Information Technology Supporting Documentation
Commonwealth of Pennsylvania
Governor's Office of Administration/Office for Information Technology

STD Number:	**BPD-INT001B**	
STD Title:	**Message-Oriented Middleware Integration Best Practice Guidelines**	
Issued by:	**Deputy Secretary for Information Technology**	
Date Issued: September 7, 2006	**Date Revised: October 25, 2010**	
Domain:	**Integration**	
Discipline:	**Messaging**	
Technology Area:	**Message-Oriented Middleware**	
Referenced by:	**ITP-INT001**	

Revision History Date:	**Description:**
10/25/2010	**ITP Refresh**

Introduction/Executive

Summary:

Integration is typically required when one application needs to access something contained in another application. The access can be broken into two broad categories: data and business logic.

Ideally, existing data and/or business logic is to be reused when possible. There are multiple approaches when reusing existing data and business logic to integrate applications. This document presents those approaches with architectural patterns and solutions for application integration, and it provides operational best practices for integration solutions.

Main Document Content:

Application Integration Approaches

No single approach is used to integrate applications. There are a number of approaches used to share data and business logic. These methods include:

1. File Transfer
2. Shared Databases
3. Remote Procedure Calls
4. Web Services
5. Messaging

No one approach is best for all integration options. Many factors go into choosing an integration approach. Those factors are addressed in the section, ***Choosing an Integration Approach***. This section identifies the approaches.

1. **File Transfer** – A file transfer occurs when one application produces a file and that file is then sent to a consuming application. In this instance, the producing application has data that another application requires. From a broader perspective, if the two applications are viewed as part of a larger system, the consuming application executes business logic as part of the larger system and the data it requires for the execution of the business logic comes from the producing application. The data in the transferred file serves to decouple the business logic of the applications.

   The primary issues surrounding the use of file transfers are timeliness of the file transfer, security of the data, format of the data in the file, and validity of the data. File transfers, in their simplest form,

Predefined supplemental document type codes are listed below:
APP = Appendix **BPD** = Best Practice Document **GEN** = General Information Document
OPD = Operations Document **RFD** = Existing Supporting Document Referenced by this ITP **WHP** = White Paper

utilize few system resources to manage the transfer but often require human intervention. Human intervention can make file transfer time a consuming, error-prone, and costly integration solution.

2. **Shared Database** – A database shared between two or more applications abstracts data into a form that is useful as input into business logic in multiple applications. The quality of the data determines the utility of the data. Data quality is determined by the authority of the data source, its validity, correctness, timeliness and completeness of the data.

   Many applications can be integrated using a shared database if the quality of data permits. Shared databases decouple data from business logic, which provides the basis for widespread application integration. The decoupling from business logic and abstraction of data often leads to long-term limitations in application integration. It is not possible to design shared databases that are ideal for unknown (future) applications at the time of the design. This can lead to the expense associated with database redesign. Other issues with shared databases involve ownership of the data and distribution of the data.

3. **Remote Procedure Calls** (**RPC**) – Remote procedure calls are another form of integrating two or more distributed applications. Whereas a share database is a data-centric approach to combining data and business logic, remote procedure calls are a functional approach. When remote procedure calls are utilized, one application invokes business logic in another application while passing enough information to direct the execution of the business logic. This execution of business logic may in turn update data for the application implementing the remote procedure call.

   Many technologies support RPC, including COM, CORBA, and Java RMI. RPCs are an effective method of business logic reuse. However, the use of remote procedure calls often leads to the tight coupling of application logic, which produces brittle application architectures that are difficult to maintain and extend over the life of applications.

4. **Web Services** – Web Services is a more recent technology that has evolved from remote procedure call mechanisms. Web Services is similar to remote procedure calls because it provides a method for one application to invoke business logic in another application. Web Services is different from traditional remote procedure calls in that Web Services is viewed as the building block of a Service Oriented Architecture (SOA). In an SOA, Web Services has a coarse granularity and often the granularity aligns to business processes. The coarse granularity and loose coupling of Web Services provides a means of quick adaptation in the business environment.

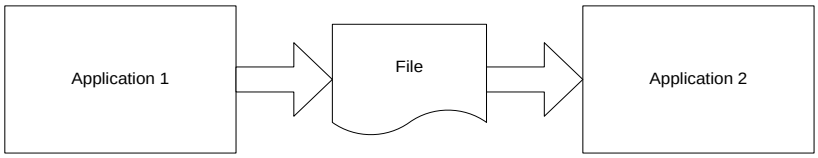
5. **Messaging** – Common messaging systems integrate messaging applications and change data back to invoke business logic. Message-oriented Middleware (MOM) differs from a file transfer because file transfer data is often batched. Messages in MOM are generally sent as the data becomes available to generate the message. MOM differs from remote procedure calls since the messages in a MOM solution are decoupled from specific application logic, whereas a remote procedure call invokes specific application logic, which closely ties two applications together.

Integration Patterns

Patterns are proven solutions to recurring problems. By recognizing a pattern in a problem, it is possible to identify a solution to an application integration problem. The following patterns are commonly found in application integration.

File Transfer Pattern

Problem: There are multiple applications built independently with different programming languages, running on different hardware and operating system platforms. The applications have a requirement for data that can be produced by one or more applications. The data is not needed in real-time and the data can be processed on a schedule.



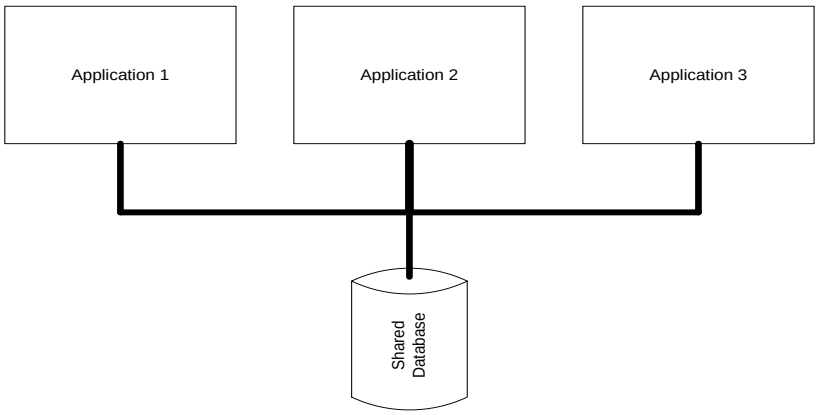
Solution: Applications can produce files and the files can be transferred on a regular schedule to other applications to consume the information in the files.

Consequences: An application rarely contains data in a format that is understandable by another application. Developers are to write transformation logic to create files that have the required data and are in a format to be understood by another application.

Best Practice: Applications are to be written to produce and consume XML files. XML files contain self-describing data and can be transformed into other formats through commonly available utilities and APIs.

Shared Database Pattern

Problem: There are multiple applications built independently with different programming languages that run on different hardware and operating system platforms. The applications have a requirement to rapidly share information in a consistent format. The data often has complex relationships and is continuously validated and enforced.

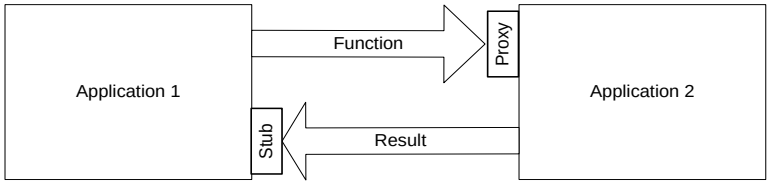


Solution: Use a shared database to integrate the applications.

Consequences: Existing applications may need to be re-factored to use the shared database if they were not already using a database. Additionally, the database design and security may need to be altered to support all applications. Simultaneous updates of the same data can be handled by transaction management tools of the database system.

Remote Procedure Call Pattern

Problem: There are multiple applications built independently with different programming languages that run on different hardware and operating system platforms. The applications have a requirement to share business logic in a responsive manner.



Solution: Applications can use remote procedure calls that provide an interface to allow other applications to execute the procedure call. Web Services is the modern remote procedure call mechanism that allows applications of different programming languages and execution platforms to communicate.

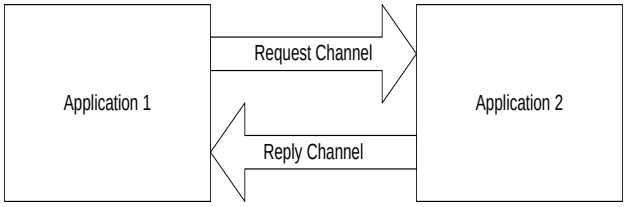
Consequences: Applications need to be designed to provide an appropriate remote procedure call interface. Designing the interface requires skill and knowledge. Many distributed applications performing remote procedure calls create architectural complexity and require re-factoring over time to manage the architectural complexity. Applications become encapsulating units of their data and are responsible for the integrity of their data. Complex logic may be required to maintain the integrity of data if an RDMS is not

used. If an RDMS is used, and the applications are primarily sharing data, a shared database may be a better approach.

Best Practice: Applications using the remote procedure call pattern are to implement Web Services.

Request-Reply Pattern

Problem: Two applications communicate data via messaging and require a bi-directional conversation.



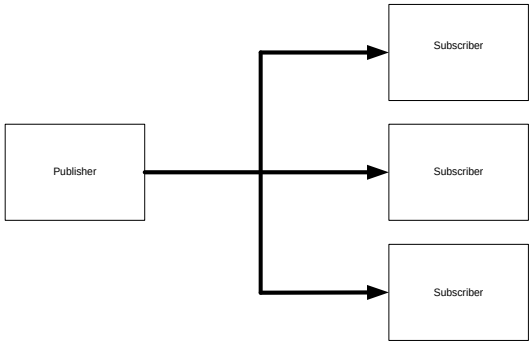
Solution: Use messaging to communicate between the two applications with one channel dedicated to the request and another channel dedicated to the reply.

Consequences: The Request-Reply pattern works with two applications.

Best Practice: Dedicate message channels to messaging in one direction.

Publish/Subscribe Pattern

Problem: Multiple applications receive messages sent from a single application, triggered by an event.

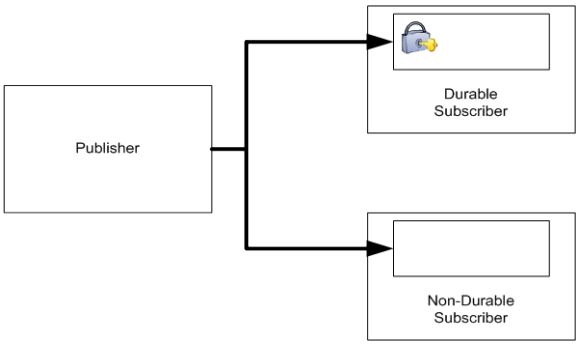


Solution: A publisher application sends the message on a publish-subscribe channel. The publish-subscribe channel delivers a copy of the message to each subscriber. In this pattern there is one publish channel and multiple output channels; each output channel has exactly one subscriber.

Consequences: Subscribers consume a copy of the message from their own dedicated channel. For a message to disappear, all subscribers are to consume their copy of the message. Depending on the type of message being published, not all subscribers may be interested in every message. Subscribers are to use logic to consume messages and determine if it is a message that the subscriber desires. This scenario increases application overhead while delegating processing to distributed subscribers.

Durable Subscriber Pattern

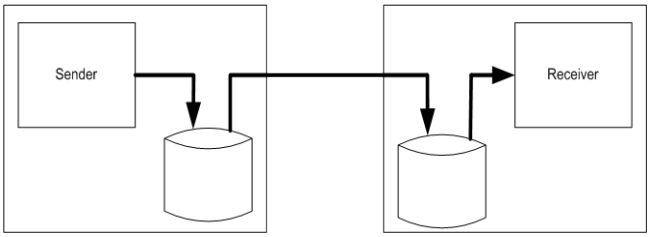
Problem: Multiple applications are interested in receiving messages from a single application when an event occurs. However, some or all of the applications may be disconnected when an event message is written and do not want to miss the message.



Solution: The messaging system can use a durable subscriber to save messages during disconnection of the subscriber.

Consequences: When a durable subscription is used, the message system saves the message for a disconnected subscriber. The subscriber will not loose any messages when in a disconnected state. When the subscriber is connected, the system behaves in the same manner as a normal publish-subscribe channel. The messaging system behaves differently only when a subscriber is disconnected. Subscribers that remain disconnected for long periods or do not return, may require operator intervention.

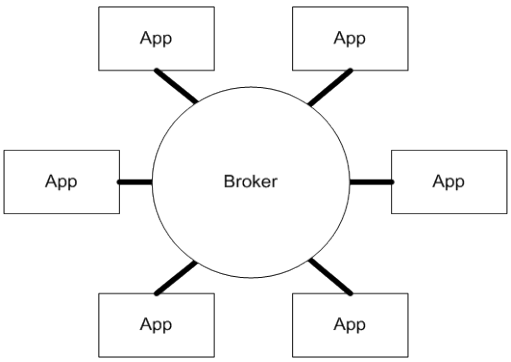
Guaranteed Delivery Pattern
Problem: A sender initiates a message to a receiver. The message is to be delivered even if the messaging system fails.



Solution: The Guaranteed Delivery pattern makes the message persistent. The messaging system uses a built-in data store to save the messages. If the messaging system fails and is later resurrected, then the messaging system checks the message store and completes the transaction.

Consequences: The guaranteed delivery pattern requires additional overhead of writing each message to a data store.

Message Broker Pattern
Problem: Decoupling destination of a message from the sender and maintaining control over the flow of messages.



Solution: Use a central message broker that can receive messages from multiple destinations, determine the correct destination and route the message to the correct channel. The message broker pattern is also known as a hub-and-spoke architecture.

Consequences: The message broker pattern is a large-scale architectural pattern. Other patterns (e.g., publish-subscribe, request-reply) can be layered into the message broker pattern. The message broker pattern creates a central point of administration, which can be an advantage; however, there can be a great amount of architectural complexity as many applications are added to the message broker.

Choosing an Integration Approach

Various criteria need to be considered when deciding on the use of any type of technology. In addition, there are broad ranges of considerations beginning with the need for the application integration. Is an application integration solution truly required? Developing a stand-alone application that does not require the services or interaction of other applications will avoid the need for application integration. Conversely, a solution that generates redundant data or business logic and creates an application silo that is disconnected from the enterprise may not be desired.

The first step when choosing an application integration solution is determining whether the data or business logic exists in a usable form elsewhere. If data or business logic does exist, then what is the method to leverage the data or business logic in the new application? The following table summarizes characteristics of integration approaches followed with detailed descriptions of the characteristics.

	File Transfer	**Shared Database**	**Remote Procedure Calls**	**Web Services**	**Message Oriented Middleware**
Cost	Varies	Varies	Varies	Varies	Varies
Simplicity	Simple	Varies	Varies	Varies	Varies
Coupling	Loose	Loose	Tight	Varies	Varies
Data Centric	Yes	Yes	No	Yes/No	Yes
Synchronicity	Asynch	N/A	Asynch/Synch	Asynch/Synch	Asynch/Synch
Data Latency	High	Low	Low	Low	Low

1. **Cost** – Costs for integrating applications vary. For instance, two Java applications can integrate using RMI (a remote procedure call mechanism) without additional hardware or software costs. Managing the programmatic integration and maintaining the two applications over time may increase the total cost of integration. Other solutions that incorporate file transfer and shared databases may leverage existing resources without significant cost. In addition to providing a lower-cost solution than file transfer or a remote procedure call solution, message-oriented middleware may already exist in an organization, or could be purchased to manage messages between many applications.

   Any potential integration technology selection is to include a cost analysis of the various integration methods, including: development, hardware, software, administrative, operational, and long-term application maintenance and extension costs.

2. **Simplicity** – Simplicity is a critical architectural construct from a design perspective as well as from an implementation and maintenance perspective. The selection of an application integration method affects the architecture of a system of applications because more than one application needs to be coupled in an application integration solution.

   An integration technology choice is to minimize the amount of program code required to implement the integration solution. Architectural simplicity is also important with respect to the long-term costs associated with maintaining and extending application in an integration solution. One-off solutions may seem architecturally simple; however, in the long term, when there are many one-off solutions, maintenance and operations cost may increase. For instance, a file transfer solution between two applications may seem like a very simple solution, but if the solution grows to tens or hundreds of files over time there is a high likelihood that the solution will become more expensive to manage and maintain than using another solution.

3. **Coupling** – Application coupling is the degree to which an application needs to know how another application works to make integration possible. Tightly coupled applications are difficult to maintain and extend in the long term. Integrated applications are to be loosely coupled when possible, allowing applications to share data and business logic while allowing applications to change over time. File sharing, shared databases, and messaging allow for loose coupling. Remote procedure call mechanisms and Web Services can provide for loose coupling with appropriate infrastructure and architectural designs.

4. **Data-Centric Logic** – One aim of integration solutions is to achieve the reuse of data and/or business logic. It is possible to meet an application's needs by providing data to an application so it can execute business logic, or by providing the business logic from a component, service or another application which uses the data of the application. In short, data and/or business logic creation costs money. If data and/or business logic already exists then a significant costs savings can occur by reusing the data and/or business logic.

   Data and business logic forms can determine the application integration solution. For instance, if data exists in a shared database the integration solution is to utilize the shared database. Likewise, a Web Service may exist to meet the business logic needs of an application. The availability of data or business logic will affect the selection of an integration solution.

5. **Synchronicity** – Application requirements determine the synchronicity needs in an integration solution. File transfer and shared database solutions can be used when applications do not require acknowledgement of data receipt. Remote procedure calls and messaging integration solutions can be implemented in a synchronous or asynchronous manner.

6. **Data format, latency, and quality** – When investigating the reuse of data in an integration solution the format of the data needs to be agreed-upon between applications for applications to integrate. XML is an ideal representation of data for application integration and the use of XML is highly encouraged in all integration solutions. However, even if data can be exchanged between applications the timeliness and quality of data may affect the integration solution. For instance, a file transfer may batch many transactions to be used between two applications, but if application requires real-time data, the file transfer integration solution may not be adequate.

Operational Best Practices

Operational best practices ensure the successful implementation of application integration projects. The best practices presented here take into account the total lifecycle of integration solutions.

File Transfer

1. Application integration solutions that utilize file transfers are to format the contents of files using XML.
2. When secure file transfers are needed, the file transfer solution is to be evaluated for the inclusion of:
   a. Secure file transfer mechanism, e.g., authenticated FTP or the use of FTPS; and
   b. Encryption of file transfer payloads.
3. File transfer solutions are to include an operation plan to monitor and administer file transfers.

Shared Databases

Refer to ITP-INF001, *Database Management Systems* for best practices in implementing database applications and integration solutions using shared databases.

Remote Procedure Calls

Web Services are to be utilized for remote procedure call integration solutions.

Message-Oriented Middleware

1. Develop a messaging architecture that aligns to application requirements prior to implementing a message-oriented middleware solution. The messaging architecture is to address the entire lifecycle of the solution, including additions of new and existing applications to the middleware solution, the distribution of messaging end points, and bridging multiple middleware products.
2. Develop operational procedures for monitoring and administrating new messaging applications.

3. Use consistent naming conventions for channels and queues.
4. Use XML for message payload. This is particularly important for the future extension, maintenance and integration of applications using transformation and orchestration tools.
5. Ensure unidirectional message channels and queues.
6. Design and configure channels and queues to align with non-functional requirements, including:
    a. Application Volume
    b. Queue Depth
    c. Security Policy
    d. Performance
    e. Expected Data Volumes (messages per second)
    f. Message Length
7. Implement and maintain Dead Letter Queues for undeliverable messages.