forward mechanism can be restricted in its platform choice and we can concentrate the skills required to build adapters on a small number of platforms. This apparent benefit is far outweighed by the drawbacks associated with the implied requirement to run multiple communication protocols to make the adapters work. This makes the operations side of the system complex and hard to monitor. In addition there is the base assumption that we have an interface that is remotely accessible - in general the only remote interface that you can count on is the one that an end-user would use (and not always then!) which implies screen scraping - a relatively difficult approach to integration.

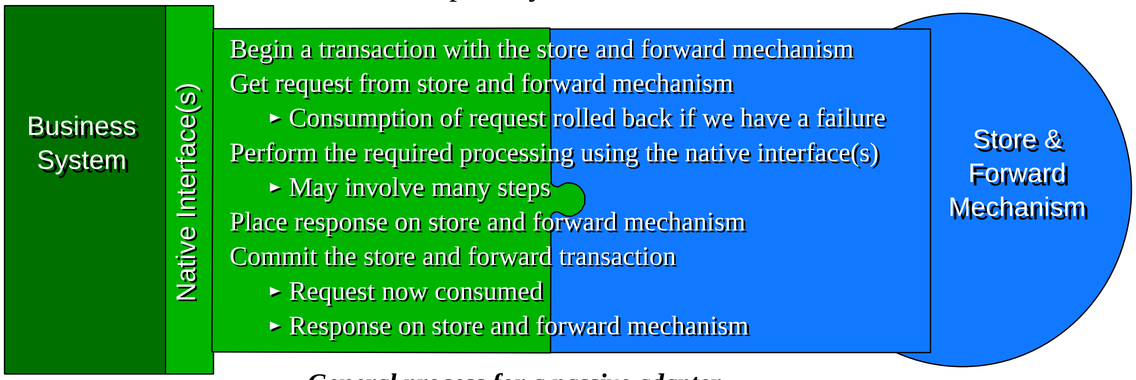
In practice the better choice is to place the adapter on the same system as the application. This means we may be able to use one protocol which we can then monitor and manage much more easily. In the case of packaged applications the application vendor may well provide an adapter for standard communication mechanisms. These supplied adapters will always be on the application server side.

This does of course mean that our chosen store and forward mechanism must be available on all of the platforms we want to integrate - ideally our selection should be for the most pervasive store and forward mechanism. The industry’s most pervasive store and forward mechanism is MQSeries. (Yes I know, I mentioned a product name again!!)

The Role of the Adapter

So the conclusion is that we want an adapter to connect our application to the store and forward mechanism. This connection needs to be very robust, particularly if we are going to provide updates to the application and need to ensure that they occur. Ideally the adapter should allow for transactional update of the application where the request passed by the store and forward mechanism is consumed as part of the transaction. If we get a failure the request is not consumed and we can retry. This presents a major problem for application integration, as the vast majority of passive adapters can not participate within a transaction in the target application. If they can then no problem, if not then a compromise design can be deployed in developing the adapter.

The required processing flow is shown below. The assumptions are that our store and forward mechanism has a transactional capability that the adapter can use and that our target system does not have an externalized transaction capability.


Business System
Native Interface(s)
Begin a transaction with the store and forward mechanism
Get request from store and forward mechanism
▸ Consumption of request rolled back if we have a failure
Perform the required processing using the native interface(s)
▸ May involve many steps
Place response on store and forward mechanism
Commit the store and forward transaction
▸ Request now consumed
▸ Response on store and forward mechanism
Store & Forward Mechanism

General process for a passive adapter

The processing occurs under the transactional scope of the Store and Forward mechanism ensuring that we can not loose the request in the event of a system failure. The placement of the response on the store and forward mechanism represents successful completion of our request. It should be noted that there is a window of opportunity between the completion of our perform required processing and the placement of a response on the store and forward mechanism, where in the event of a failure, recovery of the store and forward transaction will not allow us to determine if we completed the update on the business system - this window is extremely small, but should be considered in the design of the system.

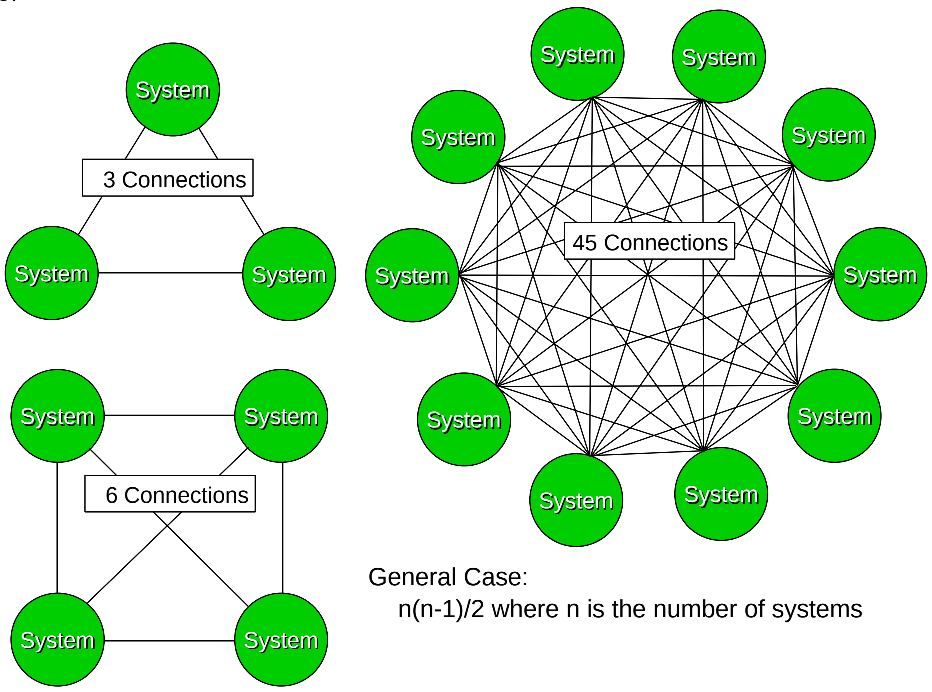
So the primary role of an adapter can be considered to be to make sure that we do what we have been asked to do and get a response back.

So what else could the adapter do?

Clearly under the mantra of "*simple matter of programming*" the adapter could do just about anything. However, the two areas most often considered are routing of the request in the store and forward mechanism and transforming the information gathered from the application into a form the requesting system requires.

In the case of routing, the store and forward adapter clearly needs to be able to locate at least one other system, or else we are not going to get very far with our request! However, we could enable the adapter to locate all possible systems and allow it to communicate directly. This point to point connection allows for communication with the least network latency. In order to enable this we would have to provide a directory for the adapter's use that contained the entire system topology that all system adapters could share. This approach has drawbacks such as; finding a directory we could access from all systems; the performance impact of such directory queries for routing; the currency of the information if we cache to improve performance - these drawbacks can be solved by using robust distributed systems services such as found in DCE or lightweight web oriented services such as LDAP based directories.
However, there is one large question which can not be resolved - that is the number of connections produced by a point to point integration strategy.

The next diagram shows the number of connections for three, four and ten, directly connected systems.



Number of Direct Connections for 3, 4 and 10 Integrated Systems

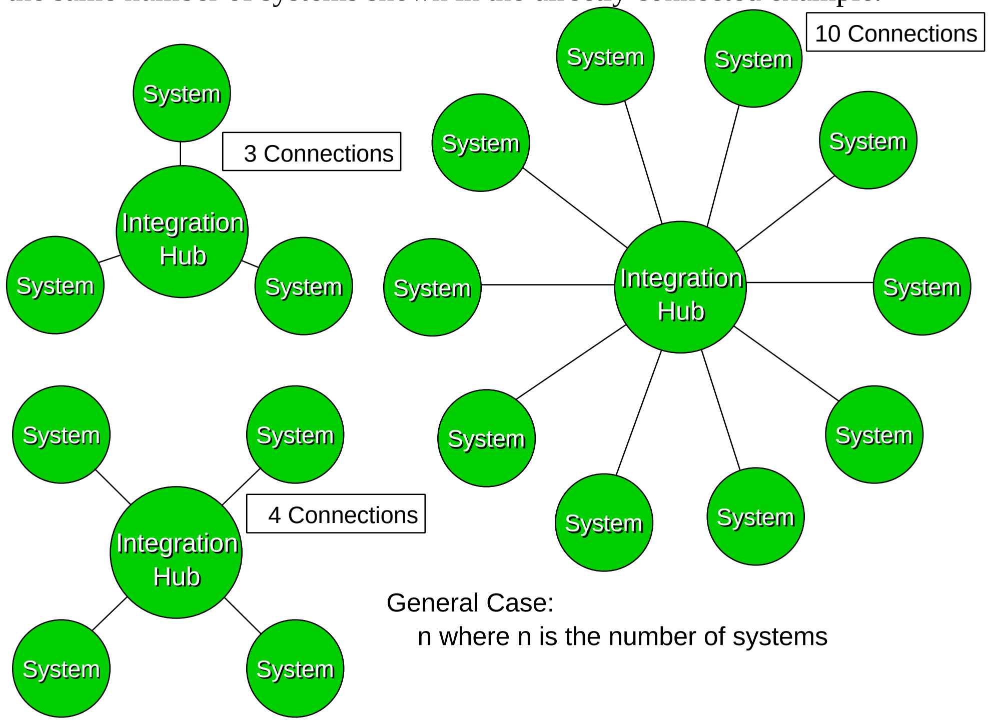
As the number of systems grows the number of connections quickly becomes a management problem - mainly associated with monitoring that all connections are operational.

Hub and Spoke Integration Architecture

A better approach is to adopt a spoke and hub connection architecture, where the systems connect to a central hub, which then make the routing decision. The systems now only need to know where the central hub is (note that this information still has to be distributed to the adapters!) and have one connection each. The hub is in charge of making the routing decision.

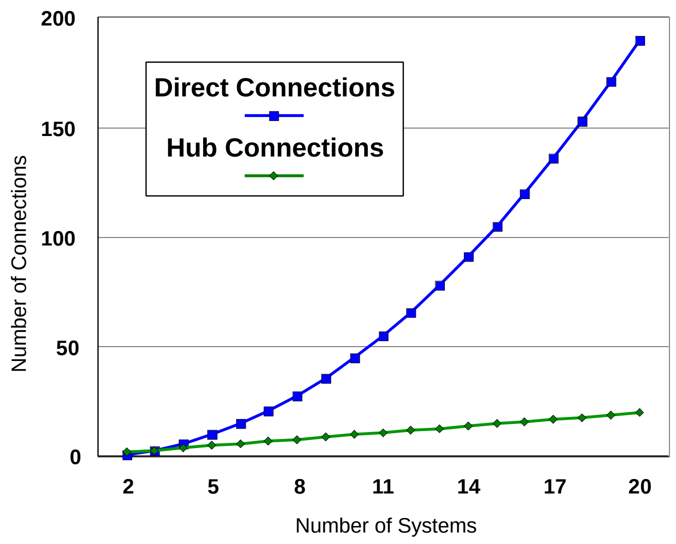
Note that this is the same principle as used in TCP/IP routing, where TCP/IP clients know a router or gateway that then routes all traffic onwards.

The following diagram shows the number of connections in the spoke and hub architecture for the same number of systems shown in the directly connected example.



Number of Hub Connections for 3, 4 and 10 Integrated Systems

A graphical comparison of the increasing complexity of the two approaches is shown below.



Complexity Comparison of Direct Connected Systems versus Hub Connected Systems

Clearly the hub approach becomes increasingly simpler to manage as we increase the systems. The question is at what point should we consider the hub approach, after all, for two system we actually have more connections with the hub!

I believe the answer to this question is **FOUR** systems.

Okay so why did I pick four? Well I am looking at the problem from the point of view of minimizing the number of points of failure. In any set of integrated systems I have to maintain the number of systems plus the number of connections.

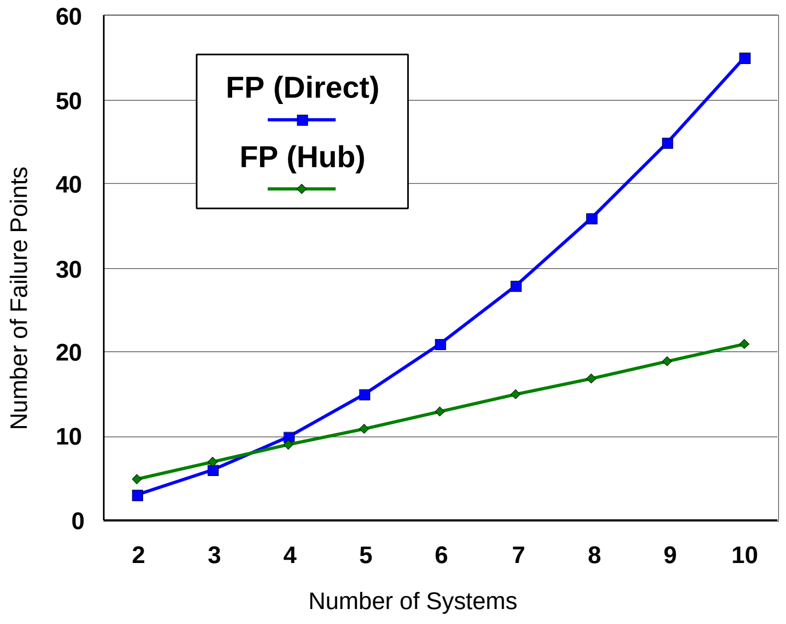
In the direct connection approach the number of failure points (FP) is the number of systems plus n(n-1)/2 number of connections, represented as:

FP = n + n(n-1)/2, where n is the number of systems

In the hub connection approach the number of failure points is the number of systems, plus one for the hub and the same number of connections as there are systems, represented as:

FP = 2n + 1, where n is the number of systems

The graphical representation of these is shown below and the cut over point is between 3 and 4 systems - hence why I picked four.



Failure Point Comparison of Direct Connected Systems versus Hub Connected Systems

Now many of you will no doubt be thinking, "he can't just say that, that is a much too simplistic approach to making that decision. What about the cost of the building the hub versus building the adapters?" Clearly, the economics of building the initial system do come into play, but the longer the system is in use the more money will be spent on monitoring and maintaining the system. Minimizing the complexity and associated failure points is the best way to control this expense and if you run the system long enough the cost benefit of the simpler system outweighs the potential build cost. The point at which this happens is very dependent on how much money is spent in the building - however, if you have four systems, I would argue that given enough time it always happens - and that is why I picked four!

I would also point out that if you are integrating two systems and intend to add at least two more at some point in the future, then you should also consider the hub approach to simplify your future development!

Once we have decided that routing belongs at the center, we then need to consider the transformation of the information that one application will send to another one. In the ideal case we would use a common form of information, however, just as in real life everyone does not speak the same language, systems tend not to have a common form. In addition even if we agree on a common form, and XML appears to be a candidate for such, then we still have to agree on the specific meaning of the phases. It is likely that transformation will be required for many years to come.

So where to do the transformation?

I believe that transformation also belongs in the central hub as this allows for the simplest adapter strategy i.e. connect me to the hub. If I put transform in the hub, I either have to make available all possible transformations or I need to adopt a common format for moving my information.

The first of these two presents a significant systems management problem and is not usually considered a viable option.

The transform to common format could be considered. From the point of view of building the system, the common format and in adapter transform seems attractive, as this allows the business system provider to do all the work their local platform, rather than have to provide a transformation at the central hub to accompany the adapter. However, if we look at the runtime aspects of the systems, particularly if we wish to change our transformation rules, is much simpler to do this at the central hub than to change all of the adapters.

This is the same principle as the management of client/server systems where code needs to be distributed to workstations versus a centrally managed system. For any reasonable number of systems (and I will say four again - just to be consistent!) the centrally managed is the more long term cost effective approach.

My conclusion is that transformation belongs at the hub, assuming you needed a hub for the routing capability.

In fact I would argue that all functionality associated with the management of information flow between integrated systems belongs in the central hub. Other examples of management include; services to publish information from business systems which can be subscribed to by other systems; and the control of which business systems should be invoked to achieve an entire business process or workflow.

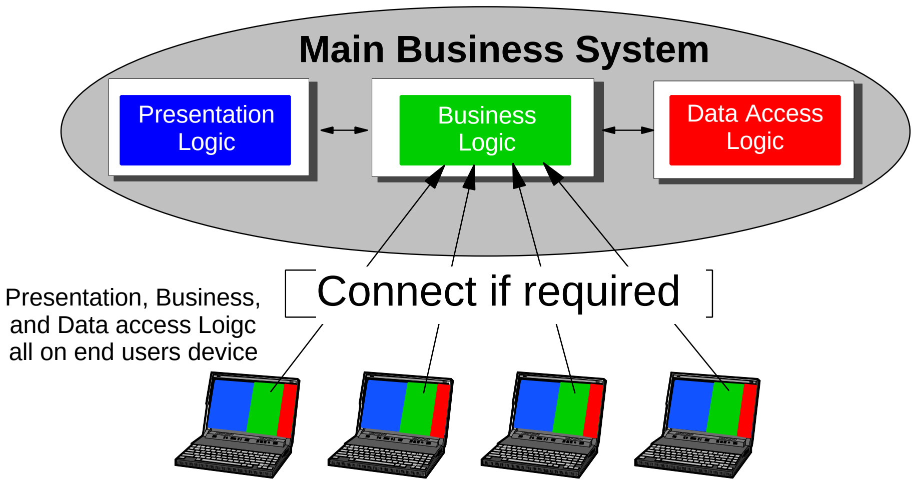
Disconnected or Mobile Users

Before we summarize the integration of business systems it is useful to consider the case where we require to build applications in support of mobile users who are not always connected to a network - i.e. disconnected users.

The principles we have discussed also apply here and it can be viewed that in these cases we are building a three tier application were we choose to deploy presentation, business logic and data access on one device for one user. Furthermore the connection of that device to our other business systems is just an example of application server integration.

However, although we may choose to use the same pervasive store and forward mechanism to connect the "mini-business" systems, it is likely that there will be a specific server set up to handle the inbound request, in which case it acts as a type of first point of contact application server, and thus the client of that server may be selected. This is particularly true where the end user's application provides e-mail and collaborative type applications.

In fact you can think of the disconnected users as having mini-business systems that need to be integrated. This is illustrated below.



Disconnected Clients - mini business systems

This integration from the disconnected client can be at the presentation layer, the business logic layer or the data layer. The choice depends on whether we just need to move completed data from our disconnected user, or whether we need to access some business functionality. It is desirable to have all of the options available.

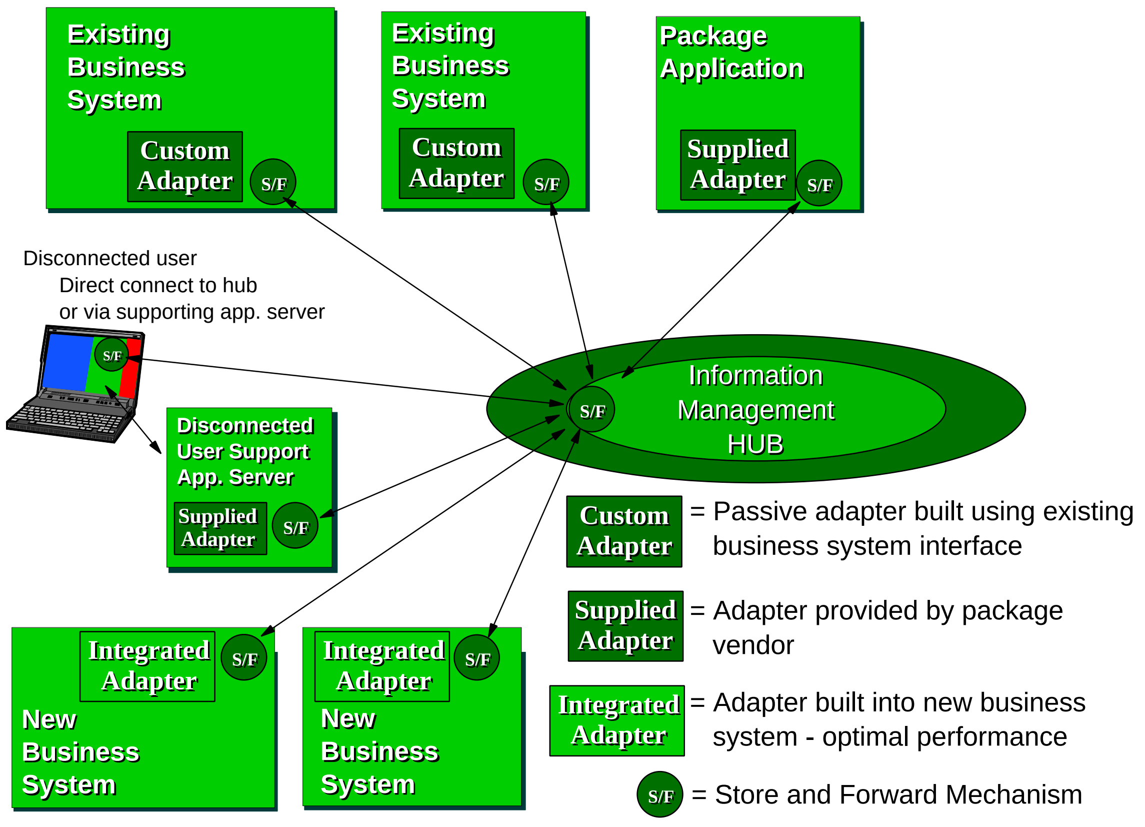
Summary - Connecting Application Servers - Business Integration

Business systems are best integrated using a pervasive asynchronous store and forward mechanism, with a spoke and hub architecture. The only case where this is not true is where there is a business requirement to have point in time synchronization of updates in two or more systems, where a synchronous, distributed transaction approach would be needed. In this case the architecture would be point to point. Very few (any?) business processes have this requirement.

Connection of the business system to the store and forward mechanism should be achieved by use of an adapter, where the adapter's role is to provide transactional (or as close to transactional as we can get it!) connection to the store and forward system.

All management of information flow should be provided by a central hub.

Disconnected or mobile users (mini business systems), can connect directly via the pervasive store and forward mechanism or use a specific application server that supports their specific requirements.
This can be represented as follows:


Existing Business System
Custom Adapter
S/F
Existing Business System
Custom Adapter
S/F
Package Application
Supplied Adapter
S/F
Disconnected user
Direct connect to hub
or via supporting app. server
S/F
Disconnected User Support App. Server
Supplied Adapter
S/F
Information Management HUB
S/F
Integrated Adapter
S/F
New Business System
Integrated Adapter
S/F
New Business System
Custom Adapter = Passive adapter built using existing business system interface
Supplied Adapter = Adapter provided by package vendor
Integrated Adapter = Adapter built into new business system - optimal performance
S/F = Store and Forward Mechanism

Example Spoke and Hub Integration Architecture

Choosing An Architecture for e-business

The previous sections have looked at the major decision points in designing three tier application architectures. The next stage is of course to choose - Oh, and we are still not going to mention product names (well no more than I have already!).

There is a broad industry consensus that there are two main aspects to e-business, namely Business to Consumer (B2C) and Business to Business (B2B).

B2C involves consumers accessing web based systems primarily using browser based devices, although there is a increase in more pervasive devices. However, in nearly all cases these end user interfaces are characterized by being thin client, server driven user interfaces.

B2B involves businesses either doing pretty much the same as a consumer in using server driven user interfaces, or providing business system to business system integration.

Finally there is the traditional systems’ usage, namely the businesses own internal users who may operate similarly to B2C, but also may need client driven user interface capability as well as the mobile disconnected capability. You may even want to build a two tier client server system, however, as that is not readily reusable in an e-business sense, I will exclude it from the discussion.

Aggregating all of these end user types we can see that there are really only four major categories of “user” for our systems.
.
1. Thin Client, Server Driven user interface, connected end users
2. Full client, client driven user interface, connected end users
3. Full client, client driven user interface, disconnected end users
4. Other business systems, the user is the other business system

If we look at the possible application servers we can deploy to act as the first point of contact application server, supporting these users, then there are three basic types:

1. Simple, non transactional application servers

   In this case the assumption is that we are going to build some business logic and manipulate some data. The simple application server has no transactional capability, so if we want to update the data, we can only have one data access connection for the server. We may also use the store and forward mechanism to access other business systems. If we are using the simple application server as a connector for our server side presentation layer we may also invoke a transactional application server if the two types of servers have identical availability profiles.

2. Transactional application servers

   In this case the assumption is again that we are going to build business logic and manipulate some data. Update of multiple sources is possible as we have transaction coordination capability. Access to other existing business processes can be achieved via the store and forward mechanism.

3. Integration hub for business system requests

   In this case the assumption is that we have the business function already, we wish to provide access to it for the requesting system.

Now that we have these pieces of information we can look at the required architecture topologies or sub-patterns that provide the basis for all types of e-business application. The rationale for looking at the sub-patterns is that not everyone needs all of the capability and the master or top level pattern can be thought of as a combination of the sub-patterns illustrated in the next few pages.

There are five sub-patterns, these are

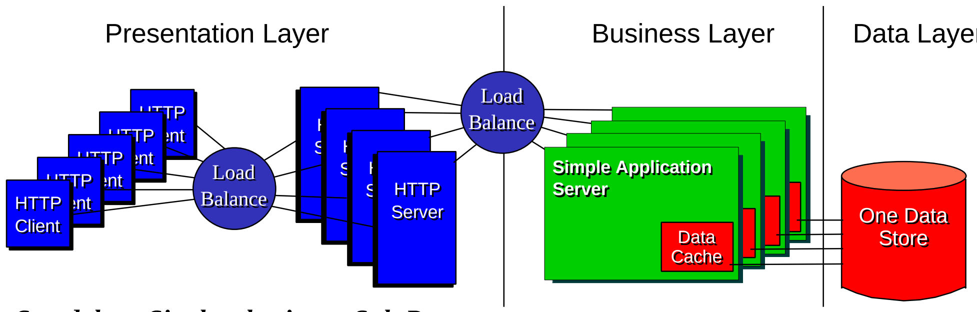
- Standalone Single e-business, Sub-Pattern
- Standalone Multiple e-business, Sub-Pattern
- Integrated Single e-business, Sub-Pattern
- Integrated Multiple e-business, Sub-Pattern
- Integrated Multiple e-business to e-business, Sub-Pattern

For clarity the combination of all five patterns is not shown.

Standalone Single e-business, Sub-Pattern

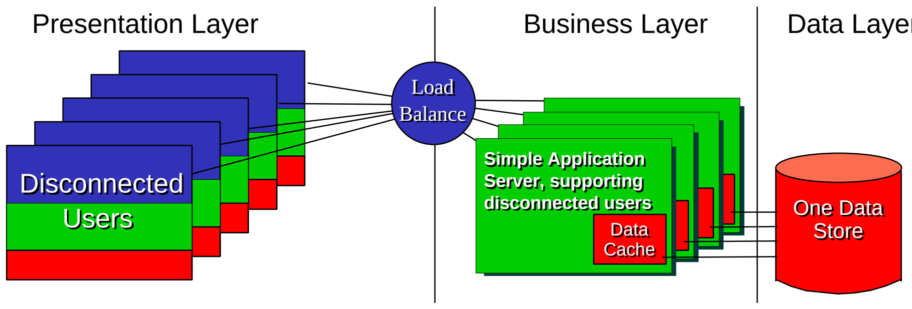
This pattern supports connected end users to a simple application server with one database. This is the common design pattern for standalone web based catalog applications that have no integration to other systems.

The diagram below shows the pattern. In the case shown the pervasive device is a web browser, server combination which could be replaced with any pervasive device and associated device connection server. The load balancing between the browser and the server is achieved by network based load balancing. The load balancing between the HTTP (pervasive device connection server) and the simple application server can either be done by a client of the application server or by a network based load balancing capability. This choice is dependent on the actual application server chosen. The data cache is an optional performance benefit as previously discussed for use in fast query operations. All updates are done directly to the data store, where ideally the datastore should update the cache on behalf of the application server.



Standalone Single e-business, Sub-Pattern

The presentation layer shown could be replaced with a disconnected user application where the application server supporting the disconnected users is non transactional in nature. The pattern, shown below, is the same. It is just the presentation device that now has more capability.

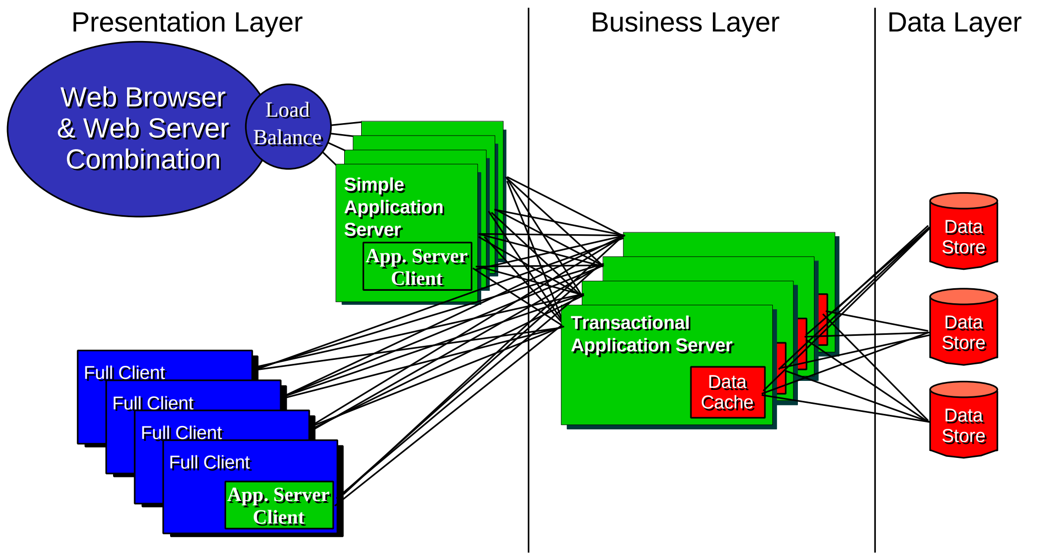


Standalone Single e-business, Sub-Pattern (Disconnected Users)

The load balancing can either be achieved through network based or client based methods, depending on the capability of the disconnected users platform.

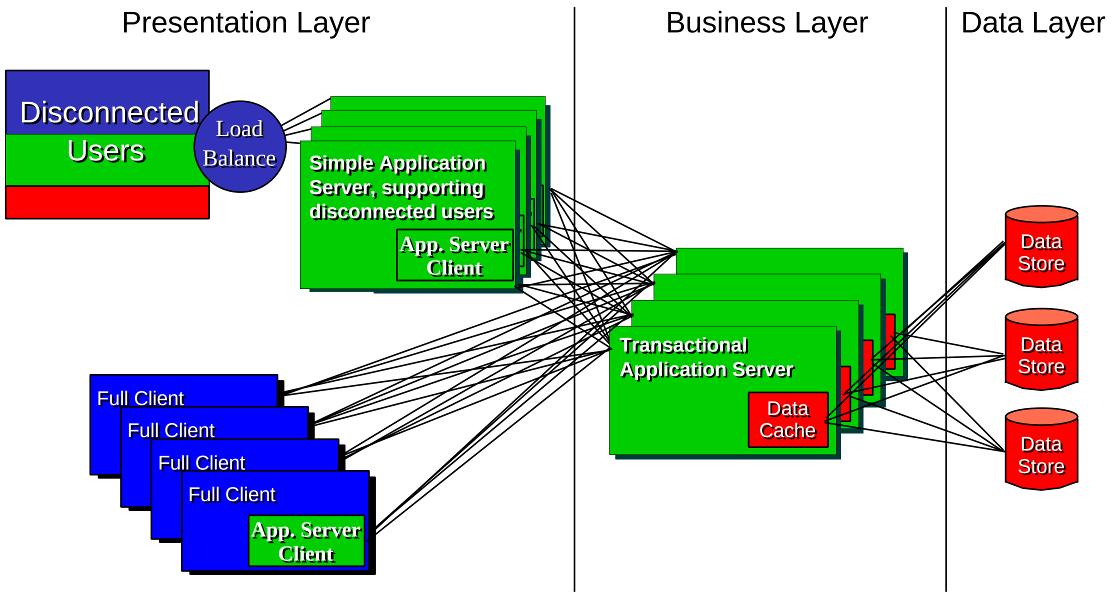
Standalone Multiple e-business, Sub-Pattern

This pattern extends the previous pattern but allows us to add client driven user interface devices as well as additional datastores to be updated.



Standalone Multiple e-business, Sub-Pattern

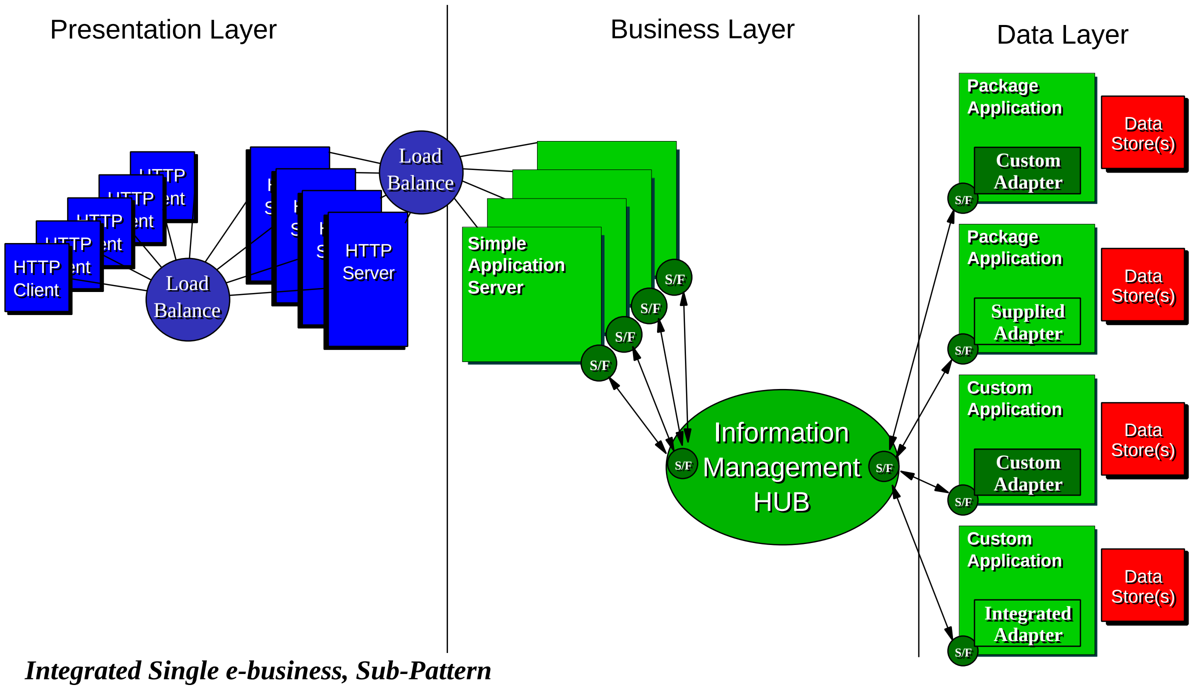
In this case the simple application server of the previous sub-pattern has become an extension of the presentation layer, providing presentation business logic and client access. Load balancing of requests across the transactional application servers is ideally accomplished from the client which optimizes performance. It is possible to adapt this pattern by inserting a network load balancing capability if the clients do not provide it themselves. Once again it is possible to replace the web browser and web server combination with a disconnected user capability.



Standalone Multiple e-business, Sub-Pattern (Disconnected Users)

If the application server supporting the disconnected users has a transactional capability then we can connect the disconnected client directly to the transactional application server. However, this is exactly the same pattern as for the full function clients, where these clients can operate in either connected or disconnected mode (this variation is not shown).

Integrated Single e-business, Sub-Pattern

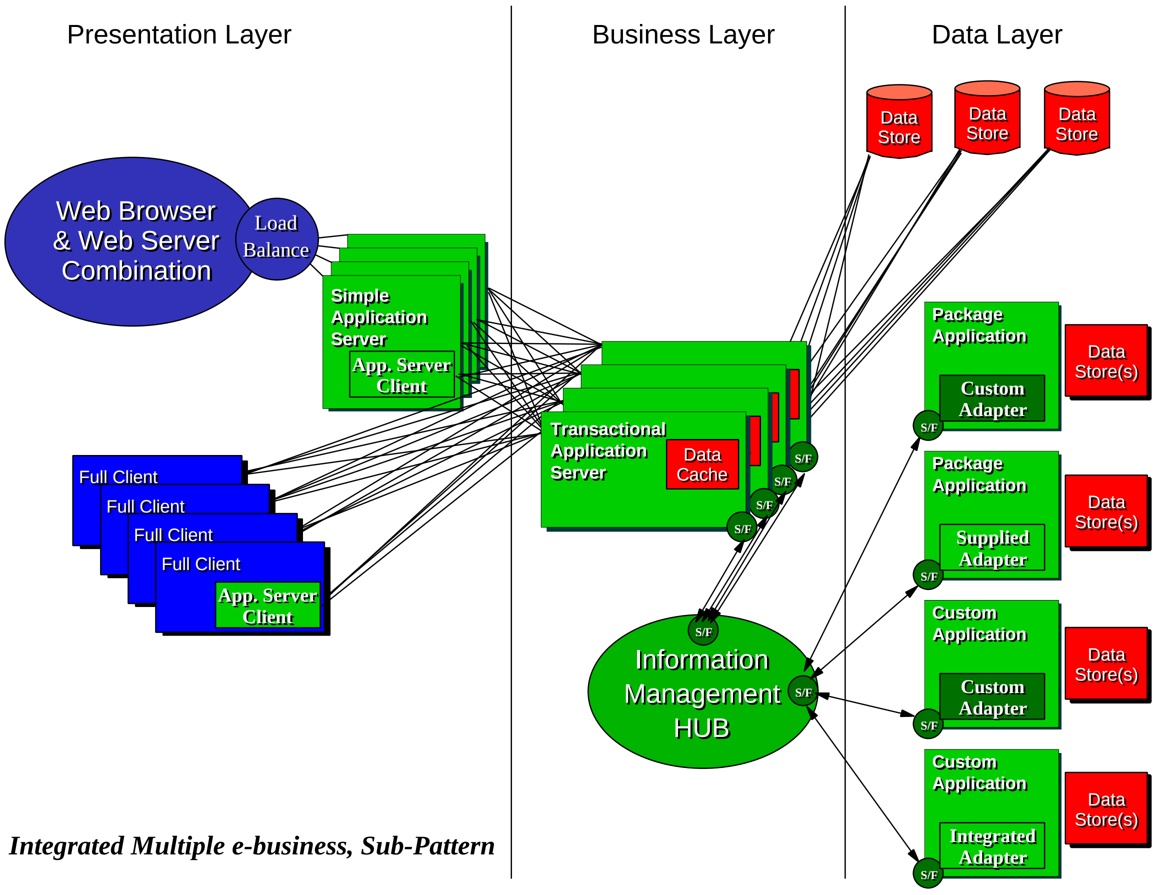


Integrated Single e-business, Sub-Pattern

In cases where we are looking to integrate our simple e-business application with other business systems then this is accomplished by connecting the simple application server to the information management hub (assuming we have three or more additional systems to connect - the simple application server counts as the first of the four!)

This is pattern solves many of the B2C scenarios, where the major requirement is to enable a web based access channel, with most of the primary business functionality available on other systems. Note that no data update is shown on the simple application server as it has no transaction coordination, which in an ideal case would be needed if we updated a datastore as well as the store and forward mechanism. However, this type of multiple update can be accomplished if either the data store or the store and forward mechanism can provide a transactional coordination between themselves.

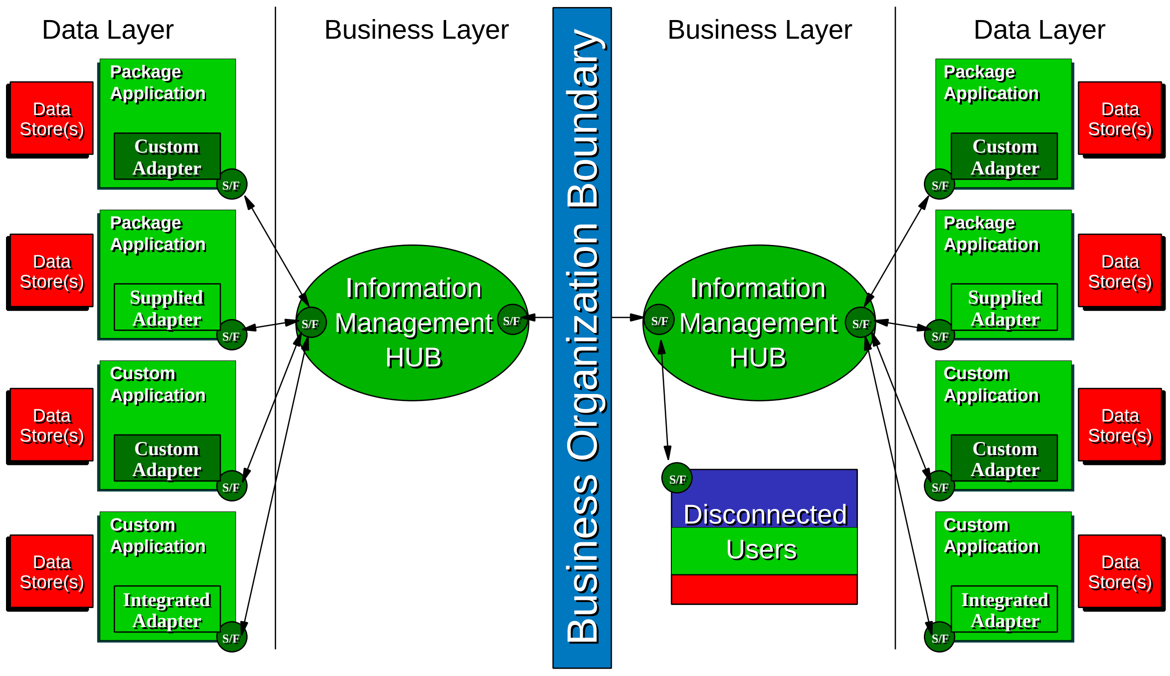
Integrated Multiple e-business, Sub-Pattern



Integrated Multiple e-business, Sub-Pattern

This is the most flexible of the sub-patterns in that it enables any type of client to be connected to applications that can update multiple data stores as well as integrate with multiple additional business systems. Therefore where the business requirements are not as well defined as possible this is the pattern that should be considered for maximum flexibility in support of B2C type applications, or B2B applications that are provided for end users of another company.

Integrated Multiple e-business to e-business, Sub-Pattern



Integrated Multiple e-business to e-business, Sub-Pattern

So far the patterns discussed have all assumed that there is a real end user that wants to gain access to some business function. However, there is also the case of B2B, where business system to business system need to communicate.

The pattern for B2B system integration is shown above, and is simply the connection of two information management hubs across some network boundary, which for example may be a private network or the Internet.

Disconnected users with no requirement for a support server, can also take advantage of this pattern as you can regard the disconnected user as a mini-business system. The most common form of this integration is from within one organization, however, there is nothing to prevent the disconnected users crossing the business organization boundary. However, for a management of information view, the connection to one's own information hub and then subsequent cross organization boundary communication is usually preferred to tightly control the communication between two different entities.

Selecting IBM Products for e-business

In the previous chapter we looked at the pros and cons of various approaches to architecting e-business systems. The outcome of this was five sub-patterns, expressed in product neutral terms. This chapter will look at what IBM products fit the patterns.

Throughout this chapter I will look at what the minimum requirement is for the pattern when looking at the simple application server. This does not mean that a higher level application server could not be used, for example WebSphere Advanced Edition could be used in place of WebSphere Standard Edition to be able to run Java servlets.

Choosing Products for Standalone Single e-business

This pattern requires that we have a web browser, web server combination, with network load balancing of the browser traffic. It also requires that we have a simple application server as well as a datastore.

Required Software	IBM Product Choice
Web Browser	No preference - any web browser will do
Web Server	Any HTTP engine (See Note 1)
Network load balancing	WebSphere Performance Pack
Simple Application Server	WebSphere Standard Edition, Lotus Domino
Datastore	Universal Database (DB2)

Note 1: WebSphere Standard Edition includes the IBM HTTP engine based on Apache. Lotus Domino provides HTTP server capability as part of the product.

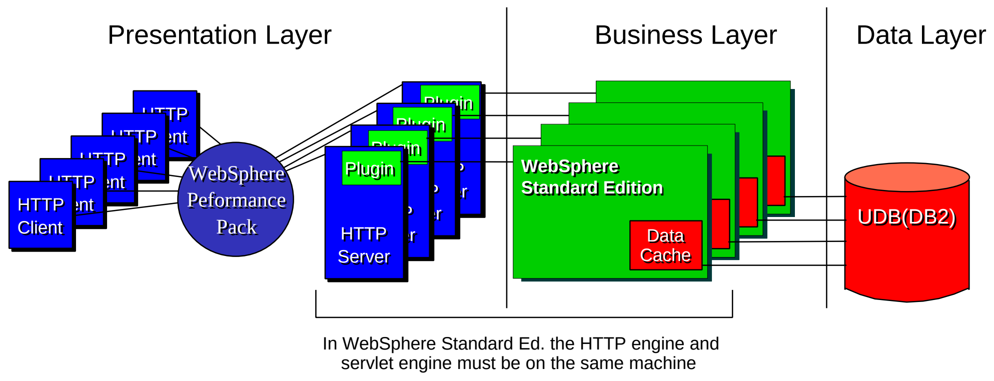
The only area of choice here is in the selection of the simple application server. This choice can be dependent on many factors, including whether you have an investment in one of these servers already. In addition, WebSphere Standard Edition which is a Java Servlet engine can be run with Lotus Domino, with Domino acting as the HTTP engine.

In the case of WebSphere Standard Edition, the servlet engine has to run on the same machine as the HTTP engine. In addition it is not possible to insert a load balancing mechanism between the HTTP engine and the servlet engine. This restricts the general pattern. For separation of the HTTP engine and servlet engine as well as providing load balancing between the HTTP engine and the servlet engine, WebSphere Advanced Edition should be used.

Possibly the most pragmatic way to make the choice is to determine if you have a requirement for all of the collaborative functionality provided by Lotus Domino, for example to provide support for mobile, disconnected users. If you do then use Lotus Domino as the application server and optionally use the WebSphere servlet engine. If you do not have this requirement then use WebSphere Standard or Advanced Edition exclusively.

You build very effective standalone e-business solutions using this pattern. In addition, should you wish to buy a ready made e-business catalog solution, then this were IBM's WebSphere Commerce suite of offerings begin (WebSphere Commerce is the follow on product to Net.Commerce). The standalone WebSphere Commerce Suite offering conforms to the this pattern.

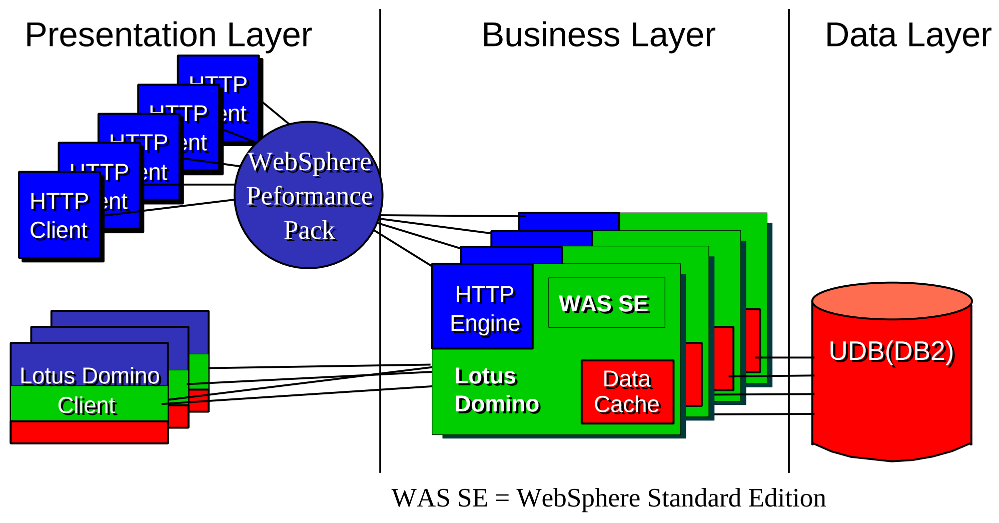
So our deployed pattern for non collaborative type applications, with product names would be.


Presentation Layer
Business Layer
Data Layer
HTTP Client
WebSphere Peformance Pack
Plugin
HTTP Server
WebSphere Standard Edition
Data Cache
UDB(DB2)
In WebSphere Standard Ed. the HTTP engine and servlet engine must be on the same machine

Standalone Single e-business, Sub-Pattern

Remember that in this case we could replace WebSphere Standard Edition with WebSphere Advanced Edition, which would allow us to physically separate the HTTP engine from the servlet execution engine. It also allows us to load balance between the HTTP engine and the servlet engine.

Where we have a requirement for collaboration we add Lotus Domino to the picture and it would be.



Standalone Single e-business, Sub-Pattern with Disconnected Users

Choosing Products for Standalone Multiple e-business

This pattern extends the standalone single e-business by adding a transactional application server for the business logic. The simple application server, which is still required, becomes an extension of the presentation layer. So we need to add a Transactional Application server to our list of products.

Required Software	**IBM Product Choice**
Web Browser	No preference - any web browser will do
Web Server	Any HTTP engine
Network load balancing	WebSphere Performance Pack
Simple Application Server	WebSphere Standard/Advanced Edition, Lotus Domino
Datastore	Universal Database (DB2)
Transactional Application Server	WebSphere Advanced Edition, WebSphere Enterprise Edition, CICS Transaction Server for OS/390, IMS for OS/390, DB2 for OS/390

The previous pattern discussed the choices for the simple application server, so now we must choose the Transactional Application Server.

The choice of application server is most commonly made from the point of view of what programming model do we wish to use to build the business logic, along with what skills do we actually have available for building business logic.

So a key decision is what programming model shall we use?

The Programming Model Choices for Building Business Logic

Virtually all IT consultants would advocate use of a component programming model for building new business function - of course while we all agree on this, we do not agree on what the model should be.

The rationale behind why components is basically to move the creation of software closer to an engineering science, than an art form, where applications can be assembled from pre-built pieces (components). These components should be of high quality and easily combined thus speeding up the development process for new applications. Of course you have to be able to provide these components in order to reuse them and that means either buy or build them.

The three major component models in the industry today are Microsoft's COM+, OMG's CORBA (which actually is not really a component model, although the next version of the CORBA specification will introduce one) and Sun's Enterprise Java Beans (EJB). The primary programming languages used for these programming choice are Visual Basic, C++ and Java respectively.

At this point I will point out that procedural programming, for example in COBOL, is another viable approach to building business logic. In fact from a performance point of view it often leads to better performing applications and procedural logic can be constructed as reusable pieces of software. This has to be offset against the reuse benefits that a component model provides, in particular the standardized approaches that make interoperation of components easier in a formal component model, as well as the tools that can be acquired to support a formal model.

This is not intended to be a description of what the programming models are, rather it is a statement of what the main programming model choices are, namely:

• Enterprise Java Beans (EJB)	(Most common language choice = Java)
• CORBA	(Most common language choice = C++)
• COM+	(Most common language choice = Visual Basic)
• Procedural programming	(Most common language choice = COBOL)

There are of course other programming styles that could be selected, however, the four listed above are the most dominant in the industry today.

The choice between them comes down to determining the mix of requirements specific to your own organization required, selecting from:

- The skills match that the programming staff has
- The reuse requirements of the application
- The deployment flexibility required
- The performance characteristics required for the application

The most important of these is actually the skills question - after all it is hard to build an application if you do not have anyone to build it!!

So we need programmers who can build to our chosen style, but how many programmers are we likely to be able to select from? The simplest way to look at this is by estimating the skill required to build to one of the styles, assuming that the distribution of skill level for all programmers conforms to a normal distribution, as with most population distributions.

We can then assign a level of difficulty to the programming model and then show them graphically as below.

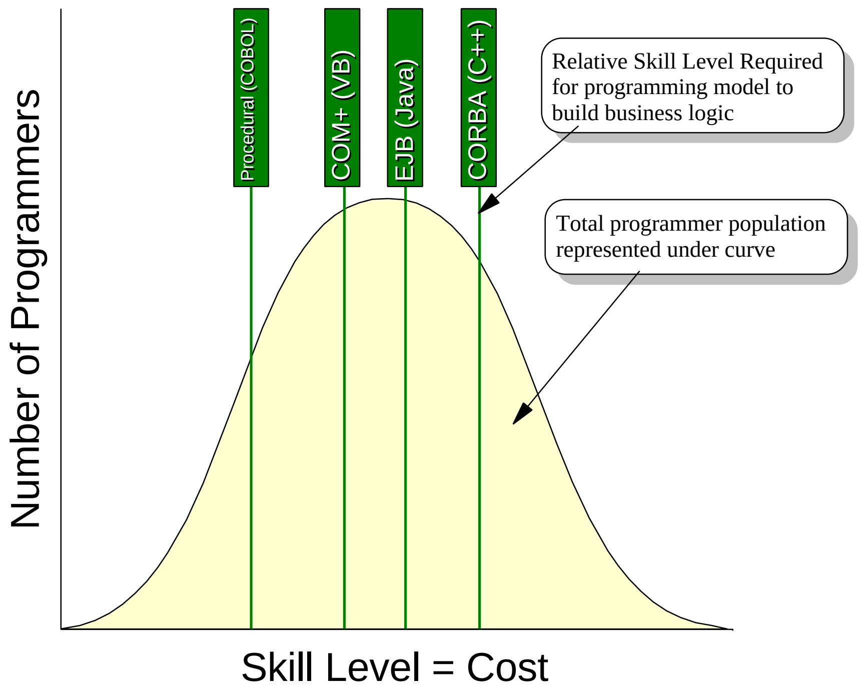


Chart of Skill Required for Building <u>Business Logic</u> Against Total Programmer Population

In the above chart the skill level required to program in the selected programming model is represented by one of the vertical lines labeled with the programming model. This chart deliberately has no numeric scale for skill level as the purpose is to highlight the relative skill level required for building business logic in each of the four programming models.

It is important to realize that this is a skills chart for building transactional business logic, rather than presentation logic (screens etc.). The same chart looking at building presentation logic using the programming languages shown in the previous chart is shown below.

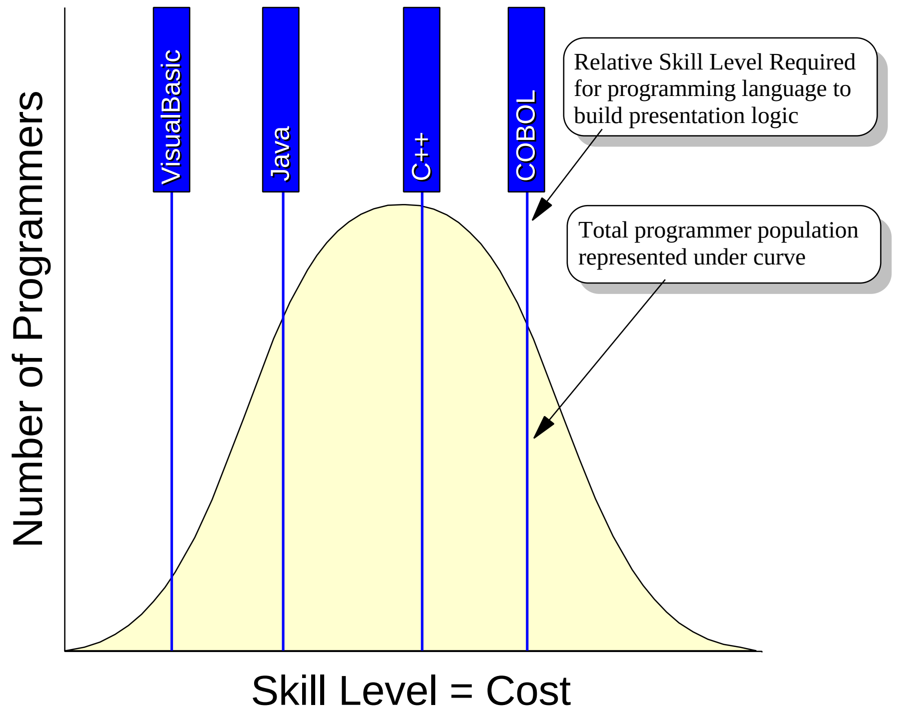


Chart of Skill Required for Building <u>Presentation Logic</u> Against Total Programmer Population

By considering these two charts together, it is perhaps little wonder that many of today's business systems combine Visual Basic clients with procedural COBOL based business logic as these are the easiest options.

It also indicates one reason why Java is the fastest growing programming language. The reason is that the same programming language can be used to build both the presentation and the business logic layers, rather than having two different languages (Visual Basic and COBOL). Of course the same could be argued for Visual Basic.

When looking at future skills availability it is necessary to look at what programming styles are being taught to tomorrow IT professionals.

IT courses in higher education teach component based programming skills with Java and C++ as featured programming environments. Tomorrow's application developers will know how to build components and not necessarily know how to build procedural style programs - so customers should have some plan to look at what components mean to them.

While skills are important, it is also necessary to consider the other requirements listed earlier.

From a reuse point of view the component models are the most reusable from the point of view of reusing other peoples components. This is primarily due to the well defined interfaces that can be achieved using the component models, however, reuse has to be thought about up front of any development, as use of a component model does not automatically imply reusable software!

In terms of the most openly reusable programming environment, this would be either CORBA or EJB as these two have the most open interfaces. In terms of the most commonly used component programming model, this would have to be the COM+ Microsoft model, due to the almost monopoly in desktop/PC Windows based applications. This is not so for server based applications.

In fact this leads to the next consideration, which is platform coverage. While COM+ is dominant in Windows based systems, it is only (with the exception of some very esoteric offerings) available for windows. Therefore, if you wished to build business logic for multiple server platform deployment it is a poor choice. The most widest choice will be provided by Enterprise Java Beans (EJB). Note that I said "will be", EJB is still relatively new, but is being quickly adopted by all middleware application server vendors as a standard programming model for application servers. Roll out of servers supporting EJB should gain pace in 2000 and on into 2001. In fact the only vendor resisting this is Microsoft, who presumably regard this as a major threat to their Windows monopoly.

The next widest choice is CORBA, although as was previously stated the current CORBA specification does not define a component model, rather provides common services from which you can construct your own. It appears that there is a convergence of EJB and CORBA, with a very likely prospect of EJB being the component model on top of CORBA services.

The most widely available procedural programming model for transactional applications is IBM's CICS, providing support for COBOL and other languages to build business logic across NT, UNIX, OS/400 and OS/390 systems.

From a performance point of view procedural COBOL is the fastest, and for high volume transaction processing the procedural approach to creating the business logic should be considered. Of the component models, CORBA tends to produce the more highly performing code, although EJB and Visual Basic are not far behind.

So let me summarize the previous discussion on building business logic in the following table:

Programming Model	**Ease of Programming**	**Reuse**	**Platform Coverage**	**Performance Volume**
EJB	Medium	Good	Widest	Medium
CORBA	Hard	Good	Good	Medium
COM+	Medium	Good	Poor	Medium/Low
Procedural	Easy	Hard	Good	High

So the selection on programming model depends on these factors. The performance factor is most likely a point in time statement and the performance of all of the component models will get better. It is unlikely that they will reach the performance of complied COBOL, due to the differences in design of a component versus procedural application, however, it is very likely to be eventually good enough for 95% of all applications. Right now EJB and CORBA are probably good enough, performance wise, for 60% of all applications.

Given this assessment, it is my conclusion that the Enterprise Java Beans programming model is the likeliest successor to the Visual Basic/COBOL combination mentioned earlier. This does not mean that existing applications will be rewritten, rather that new ones may well be focused on the EJB programming model.

Of course the skills that you have today are a key determining factor in deciding on a programming model. For example, if you have 200 COBOL Programmers it is unlikely they will be able to build an application using CORBA components without a massive retraining exercise - which may well be cost prohibitive. Similarly, if you have a Java programming skill base, trying to persuade those programmers to build CICS COBOL applications will be difficult, to say the least.

So what IBM products fit the categories that we have just discussed?

IBM’s Transactional Application Servers

This section lists the application servers that IBM offers and provides a few pointers to when you might want to use them. This section does not provide in depth descriptions of how these application server work, you should consult the product literature for this type of detail.

WebSphere Advanced Edition

IBM’s standalone EJB server supporting the EJB programming model, as well as Java Servlets. It is probably the easiest way to get started with Enterprise Java Beans. The server is available for UNIX and NT platforms. Performance should be adequate most applications, but not recommended for very high volume transaction processing.

WebSphere Enterprise Edition (TXSeries)

IBM’s UNIX and NT transaction processing monitor applications servers, providing a choice of CICS or Encina programming styles. Procedural and component based applications can be built, although the component models are proprietary to the CICS and Encina environments. Enterprise Java Bean support is planned, but not yet available.

This selection is suitable for those customers looking for robust deployment for mission critical transaction processing type e-business applications.

The choice between CICS and Encina programming styles is really a choice in skill type. CICS fits best in procedural OS/390 heritage environments, whereas Encina fits more naturally into a pure UNIX environment. If the OS/390 platform is required then CICS should be the choice as Encina is not available for OS/390.

WebSphere Enterprise Edition (Component Broker)

IBM's CORBA application server offering for UNIX and NT, providing both CORBA and Enterprise Java Bean support. This application server provides comprehensive CORBA services to allow the construction of simple to highly sophisticated component based applications. The Component Broker framework, built into the product, simplifies the complexity of building CORBA based applications, making Component Broker the simplest choice for building CORBA applications.

Component Broker is also available for OS/390, and on this platform it can also be considered as a way to provide a component bridge to other application servers, particularly IMS, to provide reusable interfaces on top of existing applications. This capability is also provided in support of CICS and DB2 applications. Component Broker provides this capability through use of supplied Component Broker Adapters that are integrated into the Component Broker framework. These adapters can be thought of as a way to build the transactional adapters discussed in the previous section on Business Integration.

Note that while this could also be accomplished using Component Broker on non OS/390 platforms, deployment there will run into the Performance / Distribution / Synchronization triangle limitations, previously discussed. If these adapters are to be transactional and we will be distributed away from the OS/390 platform our performance will suffer. Component Broker also provides a transactional adapter to MQSeries, which will be the store and forward mechanism highlighted in the next few sections.

CICS Transaction Server for OS/390

IBM's most general purpose, robust and scalable transactional application server. CICS provides the CICS application programming interface, or API, that is common across all CICS application servers. Programming languages that are supported, include, procedural languages COBOL, PL/1 or C as well as the component languages of C++ and Java - this provides choice in the style available for building CICS applications. Java can be used in both client and server development of CICS applications. Enterprise Java Beans support is currently under development, for release some time in 2000.

Due to its dominance in OS/390 business systems - the CICS programming model is usually associated with mainframe development - although it is quite possible to use without ever having a mainframe anywhere in an enterprise. In conjunction with IMS, CICS provides some 80% of the world business transaction processing capability and is the most pervasive business application sever.

CICS Transaction Server for OS/390 should be considered by those customer wishing to deploy highly scalable, in terms of number of users and performance, applications.

IMS for OS/390

IBM’s other OS/390 based transactional application server, focused similarly as CICS Transaction Server for OS/390, but with more applicability for hierarchical database applications.
Programming languages that are supported, include, procedural languages COBOL, PL/1 or C as well as the component language of C++. At present no Java support is available for server side development, although IMS does have an Java client

While customer can and do write new IMS business function, it is usually encountered where customers wish to reuse an existing IMS application.

DB2 for OS/390

Yes DB2 for OS/390, which at first glance many people would not associate with an application server. However, the DB2 stored procedure engine on OS/390 is in reality an application server that supports the DB2 programming model, as well as any other OS/390 interfaces - for example CICS EXCI (which is also transactional). Multiple stored procedure engines can be executed providing for linear scalability, and for data intensive applications the performance of the DB2 stored procedure engine can be greater than that or CICS or IMS, when these application servers are used with DB2 as the only datastore.

The stored procedure engine is focused on supporting applications that manipulate DB2 data - whereas CICS and IMS are more general purpose application servers.

The choice between a traditional application server like CICS or IMS and the DB2 stored procedure engine approach can be looked at as follows:

If you are looking to provide business function that is portable across multiple platforms, and access multiple data sources then an application server, like CICS, is the best choice.

If you want to build an application that is definitely going to run on OS/390 and primarily uses DB2 then the stored procedure engine is something that should be considered, particularly if you have real end user clients that want to manipulate the data directly, as the standard database development tools can be used on the client.

A Quick Word About COM+

You will have noticed that nowhere in the previous application server descriptions did I once say that IBM’s transactional application servers support COM+. There is a good reason for this, and that is because they do not.

COM+ is supported as a client that can request service from an IBM Transactional Application server but it is not supported as a runtime for business logic. COM+ is proprietary to Microsoft, which at present is focused solely on Windows based applications. This severely limits its capability to support large scale e-business applications.

Summary

The choice of transactional application server really depends on the style of programming that matches your programming skill set, be that today's skill set, or the skill set that you would like to have - bear in mind that the same application can be built using any of the choices, as functionally these server environments provide the same things for the sub-pattern.

The two key areas of difference today are:

- Battle tested robustness, which is where the CICS Transaction Server for OS/390 and WebSphere Enterprise Edition(TXSeries) servers have the advantage. IMS and DB2 for OS/390 can also be considered where appropriate.
- Ease of Component based reuse, where the Enterprise Java Bean programming model appears to be in the lead, supported by WebSphere Advanced Edition for the quickest or simplest start, and WebSphere Enterprise Edition (Component Broker) for even more capability.

Choosing Products for Integrated Single e-business

This pattern extends the standalone single e-business by adding a store and forward mechanism along with an information management hub. In this case our business logic is really the control logic in the information management hub, which may range from simple routing of requests to the control of workflow within an enterprise. This control logic then accesses business logic we already have on other application servers.

Required Software	**IBM Product Choice**
Web Browser	No preference - any web browser will do
Web Server	Any HTTP engine
Network load balancing	WebSphere Performance Pack
Simple Application Server	WebSphere Standard/Advanced Edition, Lotus Domino
Datastore	Universal Database (DB2)
Store and Forward mechanism	MQSeries
Information Management Hub	Combination of MQSeries Integrator, MQSeries Workflow and Lotus Domino

The new products in this case are MQSeries, as the store and forward mechanism. Given that MQSeries is the most pervasive and reliable store and forward mechanism in the industry, this should be no great surprise!

The Information management hub can potentially play many roles. The simplest is as a router and message transformer, in which case the MQSeries Integrator product provides the required capability. For more complex business process flow, for example workflow, MQSeries

Workflow adds the capability to control information flow between business processes on application servers and people, either at connected or disconnected workstations.

In the case of integrating the workflow to people, Lotus Domino adds significant capability and as such can be logically combined in the Integration hub, with requests being passed backwards and forward via the store and forward mechanism, MQSeries. It can be argued that this is a case of Domino acting as another application server for collaborative workflow. However, I believe that the consideration of what people interaction is being provided by which product, MQSeries Workflow or Domino, is required to avoid duplication of effort.

Choosing Products for Integrated Multiple e-business

This pattern adds the Transactional Application server into the integrated single e-business pattern to add a capability to create new business logic as well as reuse existing business function. The choice in Transactional Application Server is the same as was discussed in the section on Standalone Multiple e-business sub-pattern. The Information Management Hub is the same as for the Integrated Single e-business sub-pattern

Required Software	**IBM Product Choice**
Web Browser	No preference - any web browser will do
Web Server	Any HTTP engine
Network load balancing	WebSphere Performance Pack
Simple Application Server	WebSphere Standard/Advanced Edition, Lotus Domino
Datastore	Universal Database (DB2)
Transactional Application Server	WebSphere Advanced Edition, WebSphere Enterprise Edition, CICS Transaction Server for OS/390, IMS for OS/390, DB2 for OS/390
Store and Forward mechanism	MQSeries
Information Management Hub	Combination of MQSeries Integrator, MQSeries Workflow and Lotus Domino

Choosing Products for Integrated Multiple e-business to e-business

This sub-pattern is really the same as the Integrated Multiple e-business sub-pattern, except that we have to connect two Information Management Hubs together. The main challenge there is one of security, namely what sort of encryption is required for the data as it leaves our enterprise and do we have to navigate through firewalls if our network is the Internet. Both of these consideration can be addressed using the same products as before from the middleware perspective and so the product list is:

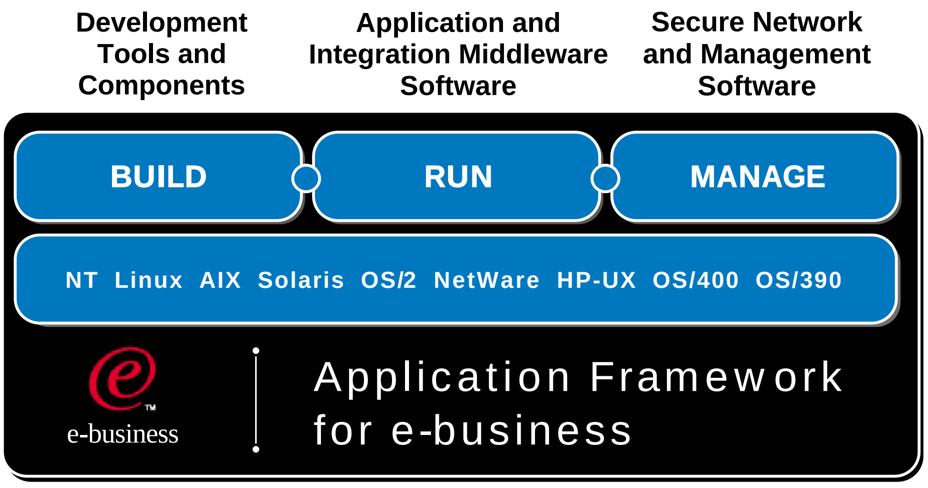
Required Software	**IBM Product Choice**
Web Browser	No preference - any web browser will do
Web Server	Any HTTP engine
Network load balancing	WebSphere Performance Pack
Simple Application Server	WebSphere Standard/Advanced Edition, Lotus Domino
Datastore	Universal Database (DB2)
Transactional Application Server	WebSphere Advanced Edition, WebSphere Enterprise Edition, CICS Transaction Server for OS/390, IMS for OS/390, DB2 for OS/390
Store and Forward mechanism	MQSeries
Information Management Hub	Combination of MQSeries Integrator, MQSeries Workflow and Lotus Domino

Conclusion

This paper has addressed two basic things with regards to e-business:

- The consideration of basic architecture design principals, regardless of product, with the goal of providing some concrete guidance on the types of architecture patterns that can be built.
- The consideration of what IBM products could be used to construct these patterns.

What this paper has not looked at are the tools that could be used to build the applications, the management software to control and monitor the deployed patterns and the security products to secure the networked applications. These software challenges can also be addressed with IBM software and are described within IBM's Application Framework for e-business.


Development Tools and Components
Application and Integration Middleware Software
Secure Network and Management Software
BUILD
RUN
MANAGE
NT Linux AIX Solaris OS/2 NetWare HP-UX OS/400 OS/390
e-business
Application Framework for e-business

More information on IBM software and the Application Framework for e-business can be found on the World Wide Web in IBM's Software Pages.

IBM Software can be found at:

http://www.software.ibm.com

and e-business specific information can be found at:

http://www.ibm.com/e-business/software/

Appendix I - Three Tier vs. Two Tier Client Server

This appendix covers the debate over three tier versus two client server architectures. It should be noted that for web based e-business applications, the whole industry agrees on a three tier foundation. This section is included to provide the background as two way that is so, as well as pointing out that not everyone builds applications for the Web, and two systems can be useful.

Three Tier Client/Server Application Architecture

Over the last few years there has been an increasing consensus in the IT industry that the best way to build large scale business critical applications is to do so utilizing a three tier Client/Server application architecture.

This is where the application functions of Presentation, Business Logic, and Data Access are separated as shown in *Figure 1*.

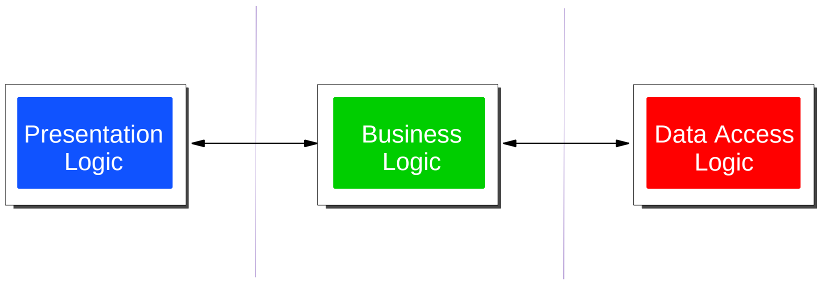


Figure 1: Three Tier Client/Server Architecture

While the three functions may all reside on the same physical hardware, there is a distinct piece of application code associated with each function and a distinct piece of middleware software looking after the application code's best interest.

This is distinct from the two tier Client/Server architecture which has the presentation and business logic on a single tier with the data access logic on separate tier, as shown in *Figure 2*.

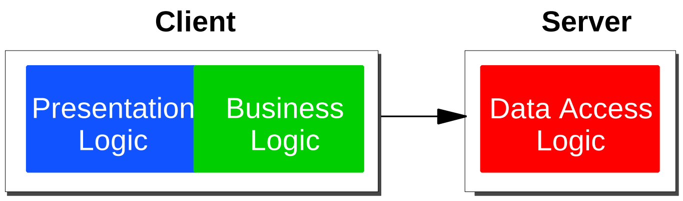


Figure 2: Two Tier Client/Server Architecture

Pace of Change in the Tiers

Much of this paper will look at why three tier is a better bet for scaling applications where many users and databases or data stores are required to support the business. However, there is another aspect that should be looked at, and that is building business critical applications that can respond to changes in technology. Business critical application stay deployed in an organization for many years, for example 25 years is not uncommon for applications based on CICS or IMS, and these applications must be supported and respond to change over what in technology terms is a very long period of time.

The three tiers, presentation, business logic, and data access all are subject to change, be it hardware or software. However, the rates of change have historically been quite different. For example, presentation devices have undergone rapid changes in the last few years, going from character based displays, to graphical user interfaces (GUIs), to web browser HTML and now Java applets - and technology like smart card, kiosks and television sets are entering the scene. Application servers tend to be a little more stable in the change, mainly driven by faster hardware and operating system improvements, or by new application development to support new or changed business processes. The data stores tend to be changed the least frequently - after all how often does an organization swap out one database for another?

The separation of the three tiers allows organizations to respond to these differing rates much more effectively than a more monolithic approach such as that in a two tier architecture. For example application developers that built applications using a tool like Powerbuilder to access a database are finding that adapting them to be accessible from a web browser can require major rework. Of course having middleware that allows you to implement a three tier architecture is no guarantee that the easy separation will occur. For example character based (3270) CICS applications that were built in the eighties usually had the 3270 screen handling logic embedded in the business logic - reuse of the business logic with new presentation devices now requires either rework of the application code or a screen scraping adapter to drive the existing logic. This may also occur when tightly integrated tools, such as Microsoft's application development environment with NT server, are used to build applications. The tools may provide an easy way to build applications for today's technology, but may well be the monolithic applications (that are hard to reuse!) of tomorrow.

The Scalability Drawbacks of Two Tier

The vast majority of Client/Server applications have been created using the two tier architecture and these applications, for the most part, have been very successful. However this architecture has been shown to fail when the number of users becomes great and update of data is required.

This can be illustrated using two examples, a simple two tier Client/Server example and a complex two tier Client/Server example

Simple Two Tier Client/Server Example

The simplest case of a two tier system is where a database client communicates with a database, the client is usually driven by a rapid application development tool, such as Powerbuilder, Visual Basic. Nearly all of the user application is provided using the tool as the development vehicle and the focus of the application is to make the individual end-user's job as easy as possible - good news for the end-user! In my simple example let's use a DB2 client and server combination, shown in *Figure 3*.

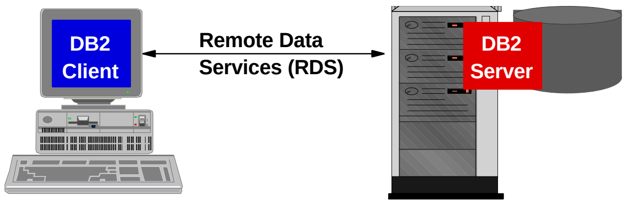


Figure 3: Simple Two Tier Client/Server Example with DB2

As the number of users (clients) is scaled up problem start to manifest in a number of areas. The major ones being:

- **Network capacity:** the more users that get added the more data flows and the more saturated the network becomes.
- **Connection Management**: All the clients are connected to the database which has finite limits in how many can be connected at one time.
- **Client Version Control:** The business logic is on the client, as more users are added the management task of maintaining the correct levels of application software increase.
- **Locking Contention:** The easy to use application development tools are focused on the individual end-user, and as a result they can take rather liberal locking strategies when allowing update of the database. As a consequence locking contention can grow as the number of users grows.

In general this simple model can support up to 1000 users doing both update and query intensive work, although the problems indicated above can appear at much lower user populations. In fact Gartner Groups chart on two tier versus three tier system scaling sets the average number of users at 250, although that also includes more complex two tier Client/Server solution, an example of which follows.

Complex Two Tier Client/Server Example

Early Client/Server solutions tended to be focused on departmental level solutions, where all that was required was access to a single database. However, as these systems developed, the requirement to access more and more data sources became apparent. Once more, using DB2 as the example, here is an example of such a system, *Figure 4*.

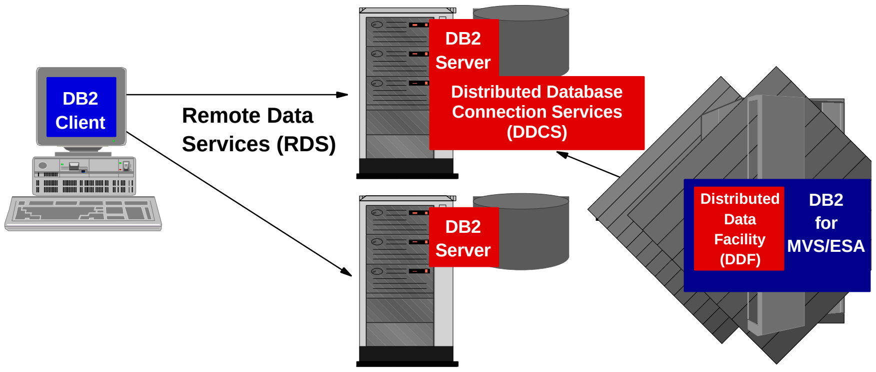

DB2 Client
Remote Data Services (RDS)
DB2 Server
Distributed Database Connection Services (DDCS)
DB2 Server
Distributed Data Facility (DDF)
DB2 for MVS/ESA

Figure 4: Complex Two Tier Client/Server Example with DB2

In this example our client now has direct access to two databases and indirect to another. This type of system suffers from the same scaling problems as in the simple case, however, some more issues arise with this increased complexity. The additional major issues are:

- **Database Availability:** In order for this system to function all of the databases have to be available, which can lead to a management nightmare as maintenance cycles have to be synchronized over potentially disparate departments or even companies.
- **Security:** The security is based on the user access the database, what tables can be updated, what rows can be changed. This is sometimes referred to as data oriented security. There is no control over how the values contained in the database may be changed, and in the case of multiple databases, the user may not belong to the organization owning one of the back end databases. In addition, in order to meet all requirements for update the security restrictions on what can be updated in particular databases can become very complex to manage over time as the interdependencies grow.
- **Database Organization:** Related to the security interdependencies is the actual organization of the databases themselves. The application logic is expecting specific database structures, any changes in these structures may well require significant re-coding of the applications**.**

The focus is again to make life easy for the end user, and in order to do this the system focus is on making the databases look like one big database. Very good for data mining applications, query only, not so good for update intensive applications.

The Benefits of Three Tier

The heading of this chapter was three tier application architectures so here is a brief look at how the previous two tier examples would benefit from a three tier approach.

Simple Three Tier Client/Server Example

In the simplest case three tier Client/Server is all about inserting an application server between the client and a single database, see *Figure 5.*

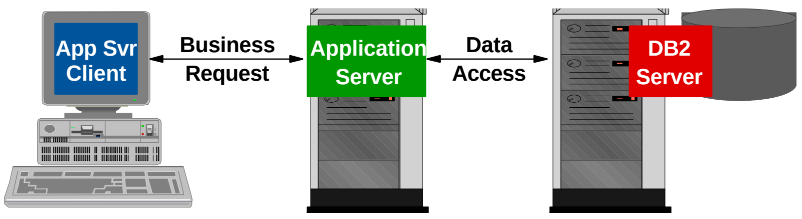


Figure 5: Simple Three Tier Client/Server Example with DB2

The client now makes business requests of the application server, and the server accesses the database while processing(executing) these requests.

Therefore, you can view the application server as a centralized execution environment for application business logic. So the question is what benefits does this have over the simple two tier architecture?

In this simple case the application server provides the following benefits:

- **Reduced network traffic:** Business requests are for the most part much smaller in size than a result set of a database query. Therefore, with the business logic no longer on the client the traffic from client to server, in this case application server is much reduced. There still has to be traffic between the application server and the database, but this can be accomplished at a central point where networking speeds can range from fiber optic to channel attachment. Certainly much faster than wide area networking or modem speeds.

- **Better Connection Management:** The number of connections into the database is much reduced as we only need a number of connections equal to the maximum number of concurrent requests that the application server is going to make while executing business requests. However, there will still be the same number of potential clients that need to connect to the application server. We now have the opportunity to replicate the application server and provide additional connection capability, while only increasing the connection to the database by the number of additional concurrent requests from this new server, see *Figure 6*. By

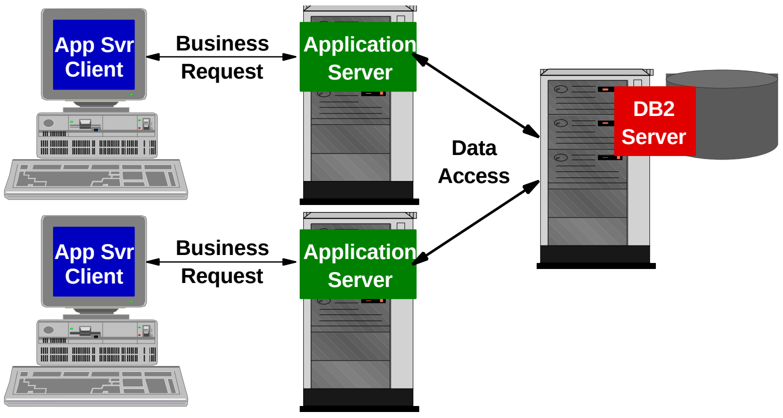


Figure 6: Replicating Application Servers

continuing to add application servers we can scale the number of end-users up to how ever many we need (or can afford!) The limiting factor in the system is how many concurrent requests can the database handle, rather than how many users can connect to the database. As an additional bonus, we also improve the system availability of the system to the end-users. Should we need to bring down an application server for planned (or unplanned!) maintenance, we can do so while allowing the remaining servers to pick up the work load, as long as we have some way to redirect client traffic.

- **Easier Version Control:** With the application logic now mainly on a more centralized server, we have significantly reduced the number of parts requiring update if and when we change the application. Of course if we change the user interface the client will need to be updated, but these changes tend to be less frequent than the business logic. (Note that if the client is a web browser, we can control the user interface from a more central point as well!)

- **Better Focus on Locking Strategy:** There is an opportunity to focus on the application logic with a view to creating applications that focus on allowing many concurrent requesters to access a data source for update, this allows us to alleviate to locking contention issue of two tier client server. However, note that it is necessary to develop the applications with this in mind.

These benefits do not come for free. In this system it is necessary to develop both the client user interface as well as the application logic that resides on the server. We can still use the same client development tools for building graphical user interface minus the database access logic. the choice of tools for the server depends on what the application server is to be, more on this later.

So there is benefit in this simple case, and these benefits are most apparent in systems that require greater than 1000 users with requirements for update of data.

Complex Three Tier Client/Server Example

In the more complex case, where we have more than one data source, the requirement for an three tier architecture becomes more compelling. In these systems the data sources are all access via application servers which communicate between one another to satisfy the business request made by the client.

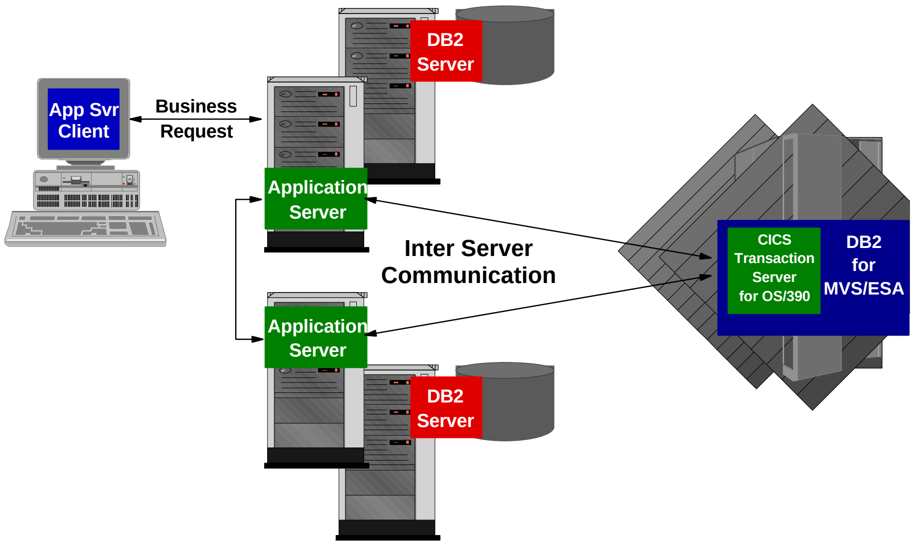


Figure 7: Complex Three Tier Client/Server Example with DB2

Figure 7 shows an example of such a complex three tier system. The application servers intercommunicate and control access their respective databases. This architecture allows us to address the problems encountered in the complex two tier case.

- **Option for Deferred Work**: The databases are now accessed via business logic residing on the application server. There is now the option of allowing the database to be not available, but still to allow the application server to process a request by storing (queuing) it for later processing. This allows us the option of isolating the database from the rest of the system for the purposes of maintenance, planned and unplanned. Of course the business requirements of the system have to allow for the database to be updated later (or asynchronously) for this to be allowed.

- **Business Function Based Security:** The use of business logic to control access to the database also allows us to provide logic to determine the limits in which individual users are allow to change the data. The end user is typically identified by a userid pertaining to the application server, this userid can be used to allow different update capabilities. For example, if the application is used to allow a bank teller to withdraw money from an account, a userid for a trainee teller may only allow a maximum of a few hundred dollars to be withdrawn, whereas the userid for the bank manager may have no limit. In addition to the increased granularity of data access, the security of each database no longer has to be interrelated, making management much easier. The database is only accessed by the application server using a database userid associated with the application server, rather than that of the end-user.

- **Isolation of Database Structure:** The use of an application server’s business logic to access a database masks the structure of that database from the end-user. Therefore there is no longer a requirement for the structures of the different databases to directly relate to one another. This frees the database administrator to determine the best structure for their database in isolation of the other, this easing the systems management workload.

Some Final Words on Two Tier vs. Three Tier Architectures

At the beginning of this chapter I stated that there was broad agreement in the industry that three tier architectures were the way to go for building scaleable business critical systems. There is the famous chart of cost versus complexity, shown as number of users, first published by Gartner Group, that illustrates this point, the chart is shown in *Figure 8*.

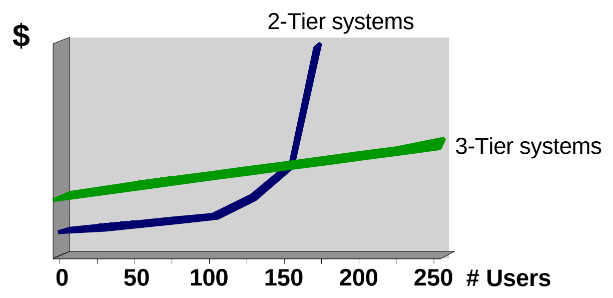


Figure 8: The cost of two and three tier architectures

Cost of Ownership Example

The Gartner group chart indicates how the costs start to rise as the number of users are added to the system, however this is not limited to the cost of building these systems. Upgrading can also show some interesting cost features as applications grow. Here is a real life example of what can happen in a two tier system as new functionality is added.

New function in two tier systems mean change to the client - that is where the business logic is. This often results in growth in the client, in terms of resource usage; disk space, memory etc. as well as functionality. The cost of this was brought home to one customer who had a branch based application, covering 32,000 PCs. The original application was implemented using a two tier architecture on DOS PC’s. As the application grew, the 640KB limit of DOS became more and more of a problem to the point that it prevented new required function. After some research the customer decided that if they converted their application to run under OS/2 and add some memory to the PC’s they could overcome this problem. Technically this all made sense. However, they then calculated how much it would cost to upgrade the PC’s. The additional hardware and software and people cost to do he upgrade was estimated at two thousand dollars per machine, about five hundred of which was the hardware. Not a great deal of money for one PC, but when you multiply by 32,000 this gives a total cost of $64,000,000 - needless to say, the customer decided that this probably was not such a good idea! The customer actually moved to a three tier implementation with about two server per branch. New business function was on the server, and any growth in the application meant that the servers were the primary machine needing money spent on them, a much smaller number, a much better business proposition.

Complexity and Quantity of Users

Understanding how the costs vary with scaling two and three tier architecture is very important. However, it may be more useful to look at the situation in terms of complexity versus number of users and determine where the two tier, simple three tier, and complex three tier systems might fit. The chart shown in *Figure 9* shows the relationships.

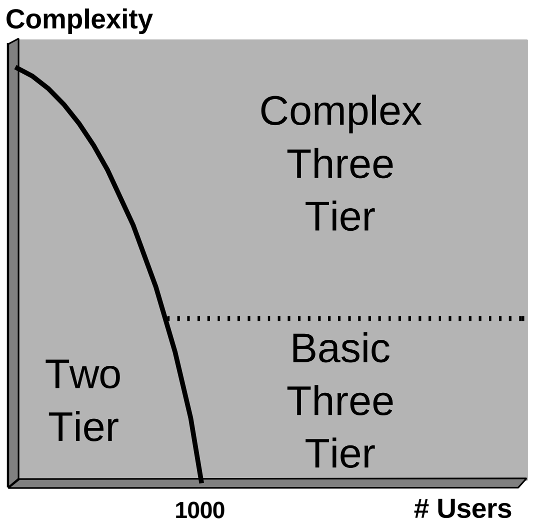


Figure 9: Realtionship of complexity to number of users

The key points that I think can be drawn from this chart are:

- Two tier systems can be scaled up to about 1000 users, however, as the complexity of the system increases, the number of users it can support declines.

- There is no real distinct line between simple and complex three tier systems. A three tier system can support as many users as you need it to do by replicating the application server, therefore basic three tier has no limit in terms of number of users. However, as the complexity rises there is a point in most business systems where multiple databases become required to support the business. In this case the multiple application servers provide a great deal of benefit and characteristics such as load balancing, distributed transaction management, high availability and systems management become firm requirements rather than nice to have features and a complex three tier solution becomes required.

From the previous text you may be under the impression that two tier Client/Server is bad, this is not so. Rather, it is necessary to consider what the requirements of an application are and then pick the appropriate implementation architecture. This will usually require that the people that write the applications and the people that will run the applications work together to determine the best fit - after all it is not much use writing an application in 10 minutes if you can not run it in a production environment, and it is also little use if the production environment restriction prevent the application from being written in the first place! A balance is required.

Appendix II - The Concept of a Transaction

In application systems design the concept of a transaction refers to a series of related data update operations that are applied to one or many data stores. This series of operations is managed with what are referred to as ACID properties. ACID is an acronym made up from the following four properties that transactions should have.

- **A**tomicity:
  All of the updates are contained within a single Logical Unit of Work (LUW) so that all of the updates either succeed or all fail
- **C**onsistency:
  At the end of the transaction the system is left in stable state
- **I**solation:
  Other applications can not corrupt the update operations that the transaction performing.
- **D**urability:
  The effect of the transaction are permanent

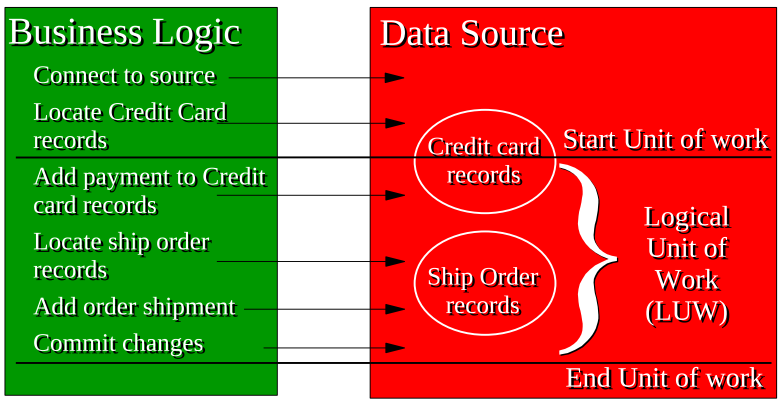
An example of a transaction might be where an application records a web user’s credit card information in a payments database and records order information in a shipment database. Clearly the web user would like to know that any charge to the credit card will result in shipment of ordered goods - so these two updates should really be done under transactional control.

Let’s take this example and look at what one system needs to do in the following cases:

1. The two updates are initiated from one application server and are applied to one database.
2. The two updates are initiated from one application server, but are applied to two databases.
3. The two updates are initiated from separate application servers, each updating a database.

One Application Server and One Data Source

The flow of activity for one application server and one data source is shown in the following diagram.



A transaction - two updates to one data source

The business logic makes two requests of the data source for update and then requests that the data source commit the update.

During the updates it is necessary for something to keep track of what we have done inside our transaction, this something can be referred to as a transaction coordinator. The transaction coordinator makes sure that the update operations either all occur, are committed or do not occur, are rolled back.

In this case all of the updates are applied to the same data source, so we can rely on the data source to provide this transaction coordination. The data source starts a unit of work when we request an update, keeps track of all of the update that follow, and then commits the update when asked to by the application. This commitment can be referred to as a syncpoint. If some problem is encountered during the commitment, the data source returns the data to the state it was before our transaction started - rolling back our updates.

The data source is also responsible for making sure that other applications can not change the information we are changing while our unit of work is in progress. There are two approaches to how this is achieved. The first is not to allow any access to the new data until commitment has finished. This means that the data is locked from any other application, including for read only operations. The second is to allow read only operations on the data, but to only show the data as it existed before we started the unit of work. This is sometimes referred to as a dirty read, as the data may change once our transaction is completed.

All of this activity is contained within one data source and can be thought of as a local transaction relative to the data source.

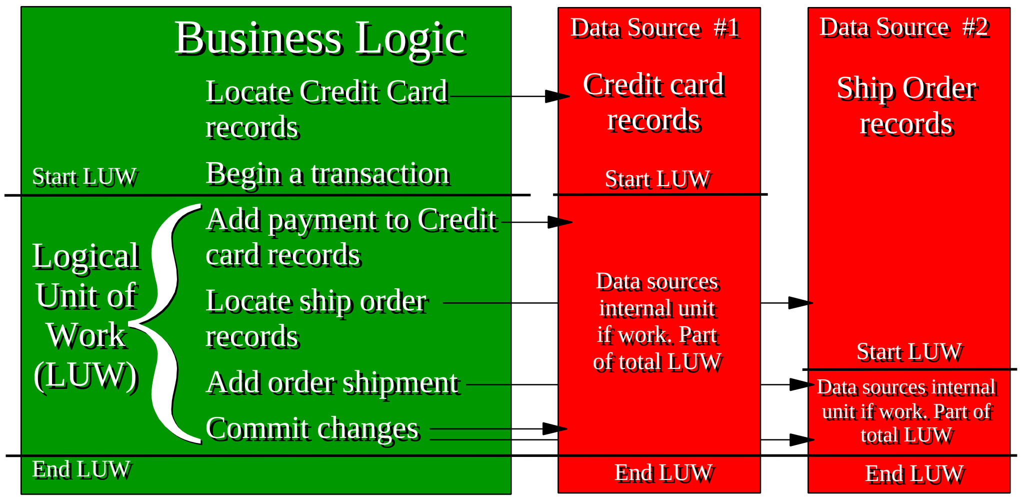
One Application Server and Two Data Sources

In the case where we have two data sources the process is a little more complex. According to the architecture the two data sources should be independent. Hence, any updates we do to one, are not known by the other data source, unless our business logic, or application server, can tell the other data source what is going on. Therefore we can no long rely on the data source to provide the transaction coordination function, the application server must do this.

However, our data sources must still provide the ACID transaction properties for their own sources of data. Therefore a logical unit of work must be tracked by the data source for its own updates, and this is sometimes referred to as transactional resource management, the data source being the resource manager.

The total logical unit of work is now tracked by the application server, which fulfills the role of transaction coordinator.

A simple flow for this process is shown in the following diagram:


Business Logic
Locate Credit Card records
Start LUW
Begin a transaction
Logical Unit of Work (LUW)
Add payment to Credit card records
Locate ship order records
Add order shipment
Commit changes
End LUW
Data Source #1
Credit card records
Start LUW
Data sources internal unit if work. Part of total LUW
End LUW
Data Source #2
Ship Order records
Start LUW
Data sources internal unit if work. Part of total LUW
End LUW

Single application server with one update to each of two data sources

If one of the data sources discovers a problem that prevents it from committing its updates associated with the LUW being coordinated by the application server, it should roll back the updates it has made and notify the coordinator that it has done so. The coordinator then instructs the other data source to roll back its updates so as to retain the ACID transaction properties for the whole operation.

This is known as a local transaction relative to the application server, but distributed relative to the data sources.

Note that in this case there are actually two requests that flow from the application server to commit the data, one to each data source. These requests can not be made simultaneously, so if we tell the first data source to commit, which it does successfully, and while the first data source is doing this we tell the second one to commit, but it fails and rolls back its updates, we end up with an inconsistent result, which violates the ACID property requirement. This is known as an in doubt situation and the time in which it can occur is known as the window of doubt.

This window can be minimized by utilizing a two phase commitment process for the final commitment of the transaction. This is sometimes referred to as a synclevel two approach.

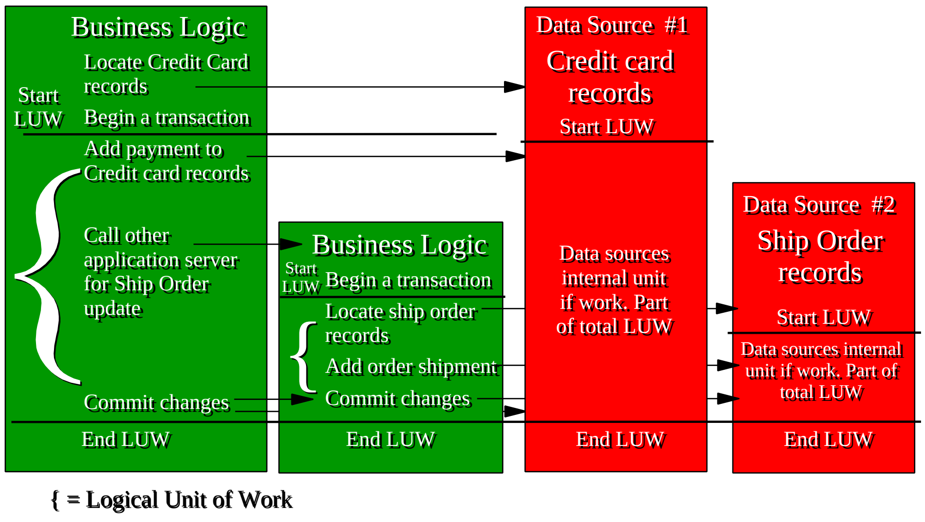
In two phase commit the transaction coordinator notifies each of the data sources (resource managers) that the it would like to commit the transaction, and to start the process off it would like the resource managers to prepare to commit any updates, by doing everything short of the final write to disk. Common preparations include flushing of buffers and preparing the physical disks for the write. Once the resource managers have prepared they send back a notification that they are ready, this is the end of the first phase. If one of the resource managers has a problem, it should roll back its updates and notify the transaction coordinator that it has failed to prepare. The transaction coordinator will then notify the other resource manager to roll back its updates.

Usually the two resource managers successfully prepare and notify the transaction coordinator that they are ready. This ends phase one and begin the second phase, where the transaction coordinator sends out a request to each resource manager to commit the data. As each resource manager should be ready, this operation takes minimal time. However, it is still possible for there to be a failure in this final stage, where one commits and the other does not, thus while the window of doubt is reduced it is still present.

This two phase commit process is a system level conversation between the application server and the resource managers. The industry standard communication interface for this conversation is known as XA, developed by the X/Open group. All of IBM's UNIX and NT transactional servers use this interface for transaction coordination. IBM's resource managers, such as DB2 and MQSeries as XA compliant and can be coordinated by an XA compliant transaction coordinator. In the case of Enterprise Java Beans the interface is known as JTA, which is a Java equivalent of XA.

Two Application Servers and Two Data Sources

This is very similar to the previous case, except now we introduce an additional resource coordinator.



Two application servers with one update to each of two data sources

The first application server acts as the Transaction coordinator and its communication with the data source it will request updates from is as in the previous case. However, in order to make the updates to the second data source, the first application server has to invoke some business process on the second application server. Further more it needs to tell the second application server that it should provide any updates under transactional control which the first application server will coordinate. In order to pass the transactional information between the application servers it is necessary to use a synchronous invocation method using a protocol that supports distributed transactions - resource coordinator talking to resource coordinator. This is different from the resource coordinator communicating with the resource manager, and different protocols are used. In the case of EJB’s this capability is provided by the Java Transaction Service (JTS), in the case of CORBA it is the Object Transaction Service (OTS) and for IBM’s OS/390 based transactional application servers (CICS and IMS) it is LU6.2. Ideally the invocation method for invoking the second piece of business logic should hide the use of these transactional protocols, making the call seem like a simple request with reply.

The second application server now requests update of the data source it communicates with and the communication is as in the previous cases.

When the time comes to commit the updates, the first application server initiates a two phase commit process that involves the data source it knows about and the second application server. Note that this first application server has no knowledge of what data has been updated by the second application server. It relies on this second application server to manage the transaction commitment for the data sources this second application server has updated. This means that the second application server will initiate a two phase commit process with its data source, passing back to the first application server the success or failure of the prepare and then the commit.

This can be thought of as chain of application servers, and it is possible to add more to the chain, where subsequent application servers invoke other application server. Of course care needs to be taken if this is done, as the longer the chain, the longer the commit process will take and hence the worse the performance will be. In addition the in doubt window will grow in size.