

Managing Evolution of Integration Middleware via Integration Architecture Design

Michael Gilfix

Industry Strategic Solutions
IBM Austin, mgilfix@us.ibm.com

Abstract

Integration technologies are evolving rapidly, making present design and investment decisions difficult. This paper will focus on a general systems integration architecture and methodology that will maximize component reuse and component lifetime as an enterprise infrastructure evolves. This flexibility is achieved by separating protocol features from the functionality offered by the integration component or application; component functionality is modeled as a web-service interface while a series of protocol mediators provide the glue and extensibility to integrate the component into the infrastructure. These mediators implement the desired characteristics for communication with the service, as well as any other appropriate communications logic. A mediator may provide the appearance of an event-driven model via polling, or implement transaction or security requirements on top of the web-service.

This methodology allows a single service to be reused in a variety of situations via the use of different mediators, such as: support for asynchronous pushing of data via an MQ event-driven mediator, transaction and security support via a JCA mediator, or a synchronous event-driven model via a traditional integration adapter mediator. In addition, mediators can be implemented as reusable meta-data driven components, as all interaction with the component or application APIs are performed via web-services.

Keywords

Business Integration, Integration Architecture, Integration Methodology, Web Services, Middleware.

Introduction

Integration middleware has become essential for managing a rapidly evolving IT infrastructure. Business process automation technology has helped streamline costs while ensuring that IT resources are meeting business needs (Georgakopoulos, D., Hornick, M., & Shet A., 1995). State-of-the-art middleware provides tooling for managing the life-cycle of process

automation: documenting an enterprise's business processes, modeling those requirements as a series of interactions between distributed enterprise components, and then driving those models into implementation.

Effective use of middleware requires a substantial investment and commitment by an enterprise. As a centralized component within the IT infrastructure, the middleware houses a significant portion of the logic (and knowledge) that ties disparate services together and makes them work as a cohesive whole. The artifacts created by process modeling tools, many of which are directly integrated with the middleware platform, become the primary source of documentation about how an enterprise does business. Moreover, the rich feature-set of current middleware platforms requires a substantial amount of staff training and implementation effort to maximize the effectiveness of the chosen middleware platform.

The difficulty in choosing which middleware platforms to invest in, as well as managing their evolution, is exacerbated by the many different kinds and capabilities of middleware (Emmerich, 2000) and the rapid evolution of middleware technologies. Architects must balance the requirements of legacy systems with the evolution of an enterprise's IT infrastructure. The result is often a mix of different middleware components is needed to solve a single integration problem. IT organizations are also wary of over-committing to a single piece of middleware, ensuring they can upgrade in the future to next generation technology and avoid product lock-in. Mergers and acquisitions have also made managing these systems increasingly difficult; IT organizations often need to consolidate two infrastructures, with what is usually a diverse set of technology, in a relatively short period of time.

To help mitigate the challenge of managing the evolution of this IT infrastructure, an architecture and methodology is presented that promotes service-oriented architecture and facilitates component reuse by isolating a service's interaction characteristics from its interface. Web service technology, such as WSDL and WSIF, are used as a language neutral means of

describing and accessing a distributed component's interface. The concept of the mediator is then introduced as a mechanism that allows a singular component to be integrated in a variety of scenarios, without change to the underlying component implementation; the mediator provides a secondary interface that abstracts the complexities of interacting with the service and presents a new interface that models the characteristics of the protocol used for integration of the component, as well as the behaviors that would be expected if that component was natively versed in that protocol. The result is an architecture that eases the transition between different kinds of business process automation technologies and simplifies any needed ad-hoc integration.

Background

Hub-and-Spoke Integration

A primary driver in the adoption of the hub-and-spoke model of integration is that establishing connections to and from enterprise applications and services remains a complex and arduous task. Integration brokers offer a flexible, more nimble integration environment: once a service is connected to a broker, new routing and transformation logic can then be defined within the broker environment to establish a new link with any other service attached to the broker. Since the integration environment is controlled by the broker's tooling, defining new service links becomes considerably simpler than establishing new service connections. Knowledge and skills in the particulars of establishing a connection to an application or service also remain isolated to the inner workings of a single end-point.

This approach still has drawbacks: the technology and APIs used to connect the integration middleware and the service varies greatly between middleware products. Many vendors offer toolkits for developing adapters that translate between broker requests to lower level service requests. A considerable amount of effort must then be expended to ensure that all of a service's APIs have been correctly exposed to the broker, that adapter code has been tuned to meet performance requirements, etc.

Unfortunately, as integration broker technology evolves, so do the toolkits and APIs that tie brokers to applications and services. This becomes particularly problematic when attempting to upgrade, switch, or add integration broker products. Although most vendors will expend a significant effort to solve backwards compatibility issues, porting these APIs to new broker architectures will often mean that existing

code will receive limited benefits from the new capabilities of the latest broker technology, and may eventually become a maintenance liability.

This tight coupling can be somewhat mitigated by using a modular design and isolating broker-dependent code. However, this approach is still not ideal; a significant amount of effort must still be expended to tie non-broker dependent code into the new infrastructure when porting a service. In addition, the broker dependent code may still contain a significant portion of critical integration logic, resulting in duplicated effort when porting the service. Instead, another mechanism is needed for expressing the service API in an implementation neutral manner.

Web-Services: Componentization and Service Composition

Web-services have increasingly gained acceptance among enterprises as an integration technology with broad applications (Papazoglou, M. & Yang, J., 2002; Yang, Jian, 2003; Mohan, C., 2002). While the term web-services encompasses a wide range of standards and technologies, some of the more interesting ones with respect to the design of enterprise integration services and business process automation are: WSDL, WSIF, and BPEL. The Web-Services Definition Language, or WSDL, provides a language-neutral description of the functionality offered by a service, as well as the data types needed by those operations. The Web-Services Invocation Framework, or WSIF, provides the API for calling a remote service, and isolates the calling code from the details of web-service invocation (Fremantle, Paul, 2002). Finally, the Business Process Execution Language, or BPEL, provides a language for specifying processes and interactive protocols, which may range anywhere from automation of a business process to composition of multiple, finer grained web-service function.

The advantage of using WSIF in conjunction with a WSDL-defined service is that the client and service can be deployed in a wide number of environments without the need to alter any of the service invocation code. For example, HTTP communications can be replaced with JMS to better match the needs of an intra-network execution environment by merely changing the web-service bindings associated with the WSDL definition. Communication need not even go over the network: Java bindings could be used to make a service invocation execute within the same Java process. The property of location transparency is extremely useful from an integration stand-point: integration architects can deploy the same service in multiple scenarios without requiring implementation changes.

BPEL is another upcoming web-service technology that is gaining increasing popularity among vendors. Many integration products currently support the importing and exporting of business processes as BPEL definitions, and these products may use BPEL exclusively in the future. This shift has implications for how integration architects should design business processes within their middleware of choice in order to ease transferring those processes to new middleware in the future.

Architecting with Evolving Middleware

Guidelines for Re-use

The key to longevity for an integration architecture is to maximize the efficacy and flexibility of the “glue” that holds the IT infrastructure together. In an ideal world, as an infrastructure evolves, the architect should be able to reassemble and reconfigure piece parts as if rearranging puzzle pieces. Meeting this ideal requires an effective componentization strategy and a good understanding of the various enterprise integration protocols and technologies. The architecture presented here uses the following architectural guidelines:

- Model functionality as a service wherever possible
- A protocol is more than a wire format: it’s a set of behaviors, constraints, and guarantees
- Push function to the end-points whenever possible (and model as a service)

The first guideline is based on the crux of software engineering: smaller components are vital for managing larger systems. Componentization is particularly important in the integration space; the interfaces defined at the component boundaries become the input to the various middleware tools that help streamline process automation. If the interface to the middleware or the communications technology is expected to change, these component interfaces will most likely be the point where the infrastructure change is managed.

The second guideline addresses the need to consider the nature of how data flows through out the integration infrastructure. Request-response protocols have markedly different characteristics from message-driven protocols and reflect the needs of the data flowing through the protocol. The difference in these characteristics is even reflected in the different flavors of integration middleware and their end point interfaces—for example, WebSphere Application Server Process Choreographer versus WebSphere Event Broker. To effectively manage evolving middleware, the IT

architect needs a methodology that spans differing paradigms.

Finally, a good rule of thumb is to try to model as much functionality as possible as an end-point. Functionality that has been implemented within a particular piece of middleware usually requires significant, if not complete re-implementation. By pushing that function to an end-point, it can be adapted to a changing environment just as any other service would. This guideline does *not* suggest avoiding implementing functionality within integration middleware environments completely! Instead, it is meant to encourage carefully calculated decisions about the trade-offs between internalizing an implementation within a particular middleware environment versus externalizing that implementation and making it available as a distributed service.

Introducing Mediators

Exposing services via web-service technology alone is insufficient to fulfill the needs of integration “glue”. This is due to the request-response nature of web-services. There are times when a message-oriented or event-driven system will need to communicate directly with a service. This system may be actual integration middleware or ad-hoc point-to-point integration with a distributed service. At this point, an adapter-like mechanism is needed to act as a bridge between the different communications methodologies. This mechanism is aptly named a mediator because it mediates between the web-services protocol and the end destination protocol. Figure 1 demonstrates an architecture for service mediation between a distributed component and various integration technologies and protocols.

First some set of function is modeled as a service. This function may be part of an interface to an industry application or a modular, reusable piece of code being exposed to the rest of the IT infrastructure. A WSDL service definition is then created and published to the world. The publishing process used depends on how that service is to be made available. If the service is to be accessed over the network, the service definition may be published within an application server and the server code made to execute within the application server environment. Alternatively, the service definition could be published on the local file system and the code linked in and executed at runtime.

Services may comprise any level or size of functionality, from the functionality of a full-fledged enterprise application to a snippet of code that performs a common function for the network. The amount of effort needed to turn legacy code into a

service depends greatly on how that code is structured and the nature of interactions with the service. If the code is written in Java, significant portion of the effort can be automated by tooling (generation of the WSDL definition, generation of EJBs for calling Java Code, etc.) and the code can be served inside a J2EE

container. C or COBOL code usually requires significantly more effort. However, this activity is a replacement for a portion of traditional adapter writing, since originally this function would have been integrated with the adapter anyway in order to expose the function to the integration middleware.

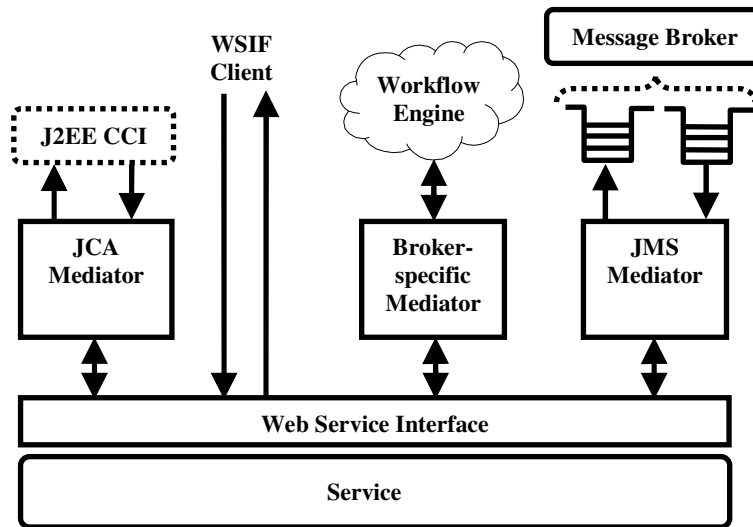


Figure 1 – Service Mediators for Several Protocols

The mediator then acts as a bridge between the web-service and the destination end-point’s protocol. Since the mediator has an intimate understanding of both protocols, it provides transformation services that map requests and responses from the destination protocol into their web-service equivalents. The mediator also acts in a manner that the end destination would expect from an entity that is natively versed in that communications paradigm. These mannerisms correspond to characteristics of the communications protocol. If mediating with a message broker, all incoming and outgoing communications through the mediator appear asynchronously message-driven. To simulate the message-driven paradigm, the mediator performs polling on the web-service, fetching events as soon as they become available and publishing those events to the message queues. The mediator should also guarantee other properties of the protocol, for example, persistence between the broker and the web-service. Alternatively, if the web-service is being exposed via a JCA connector, the mediator will interact with the web-service in a manner that provides transactional guarantees to the J2EE container in accordance with the JCA specification, either by working in conjunction with a third-party transaction manager or via functionality integrated into the web-service interface.

The resulting architecture offers numerous possibilities for component integration. If the service resides in an application server and is not constrained to a singleton instance, simultaneous interaction with the service using differing protocols, via multiple mediators, can be achieved with little or no effort. The service can also be made to execute in-process by dynamically linking the service code with the mediator and using runtime web-service bindings. The choice of which protocol to use for web-service invocation gives the IT architect another parameter to play with during deployment: HTTP might be preferable when communicating through a firewall, else JMS might more preferable for intra-network communications.

Separation of the service interface from the target protocol via the mediator allows the IT architect to select the communications paradigm that best matches legacy requirements or simplifies the integration process (Coqueret, Regis & Fiammente, Marc, 2003). The mediator also acts as the coupling point between the integration middleware and the service, isolating middleware dependencies and replacing the traditional adapter. Since many integration platforms provide a web-services adapter, this adapter could be used in place of developing a custom mediator.

Designing Services

While the majority of legacy application functionality and other distributed software components fall readily into the request-response paradigm of web-services, designing a service interface to allow for asynchronous event notification from industry applications or active processes requires some additional thought. The standard web-services assumption is that web-services technology is ubiquitous and that all software components are web-service enabled –i.e., that the application being monitored will always provide all its notifications to the monitor via web-services requests. Within the EAI world, this is rarely the case. Legacy applications provide notifications using a wide variety of mechanisms, including database tables, flat files, and queues. Some applications might even require special code to register itself with the application's internal communications bus in order to receive certain notifications. In these situations, a bit more work is required to adhere to the services paradigm.

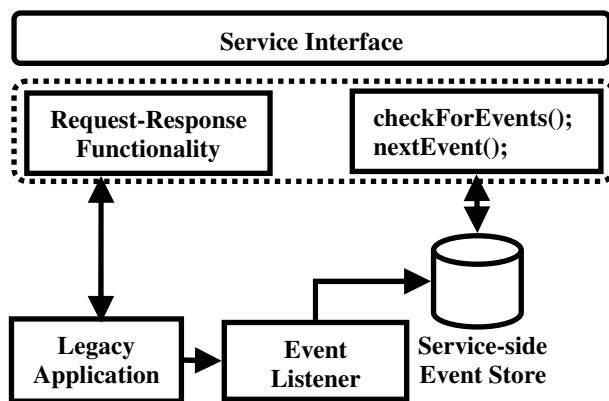


Figure 2 – Adapting Asynchronous Event Notification to the Web-Service Paradigm

An architectural solution to this problem is shown in Figure 2. Event listener code is created as part of the service implementation. This code listens for application events and upon receipt of an event, places the event within a local event store. The implementation of the event store depends on the method used to provide notifications from the application. If the application provides its notifications via JMS, then the event listener piece becomes unnecessary and the queue can be used directly as an event store. Finally, two additional operations need to be added to the service interface: one operation to check for new events by consulting the service event store for newly published application events, and another for fetching events sequentially from the store. External web-service enabled clients, such as mediators,

can then implement polling using the web-service interface.

This solution has conceptual advantages and disadvantages. It maintains uniformity with the concept of the web-service interface being a language and location-neutral equivalent to the concept of the interface in the object-oriented programming model, but assumes an active client process capable of performing polling. The alternate and more traditional web-services approach would be to have the event listener use WSIF to notify a second corresponding web-service. From an EAI perspective, this last approach is undesirable, since the secondary end-point may not be web-enabled and the event listener code becomes coupled with the secondary end-point's interface. Ultimately, an implementation can choose to have a web-service mediator perform the polling and notification of the corresponding web-service, without sacrificing architectural elegance.

Towards a Data-Driven Design

Data-driven software design is an important philosophy for enterprise integration. Software written according to the data-driven design principal is modeled as a runtime environment whose life-cycle and execution algorithms are determined from external configuration information. Data-driven software often consists of a series of loosely assembled, generic components that are dynamically loaded and configured at runtime according to user specification. Data-driven design encourages externalization of system parameters and often results in greater exposure of functionality to the end-user. From an integration perspective, data-driven designs are advantageous because they allow non-programmers to customize software without requiring in-depth knowledge of software internals. They also pave the way for streamlining the process of configuring complex software systems via tooling or machine-aided generation.

Mediators are ripe for data-driven design since the semantics of both mediated protocols are known in advance. By adopting a data-driven approach to design, mediators can be implemented as generic, reusable components, rather than requiring reimplementations for every additional service. Moreover, since web-service interface definitions are readily readable by machines, mapping between web-service requests and a corresponding integration protocol or technology could be streamlined by tooling or even determined dynamically at runtime.

Since mediators are essentially adapters that specialize in communications between two different protocols, traditional adapter design principals apply. Figure 3 shows a simplified architecture for a data-driven mediator. A runtime loader gets instantiated at start up and reads in an XML configuration file that describes which protocol handlers to load and how to configure those handlers. The runtime loader also configures the transformation service component and creates the processing pipeline between the two protocol handlers. The protocol handlers contain code for establishing, managing, and pooling connections over their designated protocol, as well as code for constructing

and processing requests. The transformation service maps requests between protocols using external user-provided configuration information. Upon receipt of a request, the request and its contents are passed to the transformation service component, which fetches the appropriate external configuration information for request processing, and provides the result of the transformation to the destination protocol handler for outbound delivery. Upon receipt of the response by the protocol handler, the response and its contents are passed back through the transformation service and the result is returned to the initiator of the request.

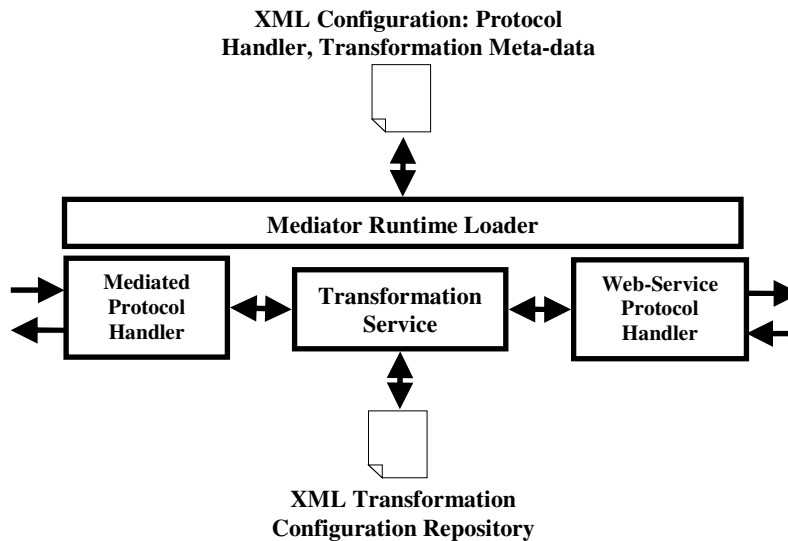


Figure 3 – Data-driven Mediator Architecture

There are many equally valid ways of implementing the transformation service, as well as good choices for the format of the external configuration data that drives transformations. XSLT is a prime candidate for data processing and transformation since XSLT is used as a transformational tool in many different integration middleware products. However, using XSLT as the format of choice requires that the mediated protocol handler accept an XML document as the data format for its internal interface. Open source technologies that provide Java-XML and Java-SQL bindings, such as Castor, are also good choices.

With mediators implemented as data-driven components, the service mediator architecture becomes considerably more attractive. New services are readily integrable with other middleware platforms, provided a mediator for that platform has already been previously written. Maintaining several generic, reusable mediators is considerable less taxing than the

maintenance of many specialized, monolithic adapters. In addition, using a data-driven design facilitates skill partitioning among IT staff, since it provides non-programmers with a greater degree of control. Data-driven mediator design also encourages the creation of tooling to streamline the process of configuring the mediators, which reduces the effort required to integrate new and maintain old functionality.

Implications for Designing Automated Processes

The architecture presented here suggests pushing a significant portion of business process functionality out to the end-points, both via the function offered by the mediator and the implementation of the service. As technology standards like BPEL become increasingly pervasive within integration middleware as a means of

describing business processes, they will influence the kinds of content that comprise the implementation of a business process. BPEL contains a considerable number of constructs for describing routing and control flow, but delegates transformational capabilities to other standards, such as XSLT, and considers transformational processing as an integrated part of the end-point interface.

Some integration middleware platforms mix the description of business process flow with data transformation, and consequently these processes will be significantly harder to port to future integration middleware platforms. Instead, the role of the integration middleware platform should be viewed as a centralized coordinator between loosely coupled services. The common data model used for data exchange is enforced not by the middleware tooling, but rather by the choice of schemas for the interfaces to the distributed services. The service-mediator architecture is an embodiment of this viewpoint: the web-service interface definitions declare the structure of request data, while mediators provide the mapping between the common data model used within the middleware and the specific parameters needed to issue the request.

Another advantage of the service-mediator architecture approach is that it forces the designer to harden the web-service interface to industry application or component function. Since legacy applications and distributed software components also have life-cycles, the integration architect must also account for their evolution in his or her design. The web-service interface provides a good point of change management; if a legacy application is being upgraded or swapped with another industry application, the implementation behind the web-service interface can be modified accordingly, without impacting business process design or dependent distributed services. Moreover, because the web-service interface is opaque, the underlying implementation could even be a composition of multiple distributed services, allowing for a transparent and graceful transition of function between legacy and replacement systems.

Conclusions

The rapid evolution of integration middleware has introduced a new challenge to the task of enterprise application integration. As a centralized point of communication and coordination, changes to integration middleware have the potential to impact nearly all facets of an IT infrastructure. The result is a costly predicament for enterprises: to compete

effectively, enterprises must reduce their costs and rapidly deploy new services and products today, while preserving their IT investment in this effort going forward.

This dilemma can be solved by adopting a design methodology that facilitates reuse and an integration architecture that is adaptable to change. The architecture presented promotes componentization of distributed function and maximizes the flexibility in how these distributed components are integrated. The semantics of how a service is integrated, including the interaction characteristics of particular protocols and integration technologies, are kept isolated from a service's functional interface; industry application and component functionality are modeled as web-services and mediators are introduced as a means of bridging a service's interface with a particular integration protocol or technology. In addition, because the semantics of both protocols are well known, mediators can be modeled as data-driven components, resulting in a single, reusable implementation for each integration protocol.

By designing with the evolution of integration middleware in mind, IT architects can help avoid version lock-in and significantly reduce long-term maintenance costs. In addition, enterprises can make investments in integration efforts today without the fear of losing all, if not the majority, of their investment as a result of technological evolution in the future. The result is a more robust integration infrastructure, and the freedom to focus solely on what's really important: creating a more dynamic and effective business.

References

- Business Process Execution Language for Web Services v1.1* [Online]. Available: <http://www-106.ibm.com/developerworks/library/ws-bpel/>.
- Coqueret, Regis & Fiammente, Marc, (2003). Choosing among JCA, JMS, and Web services for EAI. *IBM DeveloperWorks, March 2003*.
- Emmerich, W., (2000). Software Engineering and Middleware: A Roadmap. *Procs. of the conference on The Future of Software Engineering (ICSE2000), pages 117-129, ACME Press, May 2000*.
- Fremantle, Paul, (2002). Applying the Web Services Invocation Framework – Calling Services Independent of Protocols. *IBM DeveloperWorks, June 2002*.

Georgakopoulos, D., Hornick, M., Shet, A. (1995). An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2), April 1995, pages 119-153.

Mohan, C., (2002). *Dynamic e-Business: Trends in Web Services* [Online]. Available: <http://citeseer.nj.nec.com/mohan02dynamic.html>.

Papazoglou, Michael & Yang, Jian, (2002). Design Methodology for Web Services and Business Processes. *Procs. of the 3rd VLDB-TES Workshop, August, Hong Kong, Lecture Notes in Computer Science Vol. 2444, Springer, 2002.*

The Castor Project [Online]. Available: <http://www.castor.org>.

Web Services Architecture [Online]. Available: <http://www.w3.org/TR/ws-arch>.

Yang, Jian, (2003). Web Service Componentization: Towards Service Reuse and Specialization. *Communications of ACM, October 2003.*

Copyright

Copyright © 2003 Michael Gilfix.

The author(s) also grant a non-exclusive licence to EAI Industry Consortium to publish this document in full on the World Wide Web (prime sites and mirrors) and in printed form. Any other usage is prohibited without the express permission of the author(s).