Enterprise Connectivity Patterns: Implementing integration solutions with IBM's Enterprise Service Bus products

Helen M Wylie
Peter Lambros

March 10, 2009

This article describes and defines a set of connectivity patterns that encapsulate some of the more common solutions in the application connectivity space. Many of these patterns build on and refine the more general architectural pattern known as the enterprise service bus (ESB). In defining a classification scheme for these patterns and discussing the various factors that influence their selection and implementation, this article and the accompanying developerWorks wiki can help lead to the appropriate choice of solution for your particular connectivity requirements.

Introduction

The value of the enterprise service bus is well understood in the enterprise integration practice and yet the very flexibility offered by such a bus can in itself lead to a bewildering selection of choices when planning and building an integration architecture (see Related topics). This can be further compounded by the range of capabilities provided by the middleware products which support such architectural patterns. Moreover these products typically support not just the very specific notion of a service bus as an infrastructure for connecting and virtualizing services in the context of a broader service oriented architecture but the more general notion of the service bus as a connectivity infrastructure for services in a very generic sense. In this article we will use the term service bus (or enterprise service bus) in the much broader sense.

Capturing the experience gained through many real-life implementations and abstracting from this a core set of patterns provides a framework within which to make the appropriate architectural and implementation decisions, facilitate product selection and usage and benefit from a set of recommended and proven approaches to the integration of enterprise applications and services. In selecting and describing such patterns there are a number of different facets to be considered:

- the overall integration solution architecture pattern
- the interaction style(s) enforced between the endpoints
- the operational aspects required
• the security model to be applied

Taken together these define specific connectivity mediation patterns.

**Figure 1. Facets of the connectivity patterns**

This article introduces a collection of such patterns which are frequently found in solutions implemented using the IBM WebSphere portfolio of Enterprise Service Bus products: WebSphere Message Broker, WebSphere ESB and WebSphere DataPower Integration Appliance XI50. The detailed specification of these patterns and corresponding implementation notes and examples represent a work in progress which can be found on the associated developerWorks Enterprise Connectivity Patterns wiki.

**Connectivity solution architecture patterns**

From an overall integration architecture perspective we are able to identify six common categories of patterns for connectivity solution

• Service virtualization
• Service enablement
• Gateway
• Message-based integration
• File processing
• Event-driven integration

The subsequent sections describe each of these in more detail.

**Service virtualization**

The service virtualization patterns describe how we might provide true loose coupling between services by providing additional levels of indirection through an Enterprise Service Bus. Viewed in another way, these patterns also address the requirements of mediation (routing, protocol conversion, data transformation, logging, etc) between services when addressing connectivity needs in a service oriented architecture.

Service virtualization patterns encapsulate the narrower definition of a service bus and are typically used to address a number of distinct requirements, including:
• loose coupling - presentation of an existing service as a virtual service on the ESB
• provision of a provider facade to include transformation, enrichment etc.
• provision of a provider facade to provide a standard interface for non-standard services
• provision of standard service management utilities (security, logging, monitoring, charging etc.)

Figure 2. Service virtualization patterns

Some examples of this pattern follow.

• Simple service proxy
  The use of service proxy is one of the fundamental patterns of an ESB and provides the loose coupling which is a requirement of most service oriented architectures. This pattern takes an existing service and deploys a new virtual service on the ESB enabling it to be accessed through a controlled point of access and possibly with a different protocol to that of the original provider. This introduces a point of mediation and in this, the simplest of ESB patterns, the mediations would often be limited to various types of logging and security configuration. However the introduction of the ESB as a mediation point would also enable the introduction of service management functionality, for example error handling and alerting, traffic management, service charging, and performance measurement. Many of the other patterns are embellishments of this basic pattern.
  This pattern supports the governance of services in an SOA, it is often used in conjunction with requirements that all business level or enterprise level services are only accessed through the bus and that standard management functions are mandated for all such services. This pattern also provides a means for securing access and concealing the internal structure of an organisation’s IT infrastructure and thus provides a starting point for offering access to services from outside an enterprise’s own security zones.

• Service selector (Router)
  The service selector pattern extends the simple service proxy pattern and introduces the concept of routing service requests. This can be a simple end point lookup to maintain loose coupling but may also be used to route based on data within the message, on the timing of the request, or on the status of alternative provider services.
  This pattern could be used where multiple instances of a service provider are hosted in different geographical centres and the appropriate service is selected based on account
details. It could also be allowed to maintain 24*7 service across a global enterprise where some windows of downtime are required or to provide priority routing based on customer priority or on high value instances of a service request.

- **Service translator**
  The service translator pattern also extends the simple service proxy pattern. It may not always be desirable to have the interface of an end provider as the published interface of an enterprise service on the bus. The service translator pattern introduces transform(s) which allow the interface(s) presented by the ESB to differ from that of an end provider.
  Standardisation of data across an enterprise is often a goal but in most cases it is not an achievable ideal. However the definition of such a format can simplify new developments and where standard data models are defined this pattern may be implemented by provision of a service facade to the provider service which transforms from the enterprise standard to the provider specific.

- **Service normaliser**
  This pattern extends both the translator and router patterns and supports the concept of alternate service providers who offer semantically equivalent services with differing interfaces. Routing is determined in the same way as for the service selector pattern and transformation is applied to a service request after routing has determined the target service and interface.
  This pattern enables a single interface to be provided for a virtual service on the ESB, with the ESB determining which provider is most suitable for each request and applying the necessary transformation before invoking the provider service.

- **Version selector**
  This pattern is a specialisation of the service translator pattern and is included in its own right as it has a very specific rationale which may apply across a range of other patterns. While the translator pattern is intended to route requests the version selector is used to ensure that client requests are routed to the appropriate service version.

**Service enablement**

This category of pattern addresses the problem of encapsulating functionality, which in itself does not present a service interface, and presenting this functionality through a service oriented interface. This represents the transition from more traditional notions of enterprise application integration into service oriented architectures, allowing existing assets to be reused in the new style without requiring radical change. It enables the implementation of an ESB that delivers access to "services" derived from functionality implemented using diverse technologies across the enterprise and to present a single, consistent view of these services to the service consumers.
Some key examples include:

- **Simple service facade**
  In its simplest form this pattern is close to that of a service proxy but in this case the service requests are not fulfilled by a service provider but by existing applications which are accessed by messages, adapters or other APIs. Adapters may be located with the application, or more usually they are implemented as ESB components. Routing, transformation and normalisation may also extend this pattern in the same way as the service virtualisation was extended.

- **Complex service facade**
  A further set of extensions may incorporate integration logic from some of the richer features of ESBs to build service capability, potentially using multiple sources of functionality, which is then presented as a virtual service on the ESB. This pattern is often used to build higher level business or enterprise services from lower level IT services, components or fine-grained application function.
  Integration logic can be built in many ways but some typical examples would be a sequence of calls, multiple requests with aggregation of responses or information distribution.

- **Client access**
  This set of patterns provides entry points for clients which are not enabled for service-oriented client access. This is most likely to be through adapters and enables existing package applications to participate in the introduction of an enterprise architecture which incorporates an SOA approach.

**Gateway**

A *gateway* is a part of a message bus or service bus which provides boundary functions which apply to all incoming messages and are not format dependent.

Boundary functions typically utilise data from standard headers (at transport, SOAP or even data level) to determine what action to take but do not need to understand the complete format of the
message data (or body). A gateway pattern may then call a service directly or invoke a further pattern. These boundary functions include:

- request routing
- authentication
- authorisation
- audit and logging
- protocol conversion
- response correlation

**Figure 4. Gateway patterns**

Examples of gateway patterns include:

- **Security gateway**
  This rather specific example of a gateway pattern offloads all non-standard security processing from the main infrastructure and will execute authentication, authorisation and possibly audit before calling the true destination. In this mode there is almost always a trusted link between the gateway and other parts of the ESB so that the security model required for those other parts of the ESB can be simplified. Security gateways may reside in a different security zone to the rest of the ESB and would provide connectivity to external clients. This pattern would support a wide range of incoming security protocols while simplifying security in other ESB components.

- **Service connector**
  The gateway pattern can provide the simplest possible way of connecting a number of existing services into an ESB when introduced as part of an enterprise architecture based on service oriented principles. The gateway introduces a point of control in the enterprise architecture and can bring a range of ad hoc services under the control of a service registry and associated governance but without the need to develop individual mediation flows for each service. Messages enter the gateway and are routed onwards to an end provider, to a provider facade or to a further mediation flow.

- **Boundary mediator**
  The boundary mediator pattern extends the service connection pattern by enabling standard mediations to be applied to all incoming (and/or outgoing) service requests or messages.
These standard (content independent) mediations are developed once in the gateway and can be applied to all incoming messages. These mediations may include any or all of validation, logging, audit, authentication and authorization. They may be applied universally or selectively based on the gateway properties, lookup based on request data or on data within the incoming request (normally in headers).

**Message-based integration**

An ESB may extend an existing messaging infrastructure by providing an environment for building and deploying 'infrastructure level' message-based 'applications'. Examples of such "applications" would include routing and transformation services, logging services etc. This may extend a single underlying messaging infrastructure or provide a bridge between different products and technologies. Adapters may be used to provide access to applications or services which do not have an in-built messaging capability.

**Figure 5. Message-based integration patterns**

Functionally the message-based patterns in some regards correspond to the service virtualization patterns. However, there is a key difference in how we might think about them. In the service oriented model we are typically focused on the provision of services that can be consumed by many and arbitrary 'clients'; that is, we take a provider-to-consumer view of the world. In a message oriented model we are typically much more focused around the data that is flowing through the system and the set of actions that are to be applied to this in-flight data; that is, we have more of a producer-to-consumer view. Whilst not an absolute distinction this thinking may influence the context in which we seek to apply each pattern.

Some specific examples follow:

- **Message router**
  The message router pattern can be used to provide a strong level of decoupling between applications or services which need to exchange data by enabling data sent from one application to be routed to one of several potential target applications based on various conditions.
  
  *Context-based routers* select an appropriate target based on the identity of the sender or on some aspect of the data carried in a protocol header. *Content-based routers* make a selection based on the content of the message payload. *Load-based routers* use information about the application load on the various target systems.

- **Message translator**
  The message translator pattern allows data from one application to be mapped into the data format required by another application without either application being aware that such
mapping is needed or taking place. This pattern covers anything from direct mapping to highly complex transformations possibly involving lookups and cross references.

- **Messaging bridge**
  The *messaging bridge* pattern maps data from one transport mechanism to another without modifying the format or content of the message payload. An implementation of this pattern must also handle the mapping between the different addressing schemes which might be used by the separate messaging systems.
  A frequent example of the use of this pattern is the bridging between JMS implementations from different vendors.

- **Message aggregator**
  The *message aggregator* pattern addresses the need to take multiple messages from one or more applications and merge them into a single piece of data to be propagated as a new message. The inbound messages may come from independent applications or may be asynchronous response messages from a set of applications which received requests from an implementation of the message splitter pattern (described below).
  An implementation of this pattern may simply concatenate the individual source messages or may incorporate some more sophisticated set of data mapping capabilities. It must also be able to handle the failure of an expected inbound message to arrive within a prescribed time period and the subsequent late arrival of this message.

- **Message splitter**
  The *message splitter* pattern extracts subsets of a message which are then sent as separate messages to multiple target applications. A set of mapping rules define how the original inbound message is to be broken apart into constituent pieces.

- **Message request/response correlator**
  A typical message broker system will often be processing multiple requests from the same stream or queue in a parallel, asynchronous manner. When such requests are being routed to a further system (or systems) there will be a requirement to correlate any response messages to the original request. The message request/response correlator pattern provides a solution to this particular problem.

### File processing

An ESB may provide a managed execution environment for the processing of files (either locally or using an FTP protocol).
Typically this would involve activities such as the transformation/translation of data held in the files, the 'shredding' of files into multiple individual transaction records, the routing of records, the accumulation of records into target files and/or the routing of files (in their entirety or by individual record) to specified locations/processors. Some examples of this patterns and its use are:

- **File Record Processor**
  This pattern processes each record in an input file and accumulates the resulting processed records in one or more "transitory" target files which will be moved to target directories on completion. This pattern supports the need to provide processing of batch files containing end of interval transactions. This might be the transfer of accumulated transactions for a day where the source and target systems cannot support transactional interfaces, or might be the transfer of transactions which are expected to occur at a particular time.

- **Record Batcher**
  This pattern receives data in service requests, messages or through adapters and after processing generates records to transitory files which are moved to target on completion. It supports interactions between newer systems and legacy systems which support only batch interfaces. This might be the first step in a move towards a more service oriented architecture or perhaps the integration of a batch system acquired by acquisition into a more transactional environment. This pattern also supports the need to accumulate data for overnight runs which might cover invoicing, billing, issue of contracts and related documents and renewal reminders.

- **File record Distributor**
  This pattern breaks a file down into records and uses each record to make a service call, or pass a message to one or more systems with a transactional interface. A typical use of this pattern would be the processing EDI files which contain a number of transactions of different types to be routed and processed individually. This approach may also be taken to update systems with reference data especially where timing of such updates is not critical as in the case of updates with a given start date.

**Event-driven integration**

The patterns in this paper are concerned with connectivity across an ESB and we are not attempting to establish a full set of patterns for 'event driven architecture'. However, although there is as yet no clearly established definition of 'event driven architecture', a term which can cover a
range of different application scenarios, there are 'EDA' scenarios in which an ESB will play a key role. These include integration with complex event processing engines including the ability to filter information or event streams, distribution of events in 'real time' (by which we mean the speed required by the application), and processing of events from physical devices (detectors, sensors).

**Figure 7. Event-driven integration patterns**

Some examples of this patterns and its use are:

- **Event Distributor**
  This pattern provides for distribution of events (with appropriate classification by "topic" and associated data) to multiple interested parties who register their need to be informed about subsets of published events based on a tag ("the topic") or data associated with each event. Data is published to a publication point on the ESB and distributed from that publication point to all subscribers who have registered for that information.

  Events published to a publication point may represent change of status, with the latest status available to all subscribers whether new or requesting a refresh. Alternatively events may represent a stream of data with each "event" being distributed and then deleted.

- **Event Filter**
  This pattern provides for processing of events (interactions) entering the ESB to filter and transform them into a stream of significant events which are passed to an event engine. This may be a event processing engine for detection of complex events or may be to a system which monitors business events, for example a dashboard application. This pattern supports a one-way communication and is acting purely as a pre-processor to the event engine.

  A particular specialisation of this pattern would be the provision of entry points for data from a variety of sensors, actuators and adapters.

- **Event Extractor**
  This pattern monitors interactions across the ESB and processes them to extract and pass relevant events to the event processing engine. This pattern supports a one-way communication and is acting purely as a pre-processor to the event engine.

- **Event Reactor**
This pattern is like the preceding pattern in that it acts as a pre-processor for an event engine. However this pattern supports a synchronous interaction with the event engine and may receive a response which indicates that the latest event has triggered a complex event. The ESB flow may then react to that event by, for example, raising an alert to targets which need to act when such an event occurs.

**Operational aspects**

The *aspect-oriented connectivity* features of the bus include the interaction styles and security models described in subsequent sections together with various standard service management functions or utilities which will typically include such things as:

- validation
- logging
- audit
- error handling
- SLA / performance measures

These aspects of mediation are abstract and are combined with specific integration patterns according to the needs of the overall solution architecture.

**Interaction styles**

Contractual behaviour patterns define interaction styles and qualities of service. We currently identify eight key patterns of interaction:

- One way assured delivery
- Synchronous read request
- Synchronous update with no transaction
- Synchronous update with global transaction
- Synchronous update request with acknowledgement and call-back
- Synchronous update request with acknowledgement; provider assures action
- Synchronous update request with acknowledgement; ESB handles errors
- One-way with asynchronous
- One-way with poll for completion

These patterns are abstract. They define the way the interactions behave in terms of synchronous/asynchronous interfaces, level of assurance of delivery, handling of time outs and late responses, and error handling. Such contracts of behaviour are not implemented directly but combined with specific integration patterns to create concrete patterns of behaviour.
The intent of this interaction pattern is to ensure that requests are reliably delivered and processed.

- All communications are assured.
- Each called component reports errors reliably.
- Messages are persisted across failure.
- ESB does not reject messages but attempts to process.
- Errors are captured with full message content.
- Error handling (outside ESB scope) fixes the problem.
- SLA should be measured as throughput (messages / requests per time period).

**Synchronous read request**

The intent of this pattern is to provide a simple request mechanism with no side effects on failure.

- Requesters wait for response but must time out.
- There is no guarantee of timely response from provider.
- The ESB wait is less than requester and ESB may provide explicit time out errors to requester.
- Messages are not persisted.
- SLA should be measured as response time i.e. time taken for response to arrive.

**Synchronous update with no transaction**

The intent of this pattern is to provide an update mechanism which does not guarantee delivery or processing and in which a requester must take responsibility for handling errors or time out conditions.

- Basic behaviour as for synchronous read request.
- Error handling by a requester on time out may include:
  - Repeat if operation is idempotent and error is timeout.
  - Reconciliation at end of period.
  - Invocation of manual or semi automated error handling.
- Error handling by ESB on timeout or error response on call to provider may include:
  - Repeat if operation is idempotent and error is timeout.
  - Inform client.
  - Place on error queue.
  - Reconciliation at end of period.
- SLA should be measured as response time i.e. time taken for response to arrive.

**Synchronous update with global transaction**
The intent of this pattern is to provide a mechanism which synchronizes updates across a service requester, ESB and service provider(s).

- All participants must support an agreed protocol for transaction coordination, e.g. WS-AtomicTransaction
- Failure to commit causes rollback of any update made by any component
- Resources will be locked while the transaction occurs
- All participants must be prepared to roll back if a transaction is not complete within a specified time period.
- SLA should be measured as response time i.e. time taken for transaction to complete.

**Synchronous update Request with acknowledgement and callback**

The intent of this pattern is to allow requesters using synchronous protocol to know that a request has been submitted and will be processed and to inform the requester when request has been processed.

- Requester supports only synchronous protocols / behaviour
- Requester makes update request and gets acknowledgement that it has been accepted (or rejected)
- ESB does not acknowledge until the message has been secured.
- ESB uses assured delivery to make update request to provider
- Provider service has agreed behaviour to call back (asynchronously) with results of update
- Requester may keep list of outstanding update requests
- Broker must ensure that requester has received callback (retry if timeout)
- Error handling can be assigned to any participant
- SLA should be measured as response time for acknowledgement and also for requests fulfilled over a time period.

**Synchronous update Request with Acknowledgement (ESB handles errors)**

The intent of this pattern is to allow requesters using synchronous protocol to know that a request has been submitted and will be processed. This pattern guarantees that errors will be handled and no call back is made to the requester.

- Requester supports only synchronous protocols / behaviour
- Requester makes update request and gets acknowledgement that it has been accepted (or rejected). If accepted action is assured by broker and provider.
Security models

In defining security models and patterns there are two facets that need to be considered:

- the set of participants
- the set of required security enforcement policies

The participants must together enforce security to ensure that the requirements for authentication, authorisation, audit, confidentiality, integrity and usability are met without degradation of performance to an unacceptable level.

Participants in SOA security models

Any service based interaction will include the following major participants:

- Requesting Application or Process
- Intermediary services: middleware functionality providing the integration capability between requesters and providers, often classed as a bus or ESB.
- Provider Application / service provider
It is assumed that basic security hardening of these components is in place and the trust models on which the patterns are based will then assign security capability related to the run time interactions to each of these participants.

There may be additional supporting participants which contribute the implementation of the patterns and may have an impact on the strength of the mechanism used.

- Initiating User and associated devices / software when distinct from the requesting application.
- External Security Providers for:
  - authentication
  - authorisation
  - audit
  - monitoring and alerts
  - identity / credential mapping, may be referred to as a secure token server in some contexts
  Although provisioning of user registries associated with these providers is clearly a part of a total security solution these considerations are excluded from the runtime patterns and adequate provisioning mechanisms are assumed.
- Firewalls and proxies
- Network components
- Service registry if security policies or profiles are held with a service definition in the registry

The list of participants above are not necessarily discrete components and the scenarios may be built using products, or components, which operate in one or more of these participant roles. For example an application may support both requester and provider role in different end to end interactions or external security services may support one or more of the functions listed.

When security functionality is distributed across the participants in SOA transaction there must be a trust between participants that each has fulfilled its own part in the overall security.

The distribution of these functions across the participants leads to different requirements for trust between participants and for capabilities of the different components. Each of the components may have options for the way in which they fulfil their security role and end to end paths must be determined to match up the capabilities across all capabilities.

**Security patterns**

The following key patterns of trust models are frequently encountered:

- **End-to-end trust**
  The ESB trusts the requesting application to authenticate and authorise end users and service providers regard the ESB as an authenticated and authorised user. End user identities are not used in security enforcement but may be included in data for audit. The security model requires only that each of the components can be identified in an interaction, which can be achieved at network architecture level or through use of generic application identities.
This scenario is frequently used when all components are internal, under the same security management and part of the trusted computer base. Integration between a enterprise CRM system, with well defined authentication and authorisation, and enterprise back office system often follow this model.

- **ESB enforces security on application identity**
  In this model the ESB no longer fully trusts requesting applications and will require credentials to validate the identity of the requester. The ESB will ascertain whether this user is authorised to make the request before proceeding. As the requesting application is not trusted audit requirements would be met by the ESB.
  Provider services still regard the ESB as a trusted user and all requests are made under the identity of the ESB or of a known trusted service on the ESB.
  The end to end security model would mean that all providers give access to all trusted requesting applications. Where there are requirements to partition the trust, for example to restrict access to certain financially sensitive services to requests only from trusted applications which allow access only to privileged users, this model assigns the responsibility for this partitioning to the ESB.

- **ESB enforces security on end-user identity**
  This scenario is very similar to the previous case in that the ESB applies authentication and authorisation but in this case the identity of the end user is required rather than that of the requesting application. In this model provider services trust the ESB but need the identity of the end user and therefore require the ESB to assert the identity of the end user. The link between the ESB and provider services may be secured by network architecture, by ESB credentials or by encryption using PKI.
  This model is used where provider services require a secure assertion of the end user identity in order to provide fine grained authorisation, based on business logic within the provider, or to ensure accountability by securely auditing actions in the provider against the initiating user.

- **Trusted authentication by requesting application**
  This model assumes, like the end to end trust, that a requesting application is known and trusted software. In this case however the application is trusted only to execute authentication and to assert the identity of the user to the ESB, which then authorises the access and asserts the end user identity to the provider. In this model the trust between the ESB and a provider service is the same as the previous case where the ESB executes the authentication. This model supports single sign on by users to trusted applications which are able to make service requests on their behalf. Identity assertion may be as simple as a username token but may involve the acquisition of tokens by the requesting application on behalf of a user and subsequent validation of such tokens by the ESB.

- **End-to-end authentication, confidentiality and Integrity required**
  This model assumes that there is only limited trust of the ESB by a provider service, which requires credentials which allow the provider to authenticate the initiator of the request directly. In this model confidentiality and integrity may also be required.
  This is not a model which assumes no trust in the ESB, which is almost certainly trusted to deliver requests to the appropriate provider in accordance with expected service level agreements and to act as a first filter on attempts to access such sensitive services. However the additional levels of end to end security, such as digital signatures, are used to identify the
initiating user in a way which enables non-repudiation while end to end encryption may be
used to prevent access by operational staff to highly sensitive information, which might occur
if exceptions occur in normal processing or where auditing / logging is executed in the ESB.

• **Publish / Subscribe**
  Although the publish / subscribe model still has the same major participants the security
model differs as there are two levels of security:
  • control over access to the functionality
  • control over access to topics

Control over access to functionality for publishers and subscribes is the same as the control
over any requests to the ESB. In theory this can use any of the mechanisms described for
general SOA interactions and these controls will result in a user name which can then be
used for the finer grained access controls.
Control over access to topics will normally be managed by the publication engine based on
access control lists on a topic hierarchy.

**Summary**

We have introduced and described a collection of patterns that are commonly found in enterprise
connectivity solutions built using IBM Enterprise Service Bus products. These, and other related
patterns, will be described in more detail, along with specific implementation notes and examples,
in the accompanying developerWorks wiki.

**Acknowledgements**

The ideas presented here have grown out of numerous discussions with many people but
particular thanks are due to Rob Phippen, Kim Clark, and Greg Flurry for their helpful feedback
and recommendations on the early drafts of this article.
Related topics

- Explore these topics in more detail at the ESB and Connectivity Patterns developerWorks wiki
- **Enterprise Integration Patterns** by Gregor Hohpe and Bobby Woolf (Addison Wesley ISBN 0321200683; October 2003) provides a catalog of patterns and real-world solutions that demonstrate the power of messaging and helps you design effective messaging solutions for your enterprise.
- IBM provides a variety of materials about the ESB design pattern. For more on this subject, see the following:
  - From the *IBM Systems Journal*, "The Enterprise Service Bus: Making service-oriented architecture real" (December 2005).
  - "Exploring the Enterprise Service Bus, Part 1: Discover how an ESB can help you meet the requirements for your SOA solution" (developerWorks, Apr 2007).
- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

© Copyright IBM Corporation 2009
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)