By Michael Stal

Web Services: BEYOND COMPONENT-BASED COMPUTING

Seeking a better solution to the application integration problem.

"As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now that we have gigantic computers, programming has become a gigantic problem."

—Edsger W. Dijkstra,
ACM Turing Award Lecture, 1972

With the introduction of middleware technologies such as COM+ or Enterprise JavaBeans, component-based software development has finally become a major trend, at least when we focus on enterprise solutions. In addition, Web technologies as well as XML have gained broad application throughout the industry. In almost all solutions HTTP is leveraged as a bridge between the front-end Web browsers and Web servers, while components are used to implement workflow and persistent state in the back-end. Although the computation-driven back-end has always been subject to change, especially in recent years, the front-end has remained almost unchanged: it still uses a HTML-driven paradigm to transfer and display Web pages from Web servers to human users.

The naive combination of component-based middleware and Web technologies in order to integrate business processes and applications has proved to be insufficient for many reasons. For instance, this type of simple integration approach does not consider issues such as integration of different data models, workflow engines, or business rules, to name just a few. Enterprise Application Integration (EAI) solutions have become so wide-spread in B2B environments because they try to solve most of the aforementioned issues. However, the available EAI solutions are proprietary, complex to use, and do not interoperate well with each other. Thus, the question arises: Is there a better way to solve the integration dilemma, with a simple and interoperable solution? One solution is described here. The basic idea is to build Broker-based middleware using Internet protocols such as HTTP and XML as a data marshaling solution. This is the essence of Web services.

If Web Services are the Answer, What Exactly Is the Problem?

In the last few years, Web-based network technologies have become increasingly important for IT solutions. This trend leads to fully connected information systems, but also causes a number of problems developers must address. For example, all kinds of devices should be able to be connected to network-based systems. Software systems are expected to drive business processes that are no longer constrained by computer-related or company-related boundaries. Hence, one question needs to be resolved in this context. How can we connect isolated islands to produce integrated solutions?

Technically, the integration problem can be partitioned into a set of aspects emerging on different layers. The heterogeneity in these layers is primarily caused by different:

- network technologies, devices, and OSs;
- middleware solutions and communication paradigms;
- programming languages;
- services and interface technologies;
- domains and architectures; and
- data and document formats.

Even worse, the problem becomes a multidimensional problem when integrated solutions must also cope with non-functional requirements such as security aspects, availability, transactions, to name just a few.

All these facets of heterogeneity have offered many benefits and were intentionally brought into IT solutions. As a consequence, the goal should not be to try and eliminate the heterogeneity, but to help developers to master it and manage it. There have been many approaches to integrate heterogeneous technologies. However, these EAI technologies cannot provide a common solution because they try to solve the problem using an (incomplete) set of proprietary technologies. For example, when multiple companies integrate systems that were themselves integrated using different EAI products, developers face the recursive problem of integrating integration solutions. Hence, it is necessary to establish a holistic, commonly accepted and standardized approach instead of the proprietary solutions mentioned here. In this approach the systems and technologies remain heterogeneous, but their interface and collaboration patterns are standardized using lightweight standards such as XML and Internet protocols. OO middleware and component technologies such as EJB, CORBA, or COM+/.NET have already helped to solve many integration problems without causing architectural drift. What about combining Web technologies with OO middleware technologies? Finally, we have entered the world of XML Web services.

Architectural Perspective

To understand the evolution initiated by Web services and the relationship of this technology to existing approaches, it is necessary to address the architectural concepts of existing middleware. What are the fundamental software patterns of OO middleware solutions and how do they map to Web service infrastructures?

Using Client and Servant Proxies. One of the primary issues when implementing OO middleware is the provision of location-transparency and the hiding of communication details from the programmer. Examples of such low-level details include the data encoding and the communication protocol used. A common and proven solution to this problem is the application of the well-known Proxy design pattern [1].

The basic idea behind this pattern is to introduce a proxy component as an intermediate layer between the client and the servant (see Figure 1). The proxy resides within the address space of the client and implements exactly the same interface(s) as the servant. Aggregation of interfaces might be provided by the Extension Interface design pattern [2]. Using this approach, a client can remain oblivious to any details related to distribution, such as the servant location or communication protocols used.

At runtime, the proxy acts as an interceptor. It intercepts the client's method requests, obtains all arguments from the stack, and wraps the information in packets according to the wire format of the underlying communication protocol. Then, it sends the data directly to the servant. Finally, the proxy retrieves the results from the servant, copies them to the client's stack, and returns to the caller. From the client perspective all communication-related activities are hidden by the proxy. The solution developed so far is somewhat oversimplified: another proxy needs to be introduced on the servant side for the same reason we have introduced a proxy on the client side. The servant proxy is introduced to free the servant from knowledge such as the client location and the details of the communication protocol.

In summary, communication is never handled by clients and servants directly. Instead, intermediaries called proxies take responsibility for location and communication aspects. This property is often denoted as distribution transparency. An interesting issue arises when dealing with this kind of proxy-based approach. How should we structure the representation of requests, responses, and faults? Basically, client and servant programmers prefer a method invocation paradigm that fits nicely into the object models of their favorite programming languages. However, ultimately every communication paradigm is layered on top of a low-level transport protocol such as TCP/IP. As a consequence, each remote communication must be handled by message passing. Thus, in the guts of every implementation requests, responses, and faults are represented as first-class objects upon transmission over the wire. For example, each method request might be encapsulated by a structure such as the following:

```
struct MethodRequest {
    String obj_reference;
    String method_name;
    Object[] arguments; // in and in/out-arguments
    // a lot of organizational overhead
}
```

Broker Architectures. The usage of proxies is not

sufficient for implementing remote method invocations, because some forces are not addressed by the Proxy design pattern. For instance, in a pure proxy-based solution, location information is separated from clients and servants by the proxy layer. Within the proxies themselves, however, location information must be hard-coded, thus restricting both location and migration transparency. Another unresolved problem with the Proxy design pattern is the issue of servant activation and deactivation. In a naive approach, servant implementations would be activated at system startup and deactivated at system shutdown. Since servants are mostly idle between creation and removal time, this would lead to unacceptable high resource usage and low scalability. To solve these problems additional participants need to be introduced in the overall architecture of OO middleware [1], as shown in Figure 2(a). The broker component denotes a globally available participant that maps logical servant references to physical server locations. In addition, a broker is in charge of handling activation and deactivation issues, but we will not discuss this issue here for the sake of brevity.

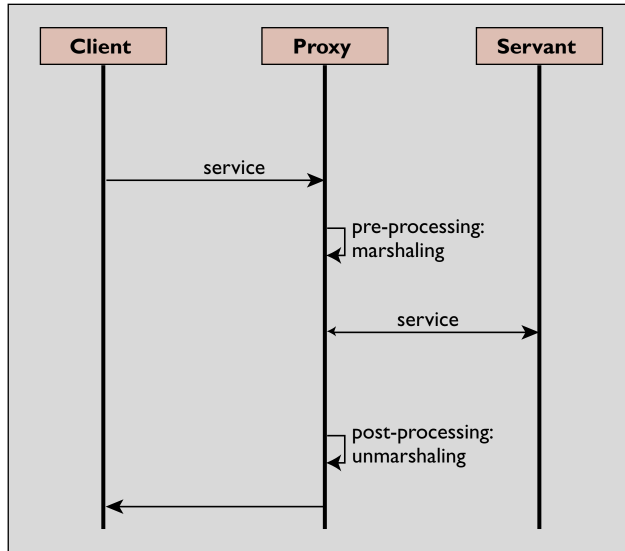


Figure 1. Proxy component as an intermediate layer between client and servant.

To reduce the number of operating system processes required multiple servants may reside within the same servers—see Figure 2(b). In this case, the server must provide dispatch functionality to dynamically map and forward method requests to the correct servants. In contrast to the previous proxy-based scenario, servers (or, alternatively, servants depending on the concrete implementation) register with the central broker component using a shared library or central data store. Typically, they obtain a free port from the broker that they use to listen for incoming requests. In other variants, the servers might themselves determine their communication ports. Client proxies in both variants don't carry any hard-coded location dependencies. Instead, they access the broker on initial communication establishment to retrieve the servant's communication endpoint. After a communication channel is established, the message flow between client and server proxies is transmitted through this dedicated channel. In summary, the Broker architectural pattern introduces a central repository to map logical object references to physical ports.

Generation of Glue. Assuming that Broker-based architectures are used to implement efficient OO middleware, one question still needs to be addressed: Who is in charge to provide all of these necessary proxy implementations? Obviously, it is not feasible to handcraft the proxies, because this activity is tedious, error-prone and time-consuming, even for trivial examples. Thus, the generation of "glue" should be automated in a perfect world. If glue generation is supposed to be handled automatically by tools, a means must be provided to express structural interface-related information as well as deployment issues in a platform-, communication-, and possibly even programming language-independent way. For this purpose, an interface description language is introduced. Interfaces could be described in the programming language of choice using coding conventions to restrict the interface specifications according to the limitations imposed by the underlying distributed object model. Another option is to introduce a language-independent interface definition language. No matter which decision is taken, a generator is used to parse the interface definitions and automatically generate client-side proxies and server-side proxies.

Web-based Middleware. The Broker pattern describes the common architectural concepts behind OO middleware such as OMG CORBA, Microsoft COM+, or Java RMI. Despite their common architectural foundation, these concrete implementations differ in many aspects, most notably in the object models provided, as well as in the implementation of their application-level communication protocols and marshaling code. In addition, all of these technologies were not implemented with the Web in mind. As a consequence, additional efforts are required to transmit CORBA or COM+ method requests seamlessly over HTTP. Such solutions are indeed available today, but rely on proprietary approaches. Thus, it is almost impossible today to invoke a CORBA servant from a Web-based COM client. Obviously, integrating existing OO middleware into the Web has not solved the interoperability problems. So what about integrating

the Web into OO middleware? If we were going to implement a Broker architecture using Internet protocols and XML, how would we design such Web-based middleware?

Step 1: Implementing the Protocol. The fundamental layer of each middleware is the implementation of the communication protocol(s). A protocol defines syntax, semantics, and order of messages exchanged between peers. To implement a transport protocol for Web-based middleware, Internet standards such as HTTP must be leveraged. In addition, we must provide a self-describing data representation format to structure and encode the messages exchanged between client-side and server-side proxies using this transport layer. As a matter of fact, XML denotes the basic means to express data representation formats. All of the requirements for providing a Web-based communication protocol are already fulfilled by the Simple Object Access Protocol (SOAP)—see www.w3.org/2002/ws/—which provides all those facilities mentioned previously. It leverages XML to structure messages and Internet protocols such as HTTP, SMTP, FTP or Telnet as message carrier. In the following, an example the encoding of a simple service request is illustrated. The method getPhoneNumber expects a string as parameter. All relevant data is contained within the sub-element <soap:Body>. Optional headers help to send additional context.

```
<soap:Envelope>
   <soap:Header>
         <transaction>
         <soap:mustUnderstandstand= "true"
         xmlns= "http://tx.com">
               <id> 12345678 </id>
         </transaction>
   <soap:Header>
   <soap:Body>
         <m:getPhoneNumber>
               <theName> Bill Gates </theName>
         <m:/getPhoneNumber>
   </soap:Body>
</soap:Envelope>
```

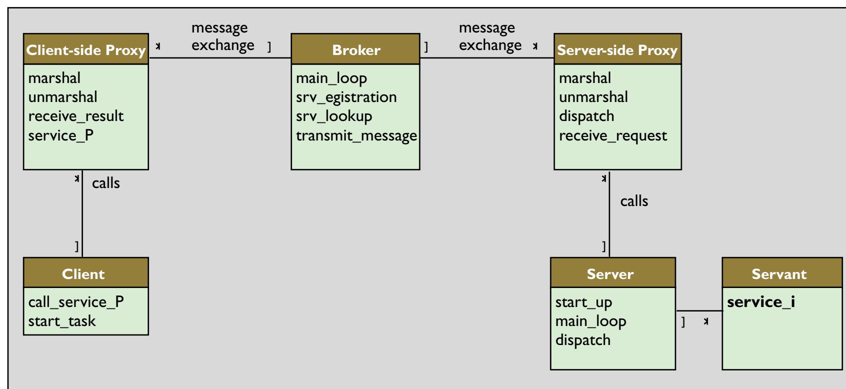
Step 2: Implementing an Interface Definition Language. In the next implementation step, an interface definition language is introduced to express structural and deployment information in an implementation-neutral fashion. From the specifications written in this language, tools will automatically generate the implementations of client-side and server-side proxies. Again, XML represents the appropriate means to define such a data representation language. As a result WSDL (Web Services Description Language)—see www.w3c.org/TR/wsdl—was created providing the following constituents:

- *Types* are used as core elements to build messages (XML Schema Notation).
- *Messages* define packages exchanged within a single message transfer. Requests and responses represent separate messages.
- *Porttypes* group messages to abstract operations.
- *Bindings* map Porttypes to concrete protocols.
- *Ports* denote the concrete communication addresses of services.
- A *Service* comprises a collection of ports.

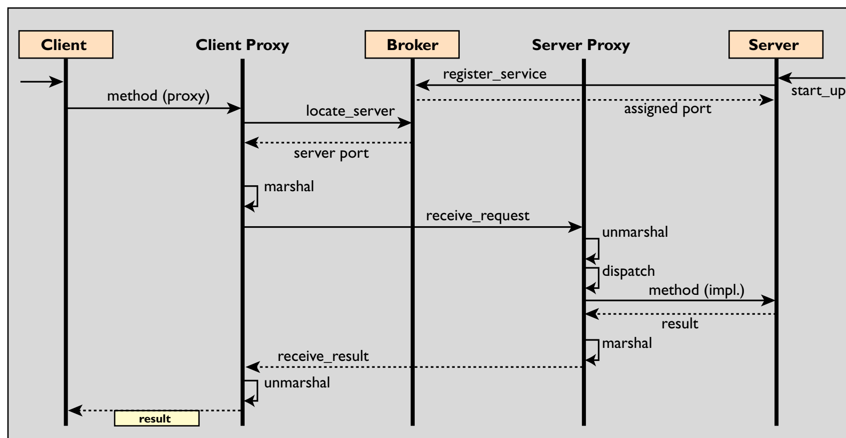
Step 3: Implementing a Service Directory. Before a client can access a service, it must find the service. For this purpose, a central broker must be available that allows implementers to register their services as well as clients to locate these services. Again, XML denotes the core technology to store and retrieve service registrations. UDDI (Universal Discovery, Description, and Integration)—see www3.ibm.com/services/uddi/standard.html—provides all functionality expected from a service broker. In UDDI, servers use the Publishers API to register services as well as additional business information with the global repository (see Figure 3). Clients access the Inquiry API to browse the repository and retrieve service descriptions. SOAP is used as communication protocol in all interactions. The client obtains the WSDL description from the UDDI repository both dynamically or statically, generates a client-side proxy, and invokes the Web service.

A Broker Is Not Enough. As was illustrated, Broker-based Web service implementations abstract away many low-level communication details. The main aspect they offer is the seamless (platform-independent) integration of programming language object models into a distributed context. For building sophisticated solutions it is not sufficient to provide a stack of protocol layers, there are many problems that must additionally be solved—a few examples include:

- Information and services must be consumable from any device, any time, any place. For this purpose, a device-independent framework needs to be available. This can be effectively achieved by providing a virtual execution system such as the Java VM or Microsoft's CLR.
- Web-based, front-end layers must provide both Web server functionality and Web service provisioning.
- Web services infrastructures should support context information such as transaction IDs, security information, and location. Thus, the result of a



a)



b)

Figure 2. a) The broker component maps logical servant references to physical server locations;
b) multiple servants may reside within the same servers.

Web service computation does not only depend on the (functional) arguments but also on the context of the service invocation.

- Web services are no islands. They can be connected to complete workflows, thus addressing B2B and B2C contexts. Workflow engines control both the computation flow and the application of business rules. They often represent gateways to existing back-end servers such as database systems, ERP servers, mainframes, or legacy code. Therefore, strong support for back-end integration is required. This is the major reason why existing OO middleware does not become obsolete due to Web services. In fact, OO middleware and especially component technologies such as EJB will become more important as an integration layer.
- Reuse and integration are supported by introducing core services application programmers might facilitate. For instance, calendar services, single sign-on functionality, and events are potential candidates for such reusable horizontal services.

Web services solutions cannot restrict themselves to UDDI, WSDL, and SOAP, because the development of sophisticated applications requires more than just a few communication layers. Instead, Web services protocols need to be supplemented by many additional technologies. In the future, whole Web service frameworks will be available addressing all or at least most of the issues explained here. Solutions such as Microsoft .NET and Sun ONE are only the beginning.

Integration via Web Services. As discussed in the previous section, Web services represent an integration technology. In contrast to other approaches such as proprietary EAI solutions, they are completely based on Web standards and XML. In combination with other standards they can help solve different kinds of integration problems:

- Web services frameworks as proposed here include virtual execution engines that can be ported to new environments very quickly, thus providing immediate platform integration. With platform integration we can hide dependencies on devices, operating systems, middleware solutions, and programming languages.
- On the low-level communication layers Web ser-

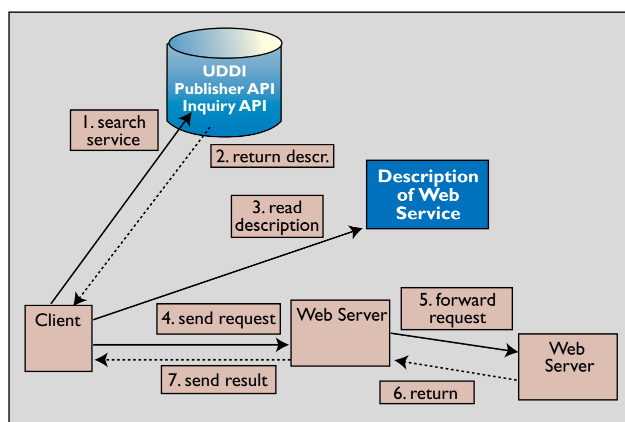
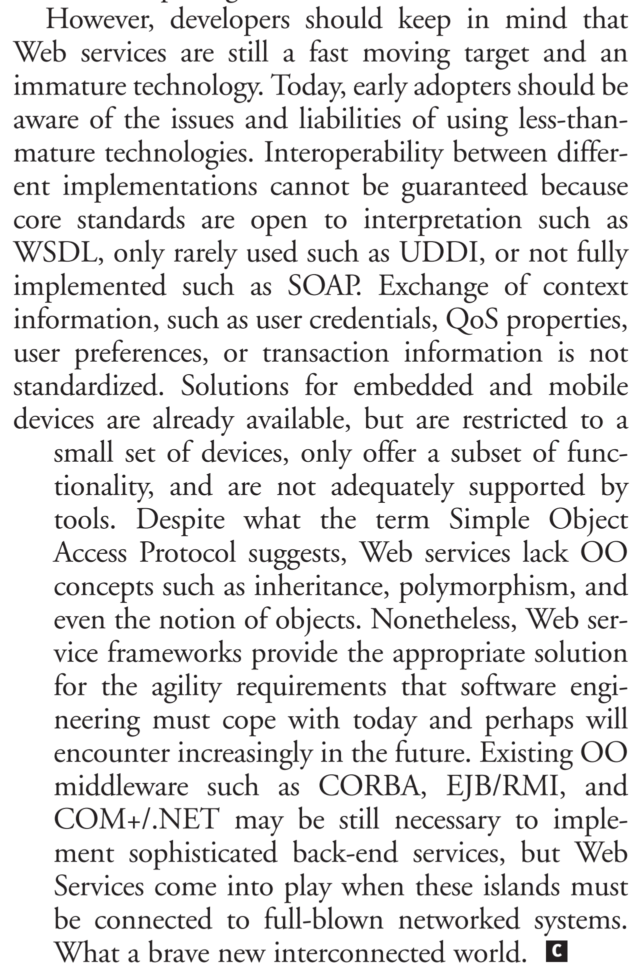


Figure 3. Example of a UDDI server scheme.

vices protocols such as SOAP, WSDL, and UDDI help to easily integrate existing OO middleware into the Web, and vice versa. Many products are available to connect CORBA, EJB, COM+ or .NET-systems via Web services. In addition, Web server implementations such as IIS or Apache can be extended to deliver both Web services and Web pages.
- To integrate isolated workflows and services, a XML-based Web Services Flow Language (WSFL) has been proposed by IBM. However, this technology is immature and further research needs to be done in this area.
- When documents and data are going to be exchanged between heterogeneous applications, mappings must be provided to adapt these proprietary entities to the formats the receivers understand. X-Schema, XSLT, and related standards are considered ideal solutions for this kind of data representation and transformation.
- In the future context information exchanged between Web services and their clients should be standardized, thus enabling even integration at the level of horizontal services.



Conclusion

Web services are a promising technology that will increasingly help to integrate heterogeneous islands to homogeneous component-based solutions. This evolutionary technology is solely based on the architectural concepts of OO middleware as well as on widespread and commonly accepted standards such as XML and Internet protocols. This is why key players such as Sun Microsystems, Microsoft, IBM, BEA, and Hewlett-Packard consider Web services the major technological trend of tomorrow's networked computing.

However, developers should keep in mind that Web services are still a fast moving target and an immature technology. Today, early adopters should be aware of the issues and liabilities of using less-than-mature technologies. Interoperability between different implementations cannot be guaranteed because core standards are open to interpretation such as WSDL, only rarely used such as UDDI, or not fully implemented such as SOAP. Exchange of context information, such as user credentials, QoS properties, user preferences, or transaction information is not standardized. Solutions for embedded and mobile devices are already available, but are restricted to a small set of devices, only offer a subset of functionality, and are not adequately supported by tools. Despite what the term Simple Object Access Protocol suggests, Web services lack OO concepts such as inheritance, polymorphism, and even the notion of objects. Nonetheless, Web service frameworks provide the appropriate solution for the agility requirements that software engineering must cope with today and perhaps will encounter increasingly in the future. Existing OO middleware such as CORBA, EJB/RMI, and COM+/.NET may be still necessary to implement sophisticated back-end services, but Web Services come into play when these islands must be connected to full-blown networked systems. What a brave new interconnected world. **C**

References

1. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture—A System of Patterns.* Wiley, New York, 1996.
2. Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. *Pattern-Oriented Software Architecture—Patterns for Networked and Concurrent Objects*. Wiley, New York, 2000.

Michael Stal (Michael.Stal@mchp.siemens.de) is Senior Principal Engineer at Siemens AG in Munich, Germany.

© 2002 ACM 0002-0782/02/1000 $5.00