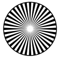
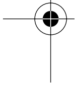
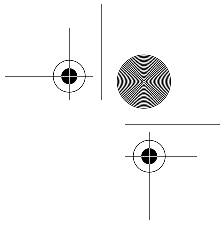


Chapter 2

Integration Styles

Introduction

Enterprise integration is the task of making disparate applications work together to produce a unified set of functionality. These applications can be custom developed in house or purchased from third-party vendors. They likely run on multiple computers, which may represent multiple platforms, and may be geographically dispersed. Some of the applications may be run outside of the enterprise by business partners or customers. Other applications might not have been designed with integration in mind and are difficult to change. These issues and others like them make application integration complicated. This chapter explores multiple integration approaches that can help overcome these challenges.

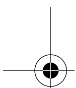
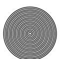
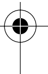
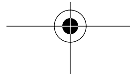
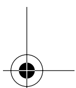
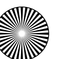
Application Integration Criteria

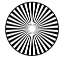
What makes good application integration? If integration needs were always the same, there would be only one integration style. Yet, like any complex technological effort, application integration involves a range of considerations and consequences that should be taken into account for any integration opportunity.

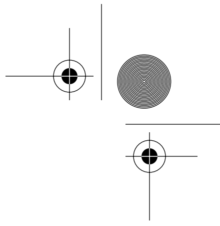
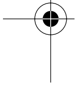
The fundamental criterion is whether to use **application integration** at all. If you can develop a single, standalone application that doesn't need to collaborate with any other applications, you can avoid the whole integration issue entirely. Realistically, though, even a simple enterprise has multiple applications that need to work together to provide a unified experience for the enterprise's employees, partners, and customers.

The following are some other main decision criteria.

Application coupling—Integrated applications should minimize their dependencies on each other so that each can evolve without causing problems to the others. As explained in Chapter 1, "Solving Integration Problems Using Patterns," tightly coupled applications make numerous assumptions about

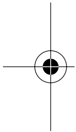
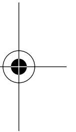


Hohpe_book.fm Page 40 Friday, September 19, 2003 2:51 AM



how the other applications work; when the applications change and break those assumptions, the integration between them breaks. Therefore, the interfaces for integrating applications should be specific enough to implement useful functionality but general enough to allow the implementation to change as needed.

Intrusiveness—When integrating an application into an enterprise, developers should strive to minimize both changes to the application and the amount of integration code needed. Yet, changes and new code are often necessary to provide good integration functionality, and the approaches with the least impact on the application may not provide the best integration into the enterprise.

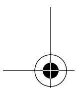
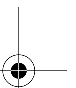
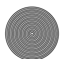
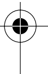
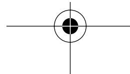
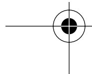
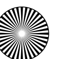
Technology selection—Different integration techniques require varying amounts of specialized software and hardware. Such tools can be expensive, can lead to vendor lock-in, and can increase the learning curve for developers. On the other hand, creating an integration solution from scratch usually results in more effort than originally intended and can mean reinventing the wheel.

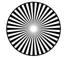



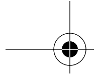
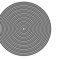
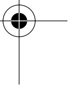
Data format—Integrated applications must agree on the format of the data they exchange. Changing existing applications to use a unified data format may be difficult or impossible. Alternatively, an intermediate translator can unify applications that insist on different data formats. A related issue is **data format evolution and extensibility—**how the format can change over time and how that change will affect the applications.

Data timeliness—Integration should minimize the length of time between when one application decides to share some data and other applications have that data. This can be accomplished by exchanging data frequently and in small chunks. However, chunking a large set of data into small pieces may introduce inefficiencies. Latency in data sharing must be factored into the integration design. Ideally, receiver applications should be informed as soon as shared data is ready for consumption. The longer sharing takes, the greater the opportunity for applications to get out of sync and the more complex integration can become.

Data or functionality—Many integration solutions allow applications to share not only data but functionality as well, because sharing of functionality can provider better abstraction between the applications. Even though invoking functionality in a remote application may seem the same as invoking local functionality, it works quite differently, with significant consequences for how well the integration works.

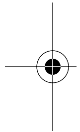
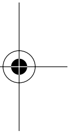


Hohpe_book.fm Page 41 Friday, September 19, 2003 2:51 AM




Remote Communication—Computer processing is typically synchronous—that is, a procedure waits while its subprocedure executes. However, calling a remote subprocedure is much slower than a local one so that a procedure may not want to wait for the subprocedure to complete; instead, it may want to invoke the subprocedure asynchronously, that is, starting the subprocedure but continuing with its own processing simultaneously. Asynchronicity can make for a much more efficient solution, but such a solution is also more complex to design, develop, and debug.

Reliability—Remote connections are not only slow, but they are much less reliable than a local function call. When a procedure calls a subprocedure inside a single application, it's a given that the subprocedure is available. This is not necessarily true when communicating remotely; the remote application may not even be running or the network may be temporarily unavailable. Reliable, asynchronous communication enables the source application to go on to other work, confident that the remote application will act sometime later.

So, as you can see, there are several different criteria that must be considered when choosing and designing an integration approach. The question then becomes, Which integration approach best addresses which of these criteria?

Application Integration Options

There is no one integration approach that addresses all criteria equally well. Therefore, multiple approaches for integrating applications have evolved over time. The various approaches can be summed up in four main integration styles.

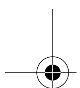
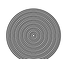
File Transfer (43)—Have each application produce files of shared data for others to consume and consume files that others have produced.

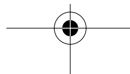
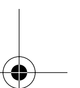
Shared Database (47)—Have the applications store the data they wish to share in a common database.

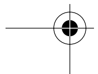
Remote Procedure Invocation (50)—Have each application expose some of its procedures so that they can be invoked remotely, and have applications invoke those to initiate behavior and exchange data.

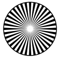
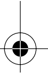
Messaging (53)—Have each application connect to a common messaging system, and exchange data and invoke behavior using messages.

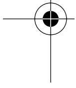
This chapter presents each style as a pattern. The four patterns share the same problem statement—the need to integrate applications—and very similar contexts. What differentiates them are the forces searching for a more elegant



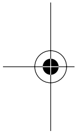
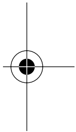
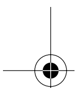
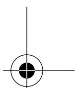
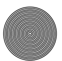



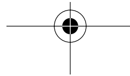
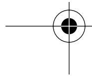
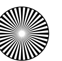


solution. Each pattern builds on the last, looking for a more sophisticated approach to address the shortcomings of its predecessors. Thus, the pattern order reflects an increasing order of sophistication, but also increasing complexity.

The trick is not to choose one style to use every time but to choose the *best* style for a particular integration opportunity. Each style has its advantages and disadvantages. Applications may integrate using multiple styles so that each point of integration takes advantage of the style that suits it best. Likewise, an application may use different styles to integrate with different applications, choosing the style that works best for the other application. As a result, many integration approaches can best be viewed as a hybrid of multiple integration styles. To support this type of integration, many integration and EAI middleware products employ a combination of styles, all of which are effectively hidden in the product's implementation.

The patterns in the remainder of this book expand on the *Messaging* (53) integration style. We focus on messaging because we believe that it provides a good balance between the integration criteria but is also the most difficult style to work with. As a result, messaging is still the least well understood of the integration styles and a technology ripe with patterns that quickly explain how to use it best. Finally, messaging is the basis for many commercial EAI products, so explaining how to use messaging well also goes a long way in teaching you how to use those products. The focus of this section is to highlight the issues involved with application integration and how messaging fits into the mix.

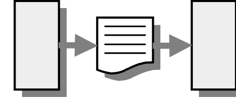






Hohpe_book.fm Page 43 Friday, September 19, 2003 2:51 AM

File Transfer

by Martin Fowler



An enterprise has multiple applications that are being built independently, with different languages and platforms.

How can I integrate multiple applications so that they work together and can exchange information?

In an ideal world, you might imagine an organization operating from a single, cohesive piece of software, designed from the beginning to work in a unified and coherent way. Of course, even the smallest operations don't work like that. Multiple pieces of software handle different aspects of the enterprise. This is due to a host of reasons.

- People buy packages that are developed by outside organizations.
- Different systems are built at different times, leading to different technology choices.
- Different systems are built by different people whose experience and preferences lead them to different approaches to building applications.
- Getting an application out and delivering value is more important than ensuring that integration is addressed, especially when that integration doesn't add any value to the application under development.

As a result, any organization has to worry about sharing information between very divergent applications. These can be written in different languages, based on different platforms, and have different assumptions about how the business operates.

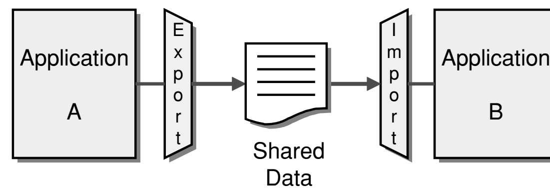
Tying together such applications requires a thorough understanding of how to link together applications on both the business and technical levels. This is a lot easier if you minimize what you need to know about how each application works.

Hohpe_book.fm Page 44 Friday, September 19, 2003 2:51 AM

What is needed is a common data transfer mechanism that can be used by a variety of languages and platforms but that feels natural to each. It should require a minimal amount of specialized hardware and software, making use of what the enterprise already has available.

Files are a universal storage mechanism, built into any enterprise operating system and available from any enterprise language. The simplest approach would be to somehow integrate the applications using files.

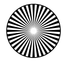
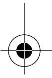
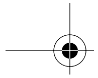
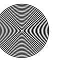
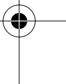
Have each application produce files that contain the information the other applications must consume. Integrators take the responsibility of transforming files into different formats. Produce the files at regular intervals according to the nature of the business.



An important decision with files is what format to use. Very rarely will the output of one application be exactly what's needed for another, so you'll have to do a fair bit of processing of files along the way. This means not only that all the applications that use a file have to read it, but that you also have to be able to use processing tools on it. As a result, standard file formats have grown up over time. Mainframe systems commonly use data feeds based on the file system formats of COBOL. UNIX systems use text-based files. The current method is to use XML. An industry of readers, writers, and transformation tools has built up around each of these formats.

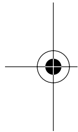
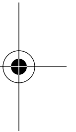
Another issue with files is when to produce them and consume them. Since there's a certain amount of effort required to produce and process a file, you usually don't want to work with them too frequently. Typically, you have some regular business cycle that drives the decision: nightly, weekly, quarterly, and so on. Applications get used to when a new file is available and processes it at its time.

The great advantage of files is that integrators need no knowledge of the internals of an application. The application team itself usually provides the file. The file's contents and format are negotiated with integrators, although if a



Hohpe_book.fm Page 45 Friday, September 19, 2003 2:51 AM




package is used, the choices are often limited. The integrators then deal with the transformations required for other applications, or they leave it up to the consuming applications to decide how they want to manipulate and read the file. As a result, the different applications are quite nicely decoupled from each other. Each application can make internal changes freely without affecting other applications, providing they still produce the same data in the files in the same format. The files effectively become the public interface of each application.

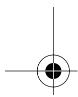
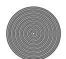
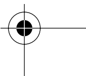
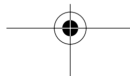
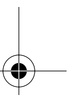
Part of what makes *File Transfer* simple is that no extra tools or integration packages are needed, but that also means that developers have to do a lot of the work themselves. The applications must agree on file-naming conventions and the directories in which they appear. The writer of a file must implement a strategy to keep the file names unique. The applications must agree on which one will delete old files, and the application with that responsibility will have to know when a file is old and no longer needed. The applications will need to implement a locking mechanism or follow a timing convention to ensure that one application is not trying to read the file while another is still writing it. If all of the applications do not have access to the same disk, then some application must take responsibility for transferring the file from one disk to another.

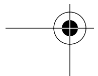



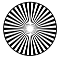
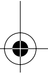
One of the most obvious issues with *File Transfer* is that updates tend to occur infrequently, and as a result systems can get out of synchronization. A customer management system can process a change of address and produce an extract file each night, but the billing system may send the bill to an old address on the same day. Sometimes lack of synchronization isn't a big deal. People often expect a certain lag in getting information around, even with computers. At other times the result of using stale information is a disaster. When deciding on when to produce files, you have to take the freshness needs of consumers into account.

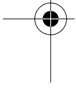
In fact, the biggest problem with staleness is often on the software development staff themselves, who frequently must deal with data that isn't quite right. This can lead to inconsistencies that are difficult to resolve. If a customer changes his address on the same day with two different systems, but one of them makes an error and gets the wrong street name, you'll have two different addresses for a customer. You'll need some way to figure out how to resolve this. The longer the period between file transfers, the more likely and more painful this problem can become.

Of course, there's no reason that you can't produce files more frequently. Indeed, you can think of *Messaging* (53) as *File Transfer* where you produce a file with every change in an application. The problem then is managing all the files that get produced, ensuring that they are all read and that none get lost. This goes beyond what file system–based approaches can do, particularly since

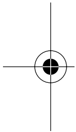
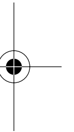
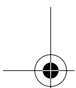
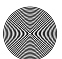
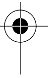
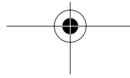
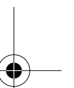
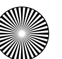
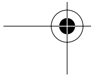




Hohpe_book.fm Page 46 Friday, September 19, 2003 2:51 AM



there are expensive resource costs associated with processing a file, which can get prohibitive if you want to produce lots of files quickly. As a result, once you get to very fine-grained files, it's easier to think of them as *Messaging* (53).

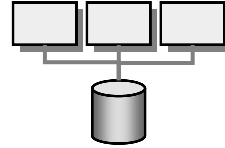
To make data available more quickly and enforce an agreed-upon set of data formats, use a *Shared Database* (47). To integrate applications' functionality rather than their data, use *Remote Procedure Invocation* (50). To enable frequent exchanges of small amounts of data, perhaps used to invoke remote functionality, use *Messaging* (53).

Hohpe_book.fm Page 47 Friday, September 19, 2003 2:51 AM

Shared Database

by Martin Fowler



An enterprise has multiple applications that are being built independently, with different languages and platforms. The enterprise needs information to be shared rapidly and consistently.

How can I integrate multiple applications so that they work together and can exchange information?

File Transfer (43) enables applications to share data, but it can lack timeliness—yet timeliness of integration is often critical. If changes do not quickly work their way through a family of applications, you are likely to make mistakes due to the staleness of the data. For modern businesses, it is imperative that everyone have the latest data. This not only reduces errors, but also increases people's trust in the data itself.

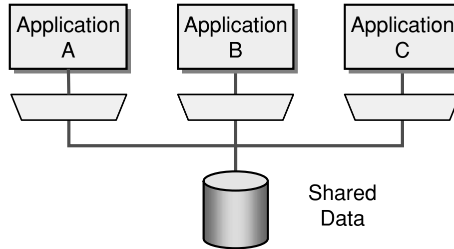
Rapid updates also allow inconsistencies to be handled better. The more frequently you synchronize, the less likely you are to get inconsistencies and the less effort they are to deal with. But however rapid the changes, there are still going to be problems. If an address is updated inconsistently in rapid succession, how do you decide which one is the true address? You could take each piece of data and say that one application is the master source for that data, but then you'd have to remember which application is the master for which data.

File Transfer (43) also may not enforce data format sufficiently. Many of the problems in integration come from incompatible ways of looking at the data. Often these represent subtle business issues that can have a huge effect. A geological database may define an oil well as a single drilled hole that may or may not produce oil. A production database may define a well as multiple holes covered by a single piece of equipment. These cases of *semantic dissonance* are much harder to deal with than inconsistent data formats. (For a much deeper discussion of these issues, it's really worth reading *Data and Reality* [Kent].) What is needed is a central, agreed-upon datastore that all of the applications share so each has access to any of the shared data whenever it needs it.

Hohpe_book.fm Page 48 Friday, September 19, 2003 2:51 AM

Integrate applications by having them store their data in a single *Shared Database,* and define the schema of the database to handle all the needs of the different applications.

Shared Database

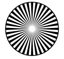
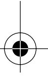
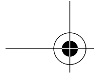
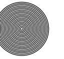
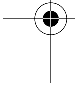
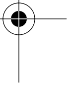


If a family of integrated applications all rely on the same database, then you can be pretty sure that they are always consistent all of the time. If you do get simultaneous updates to a single piece of data from different sources, then you have transaction management systems that handle that about as gracefully as it ever can be managed. Since the time between updates is so small, any errors are much easier to find and fix.

Shared Database is made much easier by the widespread use of SQL-based relational databases. Pretty much all application development platforms can work with SQL, often with quite sophisticated tools. So you don't have to worry about multiple file formats. Since any application pretty much has to use SQL anyway, this avoids adding yet another technology for everyone to master.

Since every application is using the same database, this forces out problems in semantic dissonance. Rather than leaving these problems to fester until they are difficult to solve with transforms, you are forced to confront them and deal with them before the software goes live and you collect large amounts of incompatible data.

One of the biggest difficulties with *Shared Database* is coming up with a suitable design for the shared database. Coming up with a unified schema that can meet the needs of multiple applications is a very difficult exercise, often resulting in a schema that application programmers find difficult to work with. And if the technical difficulties of designing a unified schema aren't enough, there are also severe political difficulties. If a critical application is likely to suffer delays in order to work with a unified schema, then often there is irresistible pressure to separate. Human conflicts between departments often exacerbate this problem.

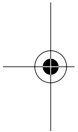
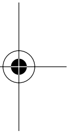
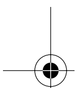
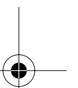
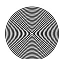
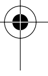
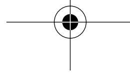
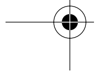
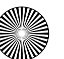







Another, harder limit to *Shared Database* is external packages. Most packaged applications won't work with a schema other than their own. Even if there is some room for adaptation, it's likely to be much more limited than integrators would like. Adding to the problem, software vendors usually reserve the right to change the schema with every new release of the software.

This problem also extends to integration after development. Even if you can organize all your applications, you still have an integration problem should a merger of companies occur.

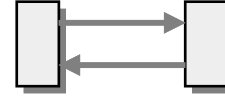
Multiple applications using a *Shared Database* to frequently read and modify the same data can turn the database into a performance bottleneck and can cause deadlocks as each application locks others out of the data. When applications are distributed across multiple locations, accessing a single, shared database across a wide-area network is typically too slow to be practical. Distributing the database as well allows each application to access the database via a local network connection, but confuses the issue of which computer the data should be stored on. A distributed database with locking conflicts can easily become a performance nightmare.

To integrate applications' functionality rather than their data, use *Remote Procedure Invocation* (50). To enable frequent exchanges of small amounts of data using a format per datatype rather than one universal schema, use *Messaging* (53).

Hohpe_book.fm Page 50 Friday, September 19, 2003 2:51 AM

Remote Procedure Invocation



by Martin Fowler

An enterprise has multiple applications that are being built independently, with different languages and platforms. The enterprise needs to share data and processes in a responsive way.

How can I integrate multiple applications so that they work together and can exchange information?

File Transfer (43) and *Shared Database* (47) enable applications to share their data, which is an important part of application integration, but just sharing data is often not enough. Changes in data often require actions to be taken across different applications. For example, changing an address may be a simple change in data, or it may trigger registration and legal processes to take into account different rules in different legal jurisdictions. Having one application invoke such processes directly in others would require applications to know far too much about the internals of other applications.

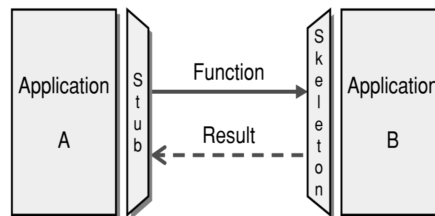
This problem mirrors a classic dilemma in application design. One of the most powerful structuring mechanisms in application design is encapsulation, where modules hide their data through a function call interface. In this way, they can intercept changes in data to carry out the various actions they need to perform when the data is changed. *Shared Database* (47) provides a large, unencapsulated data structure, which makes it much harder to do this. *File Transfer* (43) allows an application to react to changes as it processes the file, but the process is delayed.

The fact that *Shared Database* (47) has unencapsulated data also makes it more difficult to maintain a family of integrated applications. Many changes in any application can trigger a change in the database, and database changes have a considerable ripple effect through every application. As a result, organizations that use *Shared Database* (47) are often very reluctant to change the database, which means that the application development work is much less responsive to the changing needs of the business.

Hohpe_book.fm Page 51 Friday, September 19, 2003 2:51 AM

What is needed is a mechanism for one application to invoke a function in another application, passing the data that needs to be shared and invoking the function that tells the receiver application how to process the data.

Develop each application as a large-scale object or component with encapsulated data. Provide an interface to allow other applications to interact with the running application.

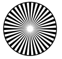


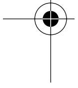
Remote Procedure Invocation applies the principle of encapsulation to integrating applications. If an application needs some information that is owned by another application, it asks that application directly. If one application needs to modify the data of another, it does so by making a call to the other application. This allows each application to maintain the integrity of the data it owns. Furthermore, each application can alter the format of its internal data without affecting every other application.

A number of technologies, such as CORBA, COM, .NET Remoting, and Java RMI, implement *Remote Procedure Invocation* (also referred to as Remote Procedure Call, or RPC). These approaches vary as to how many systems support them and their ease of use. Often these environments add additional capabilities, such as transactions. For sheer ubiquity, the current favorite is Web services, using standards such as SOAP and XML. A particularly valuable feature of Web services is that they work easily with HTTP, which is easy to get through firewalls.

The fact that there are methods that wrap the data makes it easier to deal with semantic dissonance. Applications can provide multiple interfaces to the same data, allowing some clients to see one style and others a different style. Even updates can use multiple interfaces. This provides a lot more ability to support multiple points of view than can be achieved by relational views. However, it is awkward for integrators to add transformation components, so each application has to negotiate its interface with its neighbors.

Hohpe_book.fm Page 52 Friday, September 19, 2003 2:51 AM

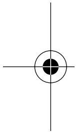
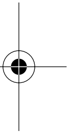
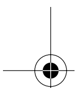
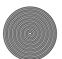
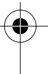
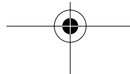
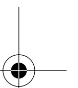
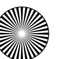
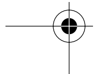




Since software developers are used to procedure calls, *Remote Procedure Invocation* fits in nicely with what they are already used to. Actually, this is more of a disadvantage than an advantage. There are big differences in performance and reliability between remote and local procedure calls. If people don't understand these, then *Remote Procedure Invocation* can lead to slow and unreliable systems (see [Waldo], [EAA]).

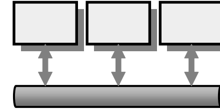
Although encapsulation helps reduce the coupling of the applications by eliminating a large shared data structure, the applications are still fairly tightly coupled together. The remote calls that each system supports tend to tie the different systems into a growing knot. In particular, sequencing—doing certain things in a particular order—can make it difficult to change systems independently. These types of problems often arise because issues that aren't significant within a single application become so when integrating applications. People often design the integration the way they would design a single application, unaware that the rules of the engagement change dramatically.

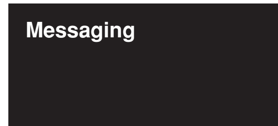
To integrate applications in a more loosely coupled, asynchronous fashion, use *Messaging* (53) to enable frequent exchanges of small amounts of data, ones that are perhaps used to invoke remote functionality.

Hohpe_book.fm Page 53 Friday, September 19, 2003 2:51 AM

Messaging





An enterprise has multiple applications that are being built independently, with different languages and platforms. The enterprise needs to share data and processes in a responsive way.

How can I integrate multiple applications so that they work together and can exchange information?

File Transfer (43) and *Shared Database* (47) enable applications to share their data but not their functionality. *Remote Procedure Invocation* (50) enables applications to share functionality, but it tightly couples them as well. Often the challenge of integration is about making collaboration between separate systems as timely as possible, without coupling systems together in such a way that they become unreliable either in terms of application execution or application development.

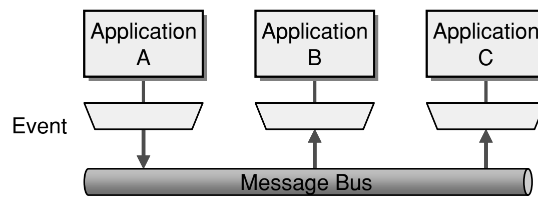
File Transfer (43) allows you to keep the applications well decoupled but at the cost of timeliness. Systems just can't keep up with each other. Collaborative behavior is way too slow. *Shared Database* (47) keeps data together in a responsive way but at the cost of coupling everything to the database. It also fails to handle collaborative behavior.

Faced with these problems, *Remote Procedure Invocation* (50) seems an appealing choice. But extending a single application model to application integration dredges up plenty of other weaknesses. These weaknesses start with the essential problems of distributed development. Despite that RPCs look like local calls, they don't behave the same way. Remote calls are slower, and they are much more likely to fail. With multiple applications communicating across an enterprise, you don't want one application's failure to bring down all of the other applications. Also, you don't want to design a system assuming that calls are fast, and you don't want each application knowing the details about other applications, even if it's only details about their interfaces.

What we need is something like *File Transfer* (43) in which lots of little data packets can be produced quickly and transferred easily, and the receiver application is automatically notified when a new packet is available for consumption.

The transfer needs a retry mechanism to make sure it succeeds. The details of any disk structure or database for storing the data needs to be hidden from the applications so that, unlike *Shared Database* (47), the storage schema and details can be easily changed to reflect the changing needs of the enterprise. One application should be able to send a packet of data to another application to invoke behavior in the other application, like *Remote Procedure Invocation* (50), but without being prone to failure. The data transfer should be asynchronous so that the sender does not need to wait on the receiver, especially when retry is necessary.

Use *Messaging* to transfer packets of data frequently, immediately, reliably, and asynchronously, using customizable formats.



Asynchronous messaging is fundamentally a pragmatic reaction to the problems of distributed systems. Sending a message does not require both systems to be up and ready at the same time. Furthermore, thinking about the communication in an asynchronous manner forces developers to recognize that working with a remote application is slower, which encourages design of components with high cohesion (lots of work locally) and low adhesion (selective work remotely).

Messaging systems also allow much of the decoupling you get when using *File Transfer* (43). Messages can be transformed in transit without either the sender or receiver knowing about the transformation. The decoupling allows integrators to choose between broadcasting messages to multiple receivers, routing a message to one of many receivers, or other topologies. This separates integration decisions from the development of the applications. Since human issues tend to separate application development from application integration, this approach works with human nature rather than against it.

The transformation means that separate applications can have quite different conceptual models. Of course, this means that semantic dissonance will occur.

Hohpe_book.fm Page 55 Friday, September 19, 2003 2:51 AM

However, the messaging viewpoint is that the measures used by *Shared Database* (47) to avoid semantic dissonance are too complicated to work in practice. Also, semantic dissonance is going to occur with third-party applications and with applications added as part of a corporate merger, so the messaging approach is to address the issue rather than design applications to avoid it.

By sending small messages frequently, you also allow applications to collaborate behaviorally as well as share data. If a process needs to be launched once an insurance claim is received, it can be done immediately by sending a message when a single claim comes in. Information can be requested and a reply made rapidly. While such collaboration isn't going to be as fast as *Remote Procedure Invocation* (50), the caller needn't stop while the message is being processed and the response returned. And messaging isn't as slow as many people think—many messaging solutions originated in the financial services industry where thousands of stock quotes or trades have to pass through a messaging system every second.

This book is about *Messaging*, so you can safely assume that we consider *Messaging* to be generally the best approach to enterprise application integration. You should not assume, however, that it is free of problems. The high frequency of messages in *Messaging* reduces many of the inconsistency problems that bedevil *File Transfer* (43), but it doesn't remove them entirely. There are still going to be some lag problems with systems not being updated quite simultaneously. Asynchronous design is not the way most software people are taught, and as a result there are many different rules and techniques in place. The messaging context makes this a bit easier than programming in an asynchronous application environment like X Windows, but asynchrony still has a learning curve. Testing and debugging are also harder in this environment.

The ability to transform messages has the nice benefit of allowing applications to be much more decoupled from each other than in *Remote Procedure Invocation* (50). But this independence does mean that integrators are often left with writing a lot of messy glue code to fit everything together.

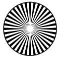
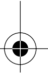
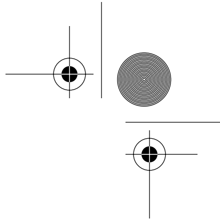
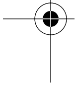
Once you decide that you want to use *Messaging* for system integration, there are a number of new issues to consider and practices you can employ.

How do you transfer packets of data?

A sender sends data to a receiver by sending a *Message* (66) via a *Message Channel* (60) that connects the sender and receiver.

How do you know where to send the data?

If the sender does not know where to address the data, it can send the data to a *Message Router* (78), which will direct the data to the proper receiver.



Hohpe_book.fm Page 56 Friday, September 19, 2003 2:51 AM



How do you know what data format to use?

If the sender and receiver do not agree on the data format, the sender can direct the data to a *Message Translator* (85) that will convert the data to the receiver's format and then forward the data to the receiver.

If you're an application developer, how do you connect your application to the messaging system?

An application that wishes to use messaging will implement *Message Endpoints* (95) to perform the actual sending and receiving.