On demand business solutions
White paper

WebSphere® software

Coupled or decoupled, and heavyweight and lightweight delivery considerations in an enterprise service bus.

By Andy Stanford-Clark, master inventor and senior technical staff member, IBM Hursley Park Laboratory

January 2005

Contents

2 Introduction
2 What is an ESB?
5 ESBs and the physical dimension
7 An ESB based on IBM products
7 The service in an enterprise service bus
9 Tightly or loosely coupled considerations in an ESB environment
11 Loosely coupled or asynchronous considerations in an ESB environment
13 Lightweight and heavyweight messaging considerations in an ESB
14 Lighter-weight messaging
16 Lightweight messaging
17 Conclusion
19 For more information

Introduction

This white paper explores coupled and decoupled, as well as lightweight and heavyweight, delivery considerations when operating an enterprise service bus (ESB). It describes what an ESB is (in both conceptual and practical terms), then delves in to tightly and loosely coupled considerations. A discussion of lightweight and heavyweight messaging considerations is followed by a more detailed look at lighter-weight and lightweight messaging.

What is an ESB?

An ESB is an architectural pattern that enables you to optimize the distribution of information among different types of applications across multiple locations. The ESB pattern is founded on and unifies message-oriented, event-driven and service-oriented approaches to integration. The core characteristics of an ESB (all of which should be oriented toward a service-based infrastructure) provide:

- *Standards-based application integration*
- *Support for Web services, message-based transport, and publish-and- subscribe (event-based) integration*
- *Transformation*
- *Intelligent routing*

An ESB (shown in Figure 1) provides a logical mechanism that enables separate information automation components to interact in flexible and adjustable ways that traditionally weren't possible without major financial investment. These components were also often delivered in forms that subsequently proved to be inflexible when other changes, improvements or connections were needed (to keep the original costs down).

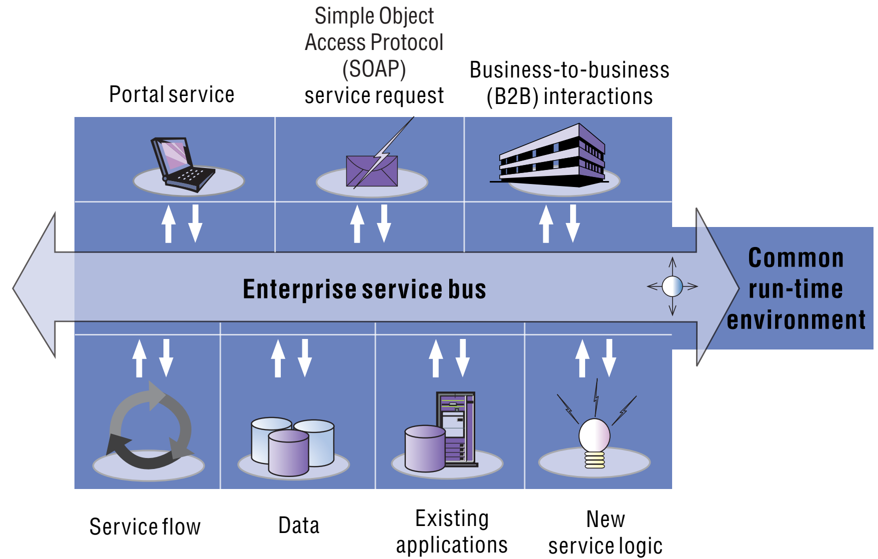

Figure 1. A conceptual view of an ESB

If this explanation appears somewhat abstract – and the ESB concept can be elusive – consider an analogy. Think of a double-decker bus. Its purpose is to transport you from one place to another (in information terms, it is to convey information from one place to another). But, whereas a double-decker bus has only one entrance, an ESB can have a number of different doors – with different entry and exit points. Each door might be of a different style and even have different-sized porches – to accommodate different sizes of people (information types) and the different ways they might choose to board or exit the bus.

These porches are as important as the doors. The porches would ask what language you prefer to speak (for example, "What protocol are you using?"). When this is understood, each porch converts everything boarding the bus to a common form. The same applies for exiting. Thus, something arriving through one porch and door in one form can leave through any other door and porch in another form. So, whatever can board the bus can also disembark – even though the departure format and protocol may be quite different to the arrival format and protocol. An ESB enables an application

that produces one form of output to submit this output to an ESB – and know that other applications will be able to receive (and use) what the original application created, without the originating application having to actually understand the destination application.

But that is not all an ESB can do. Again, consider the double-decker bus. It has two decks, or floors. The lower deck is where all the doorways and porches are located. It manages the boarding process, the rendering of information arriving into a common format and the subsequent disembarkations. But more may be needed. By paying a little extra and going upstairs to the upper deck, additional degrees of processing can be obtained – from persistence, reliability, transformation and routing to encryption, rules and content-based processing, and other value-add services. You might not want all these services all the time. But certain passengers (or their patrons) might choose to exploit these when necessary.

An ESB in practical terms

Ideally, an ESB is a modest-sized piece of software that you can deploy in multiple instances around your organization, or even between organizations. Each ESB instance is relatively lightweight, but with the inherent capability to self-organize so that two or more ESBs can link together to form a logical interconnection bus along which information can flow as required.

In contrast to a double-decker bus, which proceeds along fixed routes and at set times (traffic permitting), an ESB's value increases as it provides ever greater flexibility. By being able (in an ideal world) to install an ESB instance at any point convenient to an application (that needs to talk to another application), the combination of self-organizing ESB instances enables anything boarding at one ESB instance to be delivered through another ESB instance. The underlying communications connections between all ESB instances are delivered through a local area network (LAN), a wide area network (WAN) or combinations of these. You can also size and locate different ESB instances to suit specific requirements – at functional, departmental, enterprise or inter-enterprise levels.

The key capability that an ESB must deliver, therefore, is to link multiple ESB instances. An ESB delivers its value by operating across a network cloud, which provides the interconnections between applications and systems. Too often in the past, these connections could work together only with expensive and rigid point-to-point connections or they remained in isolation – with people acting as the middleware (the connecting mechanism) between applications. The attraction of ESBs is that they can provide automated entrance and exit doors across a logical network.

Another way of illustrating the attraction of the ESB concept is when you need to take advantage of the upper deck's value-added services. It is not necessary to locate every transformation or routing instruction on every ESB instance. Instead, high-volume, value-added services might be replicated in many places to maximize performance or throughput, whereas other value-added services might only be located in one or two places. These places can then be accessed through the self-organizing, intercommunicating ESB instances.

As a result, the inherent advantage of an ESB is that the location of all ESB instances can be arranged in a topology that is optimized to particular requirements. This can be at the functional, departmental, enterprise or inter-enterprise level – or combinations of these levels– as needed. This also enables you to deliver workload balancing, scalability and resilience. By implementing enough ESB instances, you introduce operational, as well as logical, flexibility where it is needed. In so doing, you can subsume the rigidities of, for example, dedicated point-to-point connections (for integration) into the broader ESB concept without losing the capabilities that traditional point-to-point solutions can deliver.

ESBs and the physical dimension

Talking of an ESB is correct, but can (unintentionally) be misleading. As described before, an ESB is most often a collection of physical ESB instances that self-organize to work together, to know about each other. What many find confusing is the contrast between a logical ESB and many physical ESB instances. You do not buy an *ESB*. What you purchase is the software for each ESB *instance*. The combination of many software instances working together is what delivers the ESB concept.

Conceptually, the purchase of one ESB instance can provide a logical ESB. But the practical nature of a successful ESB is that it can become mission critical. You can implement one huge, single ESB instance – running, for example, on one large centralized system – and, as a result, possess a logical ESB. However, this approach might not make operational sense. If that instance fails, so do all the connections that flow through it. Just as if the only double-decker bus on a route were to break down, that route would cease to carry passengers until that bus is repaired or replaced.

In practice, operational implementations of the ESB concept consist of several or many individual ESB instances, located around a network. Each possesses different entry and exit combinations of doors and porches, upper-deck, value-added services and the ability to communicate (among the various ESB instances). The appeal for your organization is that many ESB instances can combine to become a flexible, connected intermediary. In some cases, you might find it appropriate to accomplish this by incorporating ESB instances near centralized facilities. Alternatively, you might choose to distribute instances and facilities. Or you might implement the particular combination of these extremes that best suits your business requirements.

This is the beauty of the ESB concept. Physically it can be implemented to deliver just what you need. Logically, it is a self-organizing group of physical instances that deliver the ESB concept. For example, in broad messaging terms, an ESB can support:

- *For boarding – the push model (I want to board) and the pull model (the driver stops and tells me when to board).*
- *For exiting – the pull model (I want to get something off the bus) and the push model (the driver tells me when, where and how to disembark).*
- *Variations in between – from publish-and-subscribe to point-to-point connections to Web services support, as well as synchronous attachment and disconnection.*

An ESB based on IBM products

IBM WebSphere® MQ can provide the base for an ESB with point-to-point messaging, some publish and subscribe, and clustering. IBM WebSphere MQ Everyplace® complements WebSphere MQ with lower-level and reduced-footprint message handling. Higher in the ESB function stack, IBM WebSphere Business Integration Event Broker and IBM WebSphere Business Integration Message Broker add the upper-deck services referred to previously, like routing, transformations and rules-based processing, as well as fan-in and fan-out capabilities. If you need tight links to the Java™ development and run-time environments, IBM WebSphere Application Server can provide these links, as well as synchronous possibilities.

The *service* in an enterprise *service* bus

The term *enterprise* is well understood. Similarly, the notion of a network *bus* has been described earlier. Often one other problem emerges when people try to understand what an ESB is—namely, what does *service* encompass? Possibly the simplest way to explain service is to convey that there are three complementary dimensions to service in an ESB. The first dimension refers to the idea that a service is a well-defined, stable, loosely coupled communications interface provided by an application or software component to other applications running within a business process. This aspect of service is closely aligned with the concept of a *service-oriented architecture*, or SOA, as a means of building flexible business systems. You can also learn more about this concept by visiting **ibm.com**/software/solutions/webservices/.

The second dimension involves the translation service between different protocols. In a sense, this translation service is an elementary (but sophisticated) part of the *first hop on, and last hop off* of an ESB. After something arrives on the ESB (whether through HTTP, WebSphere MQ, Java Message Service [JMS] or so on), it must be translated into the selected language that operates on that ESB—perhaps XML, extended Structured Query Language (eSQL) or whatever else might be appropriate.

The important point here is that an ESB should not force you to translate information in a certain way. Instead, you should be able (within reasonable limits) to do more or less what you want; choices aren't imposed on you. One factor of service is that an ESB offers the choice as to what form a message should arrive in; the ESB's intertranslation service then converts that information into your chosen generic format, as well as translating it as needed to go out to your chosen destination.

The third dimension to service is associated with upper-deck, value-added possibilities. These services are relevant when information moving through an ESB requires additional work to get the results you need, such as:

- *Persistence*
- *Reliability*
- *Replay (as it relates to archiving)*
- *Transformation (in the programmatic sense, like Fahrenheit to centigrade, or kilos to pounds)*
- *Message enrichment – accessing databases (as you can do in WebSphere Business Integration Message Broker) to look up names and addresses from a customer number, before adding this data to the original message*

The essence of this third dimension to service is that activities can be automated on your behalf. Rather than have a person receive an e-mail order and print it several times, or create and send several copy e-mails to distribute an order to many required destinations, an ESB can automate these tasks. Value-added services can automate persistence or even filing. An ESB service can provide any or all of these services — whether in a database or working with an e-mail server, for example.

An upper-deck service in an ESB can perform activities where it makes little sense to have a person involved. However, a distinction should be drawn between the processing that an ESB broker delivers, where the logic to be applied is intentionally lightweight and repetitive, and the processing an application server (like WebSphere Application Server) offers,

where the logic is heavier duty. For this reason, an ESB is not a substitute or an alternative for the heavier-duty application logic that an application server is designed to deliver. An ESB should not attempt to do the work of an application server. And an application server might be overkill for ESB activities.

An ESB should facilitate the supply of data to and from the applications running on an application server. Another way to decide what should be on your ESB is to work out if processing is on the fly or not. If it is, then using an ESB is probably the appropriate approach. If it requires more work, like interacting with a customer, that element should be handed off to an application.

Tightly or loosely coupled considerations in an ESB environment

The key distinction between tightly and loosely coupled considerations in an ESB can be summarized by the differences between IBM WebSphere MQ Telemetry Transport, and WebSphere MQ or WebSphere MQ Everyplace (for example). In a tightly coupled, or *synchronous*, program model, the application has to be aware of whether it has a connection to the far end. Such awareness can be considered good or bad. The people who think it is bad tend to be adherents of the loosely coupled, or *asynchronous*, model. Conversely, people who think this awareness is good tend to be adherents of the synchronous model.

However, IBM experience has shown that the key distinguishing element boils down to the difference between attended and unattended operations. If you have an unattended device, like a telemetry device, it is valuable (and often essential) to know if connectivity exists. The application needs to do more than simply send or receive data that might one day arrive. For example, in a factory, if a temperature reading goes over a certain level, it is important to inform someone, so that actions can be taken. It is of little use if the warning is sent promptly, but only received at some point in the future, or in a way that does not facilitate an appropriate action. In unattended operation, the goal is to use the existing network connection. If this doesn't work, alternatives must be tried until a connection is successfully established. If all attempts fail, the device might have to make the autonomic decision to shift to a fail-safe operation.

Part of the price paid for synchronous or tightly coupled communication is that it increases the application developer's workload. The developer must be aware of all the possible scenarios and then create solutions to address selected ones. This makes applications larger and more complex, because different combinations of retries are attempted according to the sequence coded into the application. For example, the first connection might be through the normal internal LAN, the second might use a dial-up connection, the third connection could be by satellite and the last through Global Packet Radio Service(GPRS).

The issue is that handling all these alternatives produces extra work for the application writer. But, if you are an engineer working with a production plant and machinery, omitting them could result in disaster—a chemical plant catching fire, a refinery leaking or a food-processing system producing contaminated products. In these situations you want to avoid handing messages to an opaque middleware layer where you do not know when information might arrive. Instead, you want to process the information through a transparent middleware layer where you have sufficient control to know for certain that this mission-critical information has been delivered.

In this context, another dimension that an ESB must offer is the ability to let people choose to take direct responsibility for their data. Any ESB must be able to provide tightly coupled, or synchronous, capabilities to address such requirements, even if that means an external (to the ESB) application (monitored by a person or prompting an action by a person) must take the responsibility for the action. And you must also consider whether the status of the connection needs to be known *across* the ESB.

For example, a source application might deliver a message of some form in a synchronous or tightly coupled manner to an ESB like WebSphere Business Integration Message Broker. However, after it's inside WebSphere Business Integration Message Broker, the tightly coupled link (even if it is from a synchronous protocol like WebSphere MQ Telemetry Transport) is effectively lost when WebSphere Business Integration Message Broker accepts the input. If you need a continuous connection across the ESB (in this case,

WebSphere Business Integration Message Broker), from source through to destinations, then other considerations apply – like the destination application sending back a confirmation message along the reverse route saying your message was received.

Loosely coupled or asynchronous considerations in an ESB environment

The underlying appeal of loosely coupled, or asynchronous, solutions is that they reduce the amount of work required of application developers to make these solutions behave as intended. Rather than having to consider every circumstance and coding for the relevant ones, decoupling enables different parts of an overall process to be created by different activities. The source application need not know anything about the nature of the destination application (and vice versa). This has always been true with WebSphere MQ, which is one reason WebSphere MQ is a popular solution for large enterprises.

Adding an ESB introduces still greater flexibility. A source need only know about an ESB (like WebSphere Business Integration Message Broker) – and not all of an ESB's destinations. Similarly a destination does not need to know about individual details of every source. The ESB is the destination as far as the source is concerned (or the source where a destination is concerned).

After it has received a submission, the ESB can look after routing the source information to the one or many destinations to which it should be sent. Continuing the double-decker bus analogy, the ESB's processing capability can exploit upper-deck services to enhance, apply rules-based routing to and persist messages. The destination applications then receive the information (through the appropriate outbound doors and porches, as applicable).

Another of the decoupled services that an ESB can offer is confirmation of delivery. The source can receive a response or message indicating that delivery to the ultimate endpoint has occurred. This is important,

because applications exist that need explicit confirmation that a message has moved from source to ultimate destination. This confirmation shows that the middleware has worked as intended. Decoupled environments are ideal for organizations that don't want (or need) to handle large amounts of confirmation. Within this environment, the worry of whether sent messages are received is removed, because you can trust your ESB to complete the delivery.

An ESB can provide these capabilities – particularly in the attended application space. Instead of spending time worrying about the nuances of network connectivity (as is the case with unattended systems), the focus can be on interacting with the user and preparing the message to be sent. This applies to applications talking to each other, such as a personal digital assistant (PDA) transmitting the details of an insurance claim or a laptop computer placing an order.

In such instances, the key issue is usually not whether the claim or order has been submitted *yet*. Rather, it is whether you can be confident that it will occur *reliably*. As far as the user is concerned, pressing the submission button means that that person has completed his or her part. It is the responsibility of the ESB to help ensure that the rest happens.

The application gives the message to the middleware, which passes it to the next part of the ESB. The ESB determines what should happen next – using, for instance, the content routing logic in WebSphere Business Integration Message Broker – and then helps assure delivery to the appropriate destinations. The source application does nothing more than provide the appropriate verb to entrust the message to the middleware, from which all else happens as the message is passed to its destination.

In contrast to the tightly coupled case, users in a loosely coupled example take *less* responsibility for message transmission. The infrastructure is there to support the programmer's instructions in tightly coupled instances. In the decoupled case, the programmer expects to hand over transmission and delivery responsibilities to the infrastructure that (using something like an ESB) is capable of handling these tasks.

Lightweight and heavyweight messaging considerations in an ESB

When considering lightweight and heavyweight messages, the first step is to understand the weight of a message. Different communication environments require different solutions. On a modern enterprise LAN, you are likely to have a gigabit Ethernet. If the task is to send a message about a US$500 million transfer, you must ensure that the message arrives once and *once only*. To achieve this, you must use heavyweight messaging because it has many facilities associated (from persistence and once-only delivery to transactional capabilities and logging). Even though this message might take only a tiny proportion of the gigabit Ethernet, and even though a large overhead might be associated with relevant protocol exchanges, the value of the transaction is so great that assured end-to-end delivery in the correct manner is paramount. To lose or duplicate such a message could be hugely expensive. Yet the network cost is trivial, the delivery is high speed and latency is low.

So, in this instance, enterprise-strength messaging is practical—and preferable. It can be made heavyweight, even if the actual requirements are lightweight—because the network cost is negligible within the context of the enterprise WAN or LAN. Also, any messaging features that increase the weight—longer fields, more-informative date stamps or space for future enhancements — don't materially add to the cost per message. The cost can be entirely justified by the operational savings achieved through a consistent approach to messaging.

Most messaging has been heavyweight in the past, because it operated within an enterprise environment where the networking cost was not significant. On a per-message basis, this has meant that the characteristics and requirements associated with heavyweight messaging have been accepted as the norm. However, these habits can prove expensive, as well as inappropriate, if the real need is for lightweight messaging. This leads to a parallel conclusion. The need for lightweight messaging might, in practice, be more common than has been appreciated in the past. Why? Because lightweight messaging is applicable to a wide range of application scenarios—even if it can be more difficult or expensive (primarily through unfamiliarity) to achieve.

Lighter-weight messaging

The advent of the Internet, along with the arrival of messaging systems optimized for small, pervasive devices, changed the overall picture to favor lighter-weight and lightweight messaging. Across the Internet, bandwidth tends to be limited at the endpoints. And the basic nature of the Internet involves multihop connections from source to destination. You only have to do a trace-route search to understand just how many hops are involved. This means that if you are sending small messages, a large overhead per message is associated with submission through to delivery. This can be both expensive and undesirable, primarily because you have to wait for a long time for the messages to be transmitted. A *verbose protocol* means that the latency is much higher as the verbosity has to be handled at every one of the Internet's many hops.

Limited bandwidth and minimizing latency mean that a heavyweight messaging protocol might not be appropriate. This is not to imply that heavyweight messaging is impossible over the Internet, but it is frequently less than optimal. In other words, the assumptions that continue to underpin WebSphere MQ in an enterprise WAN or LAN environment (such as exactly once-only delivery in all circumstances) are not necessarily appropriate to many Internet connections. So, if there are circumstances where the full rigor of heavyweight messaging is not needed, then lightweight alternatives,

with less capability (and overhead) are a suitable option. For example, when you buy a book from a retailer over the Web, losing a connection is not the end of the world. You just reconnect and start again. Similarly, in a securities-trading environment, losing a price tick does not matter if another one is coming along shortly after. Just enough is good enough.

The Internet creates new opportunities to communicate with tens or hundreds of thousands of applets running in Web browsers. For such applications, there is a need for lighter-weight protocols that can be scalable and use fewer resources. (Heavyweight messaging tends to be less scalable because of the size of messages, along with all the protocol handling, logging, enabling of persistence and automated recovery, and so on) Instead, on the Internet, scale counts. For example, 450 000 people might want to be watching live Wimbledon tennis scores. They want the latest score as soon as it happens, as soon as possible. However, if they miss a point, it doesn't really matter, because the next point scored supersedes the previous score, such as moving from 40 – 15 to 40 – 30.

To obtain rapid delivery, in as near real time as the inherent latency of the Internet permits, you need each piece of transmitted information to have as little overhead as possible. The lighter-weight the message (if possible within a single TCP/IP network packet), the less time it takes to transmit it across the Internet. And any bandwidth constraints at the receiving end (like a slow modem connection) will matter less. There is a chance that a score or a stock tick will not be delivered at all, but the next update supersedes the message that did not arrive.

The IBM WebSphere MQ Real Time protocol was created for this reason. It enhances the capabilities of an ESB by enabling lighter-weight messages to be created and received—without the impositions required by heavier-weight WebSphere MQ. A single message can be published to the Internet and it might then be seen by 10 000 or 450 000 or even 10 000 000 people. Not only is this more efficient for the publisher (who does not need to create individually addressed messages for transmission), but it is better for the Internet as a whole, because the network impact of these major events is reduced.

Lightweight messaging

Lightweight messaging is driven almost entirely by the existence of small pervasive computing devices installed in anything from machinery, cars and trucks, sensors on transmission lines or pumps, and even in domestic products, such as refrigerators, air conditioners and oil tanks (see Figure 2). The key difference is in the quality of service and scaling – with the additional problem that the available bandwidth can be both low and very expensive. Typically, a 9600-baud modem is pretty much the industry average. If satellites are being used (from remote or inaccessible locations where telephone lines are not available), the cost might be as much as five cents per byte transmitted or received.

The good news is that the data to be sent or received tends to be much smaller – not a person's banking profile (of several thousand bytes), but a simple command to turn something on or off, a report of a temperature reading or an alert that a known condition has changed. This means that the data to be sent should be small – and the overhead associated with sending it needs to be as small as practical. There is little point in sending two bytes of data if this has 10KB of routing and other overhead.

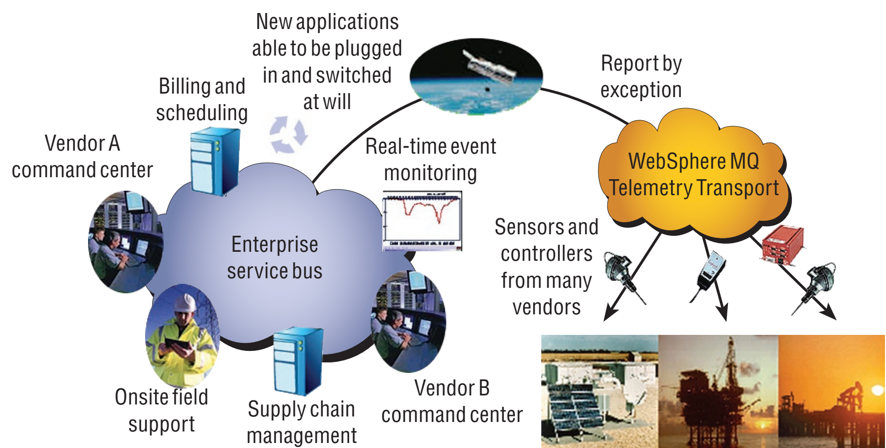

Figure 2. Lightweight messaging with WebSphere MQ Telemetry Transport

As already discussed, one of the things that makes an ESB appealing is that it can support the capability to receive and send out many different forms of messaging through its differing doors and porches. In the context of lightweight to heavyweight messages, the following represent the minimum message header sizes.

- *For WebSphere MQ Telemetry Transport: 2 bytes*
- *For WebSphere MQ Everyplace: 18 bytes*
- *For WebSphere MQ Real Time: 24 bytes*
- *For WebSphere MQ: 480 bytes*

If a satellite or similar connection is being paid for on a per-byte basis, the attractions of a WebSphere MQ Telemetry Transport or WebSphere MQ Everyplace message are apparent (assuming other associated constraints are acceptable). An ESB enables an enterprise to choose what to use where and to obtain the best of all relevant options. The actual choice between lightweight, lighter-weight or heavyweight messaging will emerge from a consideration of function, as well as the underlying resilience and bandwidth of the various communications options. But, with an ESB, combinations of options are possible—which is why flexibility is improved.

Conclusion

It is true that with lightweight messaging, you can't do everything. For example, you can't readily bracket transactions across multiple messages, although it is possible to program ways to address such a requirement when using lightweight messaging. At the same time, using heavyweight messaging to deliver tennis or golf scores, or to carry traffic from remote sensors using a satellite is too expensive in practice (although, again, it is possible). The benefit of an ESB is that the best of each form of messaging can be selected and combined to produce solutions that do not require an expensive, custom-developed or rigid infrastructure.

With an ESB you could even designate most of your messages as lightweight, but when the occasional important or valuable one appears, the ESB can decide to communicate it—because of its content—using a heavyweight mechanism, like WebSphere MQ rather than WebSphere MQ Telemetry Transport. Similarly, loosely and tightly coupled components can be brought together as required. An ESB enables different styles and approaches so that you can satisfy your organization's immediate requirements in the first instance, and then modify or improve the components as business circumstances evolve or change.

In much the same way, the notion that an ESB is an overall concept made up of one or many physical ESB instances working together to automate what previously required human intervention, is attractive. While an ESB does not try to do everything, and should not (levels of business processing above an ESB exist, such as workflow and process choreography, as found in IBM WebSphere Business Integration Server), an ESB can automate the routine integration of applications, systems and sensors in a way that wasn't possible before.

Many argue that an ESB is the full expression of the true value proposition of middleware—because it is the intermediary that delivers flexibility between applications. An ESB enables integration between applications, which is what most organizations are looking for because they can no longer tolerate the inability of their existing and new IT investments to work together unless people are involved. If systems and applications can work together to do the repetitive and regular tasks, people can be free to concentrate on higher-skilled activities with higher returns.

In more technical terms, introducing an ESB removes the need to worry about the granular detail of how connections are made between applications. Just as TCP/IP and its protocols are the backbone of the Internet, because they route all your requests (whether from a Web browser, e-mail or instant messaging) to the selected destination, an ESB moves this to a higher and more sophisticated plane. The broad concept is the same although what is produced can be much more valuable.

Using an ESB, organizations start to eliminate today's common scenario of static links that have to be physically administered (from definition through configuration and maintenance) by people. This is too expensive, as well as too inflexible, to meet today's business needs. Instead, an ESB is the realization of the value proposition in which function is placed above wire connectivity (TCP/IP) to provide increased and automated dependability and flexibility between applications.

For more information

To learn more about the enterprise service bus, visit:

ibm.com/software/integration/esb

© Copyright IBM Corporation 2005

IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

Produced in the United States of America
01-05
All Rights Reserved

Everyplace, IBM, the IBM logo, the On Demand Business logo and WebSphere are trademarks of International Business Machines Corporation in the United States, other countries or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries or both.

Other company, product and service names may be trademarks or service marks of others.

This information contained in this document is provided AS IS. Any person or organization using the information is solely responsible for any and all consequences of such use. IBM accepts no responsibility for such consequences.

All statements regarding IBM future direction or intent are subject to change or withdrawal without notice and represent goals and objectives only.

G224-9149-00