(19) **United States**

(12) **Patent Application Publication**

Karakashian et al.

(10) **Pub. No.: US 2007/0150546 A1**

(43) **Pub. Date: Jun. 28, 2007**

(54) **WEB SERVICES RUNTIME ARCHITECTURE**

(75) Inventors: **Todd Karakashian**, San Francisco, CA (US); **Manoj Cheenath**, San Ramon, CA (US); **Adam Messinger**, San Francisco, CA (US)

Correspondence Address:
FLIESLER MEYER LLP
650 CALIFORNIA STREET
14TH FLOOR
SAN FRANCISCO, CA 94108 (US)

(73) Assignee: **BEA SYSTEMS, INC.**, San Jose, CA (US)

(21) Appl. No.: **11/682,164**

(22) Filed: **Mar. 5, 2007**

Related U.S. Application Data

(63) Continuation of application No. 10/366,236, filed on Feb. 13, 2003.

(60) Provisional application No. 60/359,098, filed on Feb. 22, 2002. Provisional application No. 60/359,231, filed on Feb. 22, 2002.

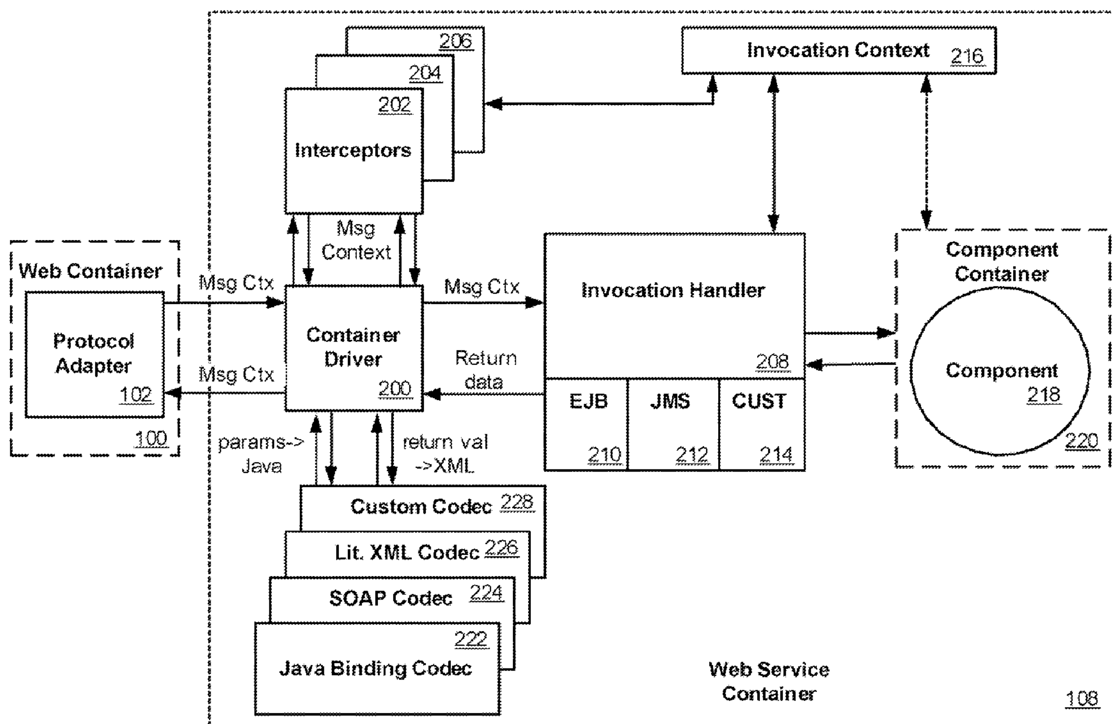
Publication Classification

(51) **Int. Cl.**
G06F 15/16 (2006.01)

(52) **U.S. Cl.** **709/207**

(57) **ABSTRACT**

A runtime architecture for Web services utilizes a container driver to accept an invoke request for Web services. The container driver performs any necessary data binding/unbinding required to process the invoke request and associated message context, utilizing an appropriate plugin component. An interceptor receives the context information and modifies the message context for Web service compatibility. An invocation handler receives the formatted context information and passes parameters from the message context to the target of the request. The invocation handler processes values returned from the target and passes them to the container driver, which can formulate and return a response, along with the message context, to the client or protocol adapter. This description is not intended to be a complete description of, or limit the scope of, the invention. Other features, aspects, and objects of the invention can be obtained from a review of the specification, the figures, and the claims.



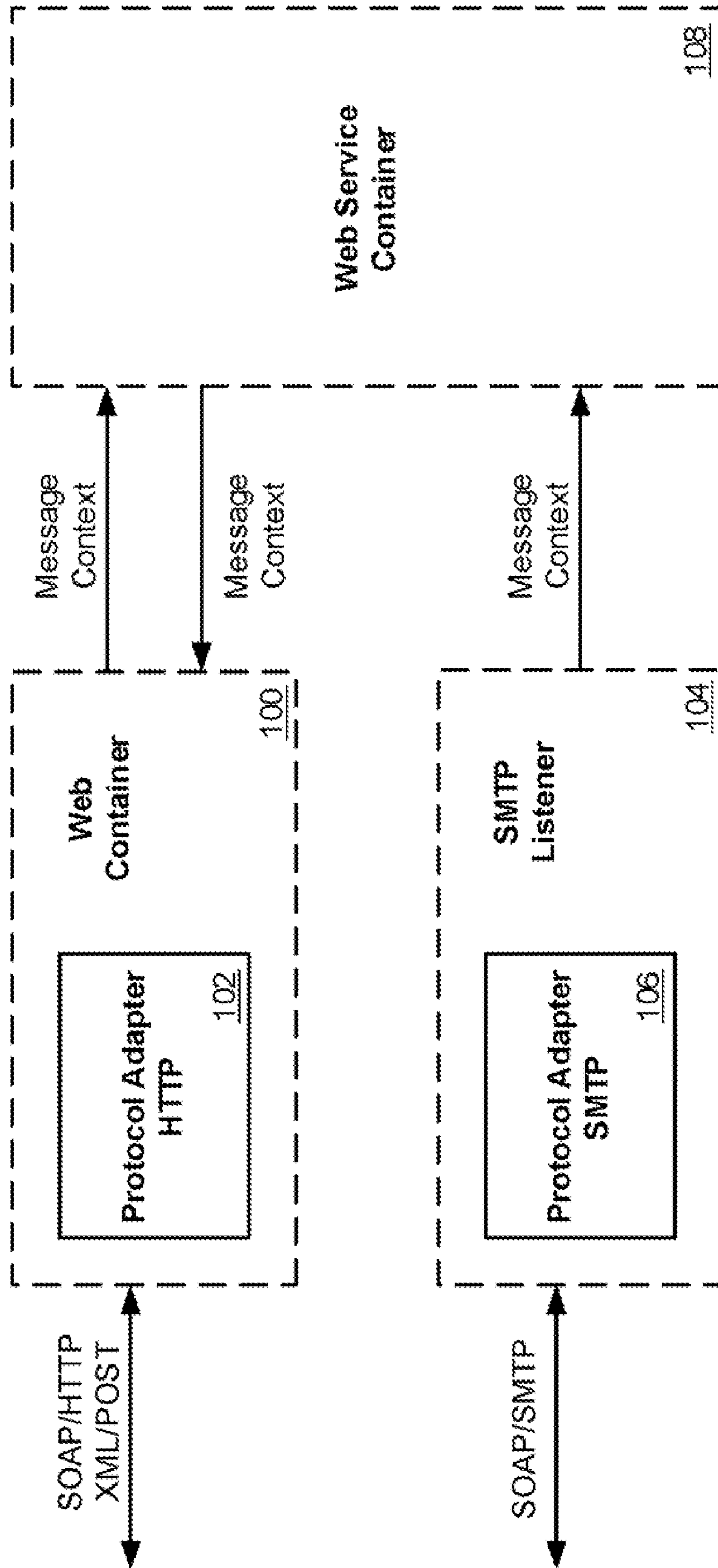


Figure 1

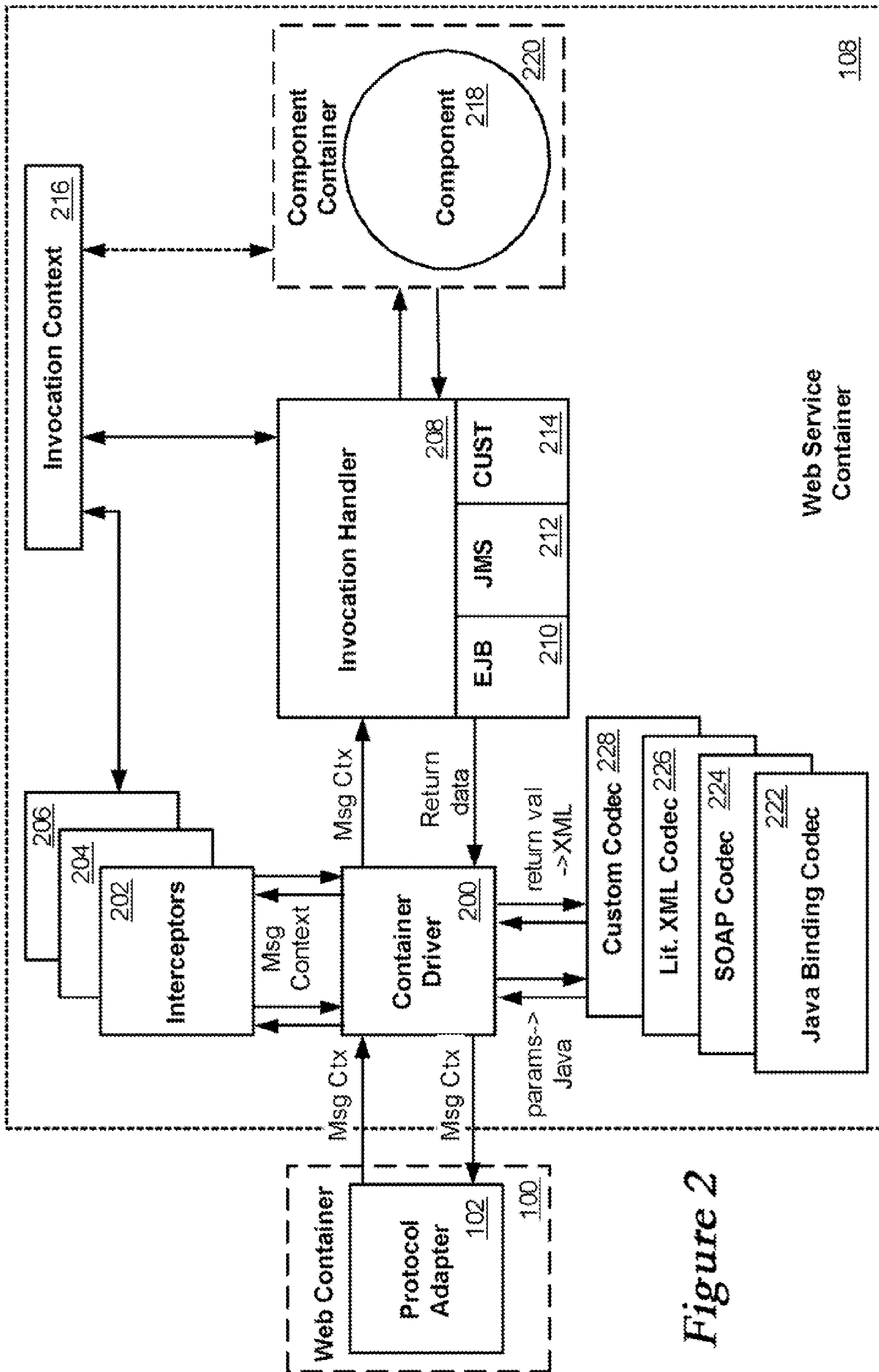


Figure 2

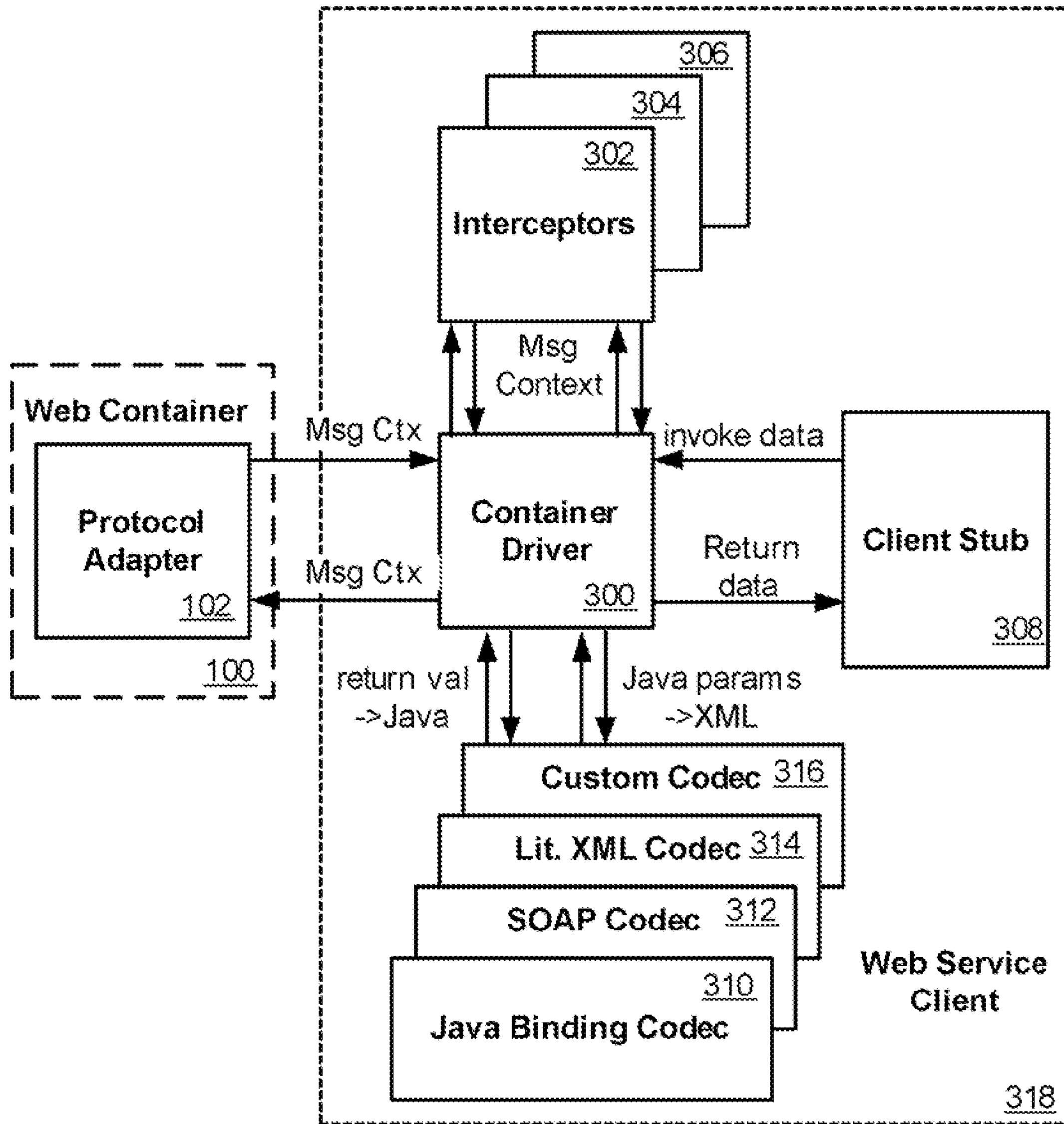


Figure 3

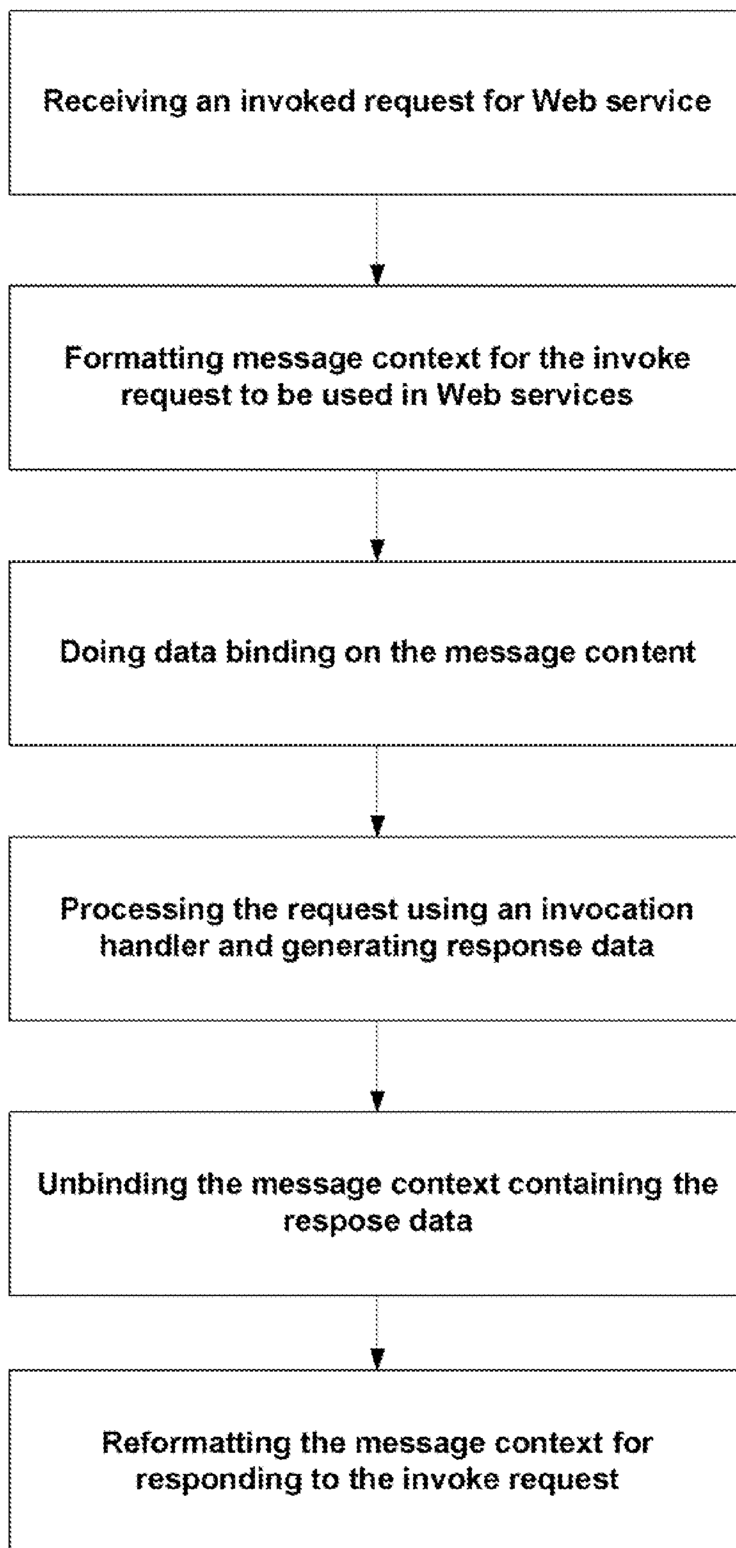


Figure 4

WEB SERVICES RUNTIME ARCHITECTURE

CLAIM OF PRIORITY

[0001] This application is a Continuation of and claims priority to U.S. patent application Ser. No. 10/366,236, filed Feb. 13, 2003, entitled "WEB SERVICES RUNTIME ARCHITECTURE," which application claims priority to U.S. Provisional Patent Application No. 60/359,098, filed Feb. 22, 2002, entitled "WEB SERVICES RUNTIME ARCHITECTURE," as well as U.S. Provisional Patent Application No. 60/359,231, filed Feb. 22, 2002, entitled "WEB SERVICES PROGRAMMING AND DEPLOYMENT," each of which is hereby incorporated herein by reference.

COPYRIGHT NOTICE

[0002] A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document of the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

FIELD OF THE INVENTION

[0003] The present invention relates to the implementation of web services.

BACKGROUND

[0004] Web services are becoming an integral part of many application servers, with an importance that can rival HTTP or RMI stacks. Java standards for Web services are being developed through the Java Community Process. Businesses are building important applications on top of Web services infrastructures, such as is available in WebLogic Server from BEA Systems of San Jose, Calif. Presently, however, there is no complete implementation of Web services upon which to build.

BRIEF SUMMARY

[0005] A system and method in accordance with the present invention overcome deficiencies in the prior art by utilizing a runtime architecture for Web services. A container driver is used in the architecture for accepting an invoke request for Web services, such as from a protocol adapter. The container driver can perform any data binding and unbinding required to process the invoke request, utilizing a plugin component such as a Java binding codec, SOAP codec, XML codec, or custom codecs.

[0006] An interceptor can receive the context information for the invoke request from the container driver and modify the message context to be used with Web services. An invocation handler can receive the context information from the container driver after the message context has been modified by the interceptor. The invocation handler can pass parameters from the message context to the target of the request and process values returned from the target. The invocation handler can then pass these values to the container driver, such that the container driver can formulate a response to the invoke request. The response and message context can then be returned to the client or protocol adapter.

[0007] Other features, aspects, and objects of the invention can be obtained from a review of the specification, the figures, and the claims.

BRIEF DESCRIPTION OF THE DRAWINGS

[0008] FIG. **1** is a diagram of a system in accordance with one embodiment of the present invention.

[0009] FIG. **2** is a diagram of a Web service container that can be used with the system of FIG. **1**.

[0010] FIG. **3** is a diagram of a Web service client that can be used with the system of FIG. **1**.

[0011] FIG. **4** is a flow chart of a method fo the present invention.

DETAILED DESCRIPTION

[0012] Systems and methods in accordance with one embodiment of the present invention can overcome deficiencies in existing Web service implementations by providing a more stable, complete implementation that is suitable as an application integration platform.

[0013] A Web services architecture can allow for communication over a number of transports/protocols. HTTP can continue to be a primary transport mechanism for existing applications, while transport mechanisms such as SMTP, FTP, JMS, and file system mailboxes can also be supported.

[0014] Message formats such as SOAP 1.1 and 1.2 with attachments can be used as primary message formats. It is also possible to accept Web service requests that are XML-encoded and submitted via HTTP posts. A Web services architecture can support plugging in other message formats and provide a mechanism for access to the raw message for user code.

[0015] A Web services architecture can utilize a stack that supports a standards-based default binding of fundamental XML data types supported in various Web service platforms. An API can be used to allow other binding mechanisms to be plugged in. The binding to be used can be specified on a per-operation basis.

[0016] Various security and messaging standards require a context or state to be associated with Web service messages. A Web services architecture can support the processing of multiple message contexts, such as those related to security or conversation ID, and can offer user code access to these contexts. Many of these contexts can be encoded into a SOAP header, although it is also possible to include the contexts in the underlying protocol layer (e.g., cookies in HTTP) or in the body of the message (e.g., digital signatures). A Web services container can explicitly process contexts associated with planned security features for Web services.

[0017] A Web services stack can support a number of dispatch and synchrony models. Dispatch to both stateless and stateful components can be supported, using both remote procedure call (RPC) and messaging invocation semantics. Synchronous and asynchronous processing of requests can also be supported. In particular, it can be possible to enqueue an incoming message, to enqueue an outbound message, and to make asynchronous outbound calls. Enqueuing involves downloading files, one at a time, from a queue.

[0018] A component such as a session EJB can be used to implement application-hosted Web services, such as for business logic. An API can be provided for sophisticated users to integrate other types of components, even custom components, into a Web service mechanism. This may be used rarely, such as by developers who wish to build higher-level facilities and infrastructure on top of application-specific Web services.

[0019] A Web services architecture should not preclude the embedding of a Web service container in a lightweight server running in a more restricted Java platform, such as J2ME/CDC. A very thin Web service Java client, such as less than 100 kb, can also be supported. Such an architecture should not preclude support for future Web service standards, such as JAX-RPC. However, due to the present lack of maturity and quality of Java Web Service standards, application-specific APIs can be defined for the implementation of Web services.

[0020] A runtime Web services architecture can support both synchronous and asynchronous ("one-way") RPC style Web services, such as may be backended by an EJB. Message-style Web services can also be supported, which allow content to be submitted to and received from a JMS destination. Endpoints can be supported in a transport-specific way. These endpoints can associate a transport-specific address with a Web service. For instance, in HTTP a particular address can be associated with a Web service to be invoked. For SMTP an email address can be associated with a Web service.

[0021] FIG. **1** shows the relationship of a Web container **108** and SMTP listener **104** and a host server or Web service container **108**, utilizing an architecture in accordance with one embodiment of the present invention. An HTTP protocol adapter **102** is shown in the Web container **100**. A protocol adapter for HTTP **102** is shown in a Web container **100**, that can intercept a Web service invoke via HTTP from a Web services client. A protocol adapter for SMTP **106** is also shown in an SMTP listener **104**, which can accept a Web service invoke via SMTP. This architecture allows for pluggability in a number of places.

[0022] FIG. **2** shows a diagram of the architecture of the Web service container **108** of FIG. **1**. The HTTP protocol adapter **102** of the Web container **100** is shown passing message context to, and receiving message context from, a container driver **200**. The container driver **200** receives the message context from the protocol adapter **102** and sends the message context to the registered inbound interceptors **202**, **204**, **206**. After extracting the operation parameters and performing any necessary data binding, such as by using a Java Binding codec **222**, a SOAP codec **224**, an XML codec **226**, or a custom codec **228**, the container driver **200** submits the operation request to the appropriate invocation handler **208**, such as for EJB **210** or JMS **212**, or to a customized invocation handler **214**. After receiving data back from the invocation handler **208**, the container driver **200** can perform any data unbinding using the appropriate codecs **222**, **224**, **226**, **228** and send the response to the outbound interceptors **202**, **204**, **206**. The container driver **200** can then return the response to the protocol adapter **102**. The protocol adapter, interceptors, and invocation handler can each have access to an invocation context object **216**. The invocation handler **208** can also provide context access to the component **218** to which it delegates, which can be contained in a component container **220**.

[0023] A message context is a representation of a Web service invocation flowing through a container. A message context can contain a request message, which is the Web service request. A message context can be rendered into the canonical form of SOAP plus attachments. A response message is the Web services response, or at least a placeholder for the response if the response has not been formulated yet. A response message can also be in the canonical form of SOAP plus attachments. Transport information can contain relevant information that is specific to the transport over which the request came, and over which the response must be sent. For example, the transport information can contain the HTTP request and response streams for HTTP transport. An invocation context can also be used, which is described below.

[0024] A protocol adapter can be inserted into the subsystem of a host server. A protocol adapter can be responsible for processing incoming requests for a particular transport/protocol, such as HTTP or SMTP. This allows the Web service container to process Web service messages in various formats that are sent over multiple protocols. It will also allow the Web service container to reside in different kinds of servers. One condition for a protocol adapter is that the host server can support the protocol and that the message format can be converted into SOAP internally. There are no known important message formats that cannot be expressed via SOAP.

[0025] A protocol adapter can be responsible for identifying requests as Web service messages, as well as routing the messages to a Web services container. If the protocol being used supports synchronous responses, a protocol adapter can also receive the response data and return the data to the originator of the request. The protocol adapter can convert the message to the original message format if it is not SOAP plus attachments. A protocol adapter can deal with any message context that is required by the container, such as a conversation ID, and is transmitted at the protocol level, such as cookies in HTTP. The protocol adapter can propagate the message context to and from a Web services container.

[0026] The actual implementation of protocol adapter functionality can depend on the architecture of the host server, as well as the way that the protocol is hosted in the server. For example, the functions of a protocol adapter can be implemented in part by the normal function of a Web container for an HTTP protocol. Due to the dependency of protocol processing on server internals, there may not be many public protocol adapter APIs.

[0027] An invocation context can be used, which is an inheritable thread-local object that can store arbitrary context data used in processing a Web service request. The context can be available from various components of the architecture involved in the processing of the request and response. Typical data that might be stored in such a context are a conversation ID, a message sequence number, and a security token. A particular invocation handler can choose to make the invocation context available to the target component. This can allow application code to read and write to the invocation context.

[0028] An architecture can utilize interceptors. Interceptors are plugins that can provide access to inbound and

outbound Web service messages. An interceptor API can be public, and an implementation of an interceptor API can be part of a Web service application. An interceptor can modify SOAP messages as required. An interceptor can also read and write information on the invocation context. Interceptors can be associated with either operation, or with the namespace of the message body.

[0029] There are different types of interceptors. Header handlers can be used, for example, which operate only on message headers. Header handlers must declare which message headers they require so that the header information can be exposed, such as in the W3C Web service definition language (WSDL) generated for the Web service. Flow handlers, on the other hand, can operate on full message content. Flow handlers do not require a declaration of which message parts are processed, and do not result in the existence of any additional information in the generated WSDL. Application developers may use header handlers primarily, while business units that are building infrastructure on top of an application server may choose to use flow handlers. Both APIs, however, can be public.

[0030] XML serialization and deserialization plugins can be supported, which can handle the conversion of method parameters from XML to Java objects and return values from Java to XML. Built-in mappings for the SOAP encoding data types can be included with an application server. The processing of literal XML data that is sent outside any encoding can also be supported, as well as Apache "Literal XML" encoding. Users can also implement their own custom data type mappings and plug those mappings in to handle custom data types.

[0031] A container driver can be used with a Web services architecture in accordance with one embodiment of the present invention. A container driver can be thought of as the conceptual driver of a Web service container. A container driver can implement the process flow involved in performing a Web service request.

[0032] For RPC Web services hosted on an application server, the default target of a Web service invocation can be an EJB instance. For message-style Web services, the default target can be a JMS destination. In certain cases, it may be desirable to allow other components or subsystems as targets. People can build middleware infrastructure on top of application servers to require this functionality. Therefore, an invocation handler API can be supported to allow the Web service requests to be targeted at different components besides EJBs.

[0033] An invocation handler can insulate the Web service container from details of the target object lifecycle, transaction management, and security policies. The implementer of an invocation handler can be responsible for a number of tasks. These tasks can include: identifying a target object, performing any security checks, performing the invocation, collecting the response, and returning the response to the container driver. The implementer can also be responsible for propagating any contextual state, such as a conversation ID or security role, as may be needed by a target component.

[0034] A protocol adapter can perform the following steps in one embodiment. The protocol adapter can identify the invocation handler of the target component deployment, such as a stateless EJB adapter. The protocol adapter can identify any additional configuration information needed by the invocation handler to resolve the service, such as the JNDI name of a deployed EJB home. This information can be in the deployment descriptor of the protocol adapter deployment, such as a service JNDI name, and/or the information could be in the headers or body of the request or in the protocol.

[0035] A protocol adapter can identify the style of a Web service request, such as one-way RPC, synchronous RPC, or messaging. If necessary, a protocol adapter can convert an incoming request message into the SOAP with attachments canonical form. A protocol adapter can create a message context containing the request, a placeholder for a response, information about the transport, and information about the target invocation handler. A protocol adapter can also dispatch message context configuration to the Web service container.

[0036] A container driver can manage the flow of processing in the container. The container driver can receive the message context from the protocol adapter and, in one embodiment, sequentially performs the following steps. The container driver can dispatch to registered inbound interceptors, extract operation parameters, and perform data binding. The container driver can submit the operation request to the appropriate invocation handler, which can delegate the invoke to a target object. The container driver can receive a response from the invocation handler, possibly including a return value. If there is a return value, the container driver can perform data unbinding. If the synchrony model is request-response, the container driver can formulate a SOAP response. The response can be dispatched to registered outbound interceptors and returned to the protocol adapter for return to the caller.

[0037] The protocol adapter can return the SOAP response to the caller, converting the response back to the original message format if it was not SOAP. The protocol adapter, interceptors, and invocation handler can each have access to the invocation context object. Any necessary state needed during request processing can be propagated through this context. The invocation handler can also provide access to the context, such as to the component to which the invocation handler delegates.

[0038] An invocation handler that has been targeted to process an invoke can receive the following data from the container: the operation name, an array of Java Object parameters, any invocation handler configuration data, and the invocation context. The invocation handler can perform the invocation and return an array of Java Object return values.

[0039] An invocation handler can perform the following steps for one method in accordance with the present invention. A target object can be identified for the invocation. The invocation can be performed bypassing the parameters to the target. The invocation context object can be provided to the target. Also, a transaction, security, or component-specific context can be passed to the target object. Any return value(s) from the target can be processed and returned to the container driver.

[0040] An API can be used for invocation handlers. Invocation handlers can be configured when the protocol adapter is deployed. For example, the HTTP protocol handler can be a Web application.

[0041] Many types of built-in invocation handlers can be used. One such invocation handler is an EJB invocation handler. EJB invocation handlers can require a service identity, such as the JNDI name of the EJB home, and possibly a conversation ID, which can be extracted from a cookie, in the case of stateful EJB targets. The body of the request can indicate the operation name that will be mapped to the proper method call on the EJB.

[0042] A stateless EJB invocation handler can be used to dispatch Web service invokes to an EJB. This handler can require the JNDI name of the stateless EJB home. The handler can obtain an instance of the EJB and can dispatch the invoke and return the return value, if there is one.

[0043] A stateful session EJB invocation handler can be used to dispatch invokes to a stateful session bean. The handler can require the JNDI name of the stateful EJB home, as well as a conversation ID, which can be extracted from the message. The default approach for HTTP can be to extract the conversation ID from a cookie in the HTTP protocol handler and to put it in the invocation context under a documented name. If this default behavior is not suitable, the developer can provide a header handler that extracts the conversation ID from message headers and places the ID in the invocation context.

[0044] A stateful session bean (SFSB) invocation handler can maintain a table of mappings between a conversation ID and EJB handles. If no conversation ID is found, the stateful EJB invocation handler can create a new conversation ID, a new session bean instance, and can add its handle to the mapping table. The invoke can then be dispatched to the SFSB referenced by the handle.

[0045] A JMS invocation handler can dispatch the body of a SOAP message to a JMS destination. The handler can require the JNDI name of the destination, the JNDI name of the connection factory, and the destination type.

[0046] The configuration of protocol handlers can involve specifying the mapping between Web service endpoint URIs, such as URLs in the case of HTTP or email addresses in the case of SMTP, and the name of an invocation handler. A particular invocation handler can require additional configuration information, such as the JNDI-name of a target EJB deployment.

[0047] An HTTP protocol handler can be a special Web application that is automatically deployed when a Web archive file (WAR) is deployed. The URL mappings to invocation handlers can be extracted from the WSP ("Web service provider") description of the Web service. An HTTP protocol handler can map HTTP headers to the invocation context and can attempt to extract a conversation ID from an HTTP cookie, if one is present. An SMTP Protocol Handler can also be utilized.

[0048] An HTTP-based Web Service can be packaged in and deployed from a J2EE WAR that is contained inside a J2EE Enterprise Archive File (EAR). The WAR can contain a Web service WSP document, which defines a Web service. The WSP can describe the shape of the Web service and how the implementation maps to backend components. A WSP can be referenced in the URL of a Web service, like a JSP. It can also allow reference to user-defined tags, like a JSP which can integrate user-developed functions into the definition of the Web service. It can also support the scripting of Web service functions. Unlike a JSP, however, a WSP may not compile down to a servlet. The WSP can be directly utilized by the Web service runtime.

[0049] The remaining contents of the EAR can include EJB-JARs or other classes that are part of the implementation of the Web service.

[0050] A Web container can manage the deployment of HTTP WSPs in a similar manner to JSPs. There can be a default WSP servlet registered with each Web application that intercepts requests for WSPs. The default servlet can then redirect each request to the appropriate WSP handler.

[0051] A user can open a Web application in a console, or in a console view, and can view the names of the WSPs that are part of that Web application. It can be necessary to modify an MBean, such as WebAppComponentMBean, on order to provide a list of WSPs.

[0052] Java-based Web services client distributions can be used with services hosted on many different platforms. A full set of features supported on a client can include:

[0053] HTTP protocol with cookie support

[0054] SOAP 1.2 with attachments

[0055] JAX-RPC client API, including synchronous and "one-way" RPC invokes

[0056] Header Handler and Flow Handler API

[0057] Message-style Web service client API

[0058] Support for "dynamic mode" (no Java interfaces or WSDL required)

[0059] Support for "static mode" (Java interfaces and service stubs required)

[0060] The full set of SOAP encodings+Literal XML+ support for custom encodings

[0061] Security support:

[0062] 128-bit two-way SSL

[0063] Digital Signatures

[0064] XML Data Encryption

There is an inherent tradeoff between the "thinness" of a client and the richness of features that can be supported. To accommodate customers with differing needs regarding features and footprint, several different client runtime distributions can be offered with varying levels of features.

[0065] A J2SE Web Service client, which can have a footprint of around 1 MB, can be full-featured. SSL and JCE security functions can be included in separate jars. This client can run in regular VM environments, such as those hosting application servers. A J2ME/CDC thin client can have a limited set of features, but can be designed to run in a J2ME CDC profile on devices. A JDK 1.1 thin client can have a limited set of features, but can be intended to run in JDK 1.1 virtual machines, including those hosting applets.

[0066] Client distributions can include classes needed to invoke Web services in "dynamic" mode. Utilities can be provided to generate static stubs and Java interfaces, if given WSDL service descriptions.

[0067] A Java™ 2 Platform, Standard Edition (J2SE) Web service client can be a standard, full-featured client, which can be intended to run inside an application server. The client can be included in a regular server distribution, and can also be available in a separate jar so that it may be included in other J2EE or "fat client" JVMs. There may be no size restriction on this client. The client can utilize JDK 1.3.

[0068] FIG. **3** shows an architecture of the client-side for a J2SE Web Service client **318** in accordance with one embodiment of the present invention. The client is closely related to the Web service container. The client can be an embeddable Web service container that can run in lighter weight servers. This can allow asynchronous callbacks to be invoked on the client.

[0069] In FIG. **3**, the HTTP protocol adapter **102** of the Web container **100** is shown passing message context to, and receiving message context from, a container driver **300**. The container driver **300** can receive message context from the protocol adapter **102** and send the message context to the registered inbound interceptors **302**, **304**, **306**. After extracting performing any necessary data binding or unbinding, such as by using a Java Binding codec **310**, a SOAP codec **312**, an XML codec **314**, or a custom codec **316**, the container driver **300** can return the data to the client stub **308**. If receiving invoke data from the client stub **308**, the container driver **300** can perform any data binding or unbinding using the appropriate codecs **310**, **312**, **314**, **316** and send the invoke request to the outbound interceptors **302**, **304**, **306**. The container driver **300** can then send message context for the invoke to the protocol adapter **102**.

[0070] The foregoing description of preferred embodiments of the present invention has been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Many modifications and variations will be apparent to one of ordinary skill in the art. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application, thereby enabling others skilled in the art to understand the invention for various embodiments and with various modifications that are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalence.

What is claimed is:

1. A system for asynchronous invocation of a web service by a web services client using Java Messaging Service protocol comprising:

a web container that provides an end point for a Java Messaging Service protocol based web service;

a Java Messaging Service invocation handler that receives a web service request from the web container, wherein said web service request includes a SOAP message; and,

a Java Messaging Service destination that receives a body of the web service request's SOAP message dispatched by the invocation handler, wherein the Java Messaging Service destination provides a web service response for the web service client to retrieve at a later point of time.

2. The system of claim 1, wherein the Java Messaging Service invocation handler is in the web container.

3. The system of claim 1, wherein the Java Messaging Service destination is in the web container.

4. The system of claim 1, wherein the Java Messaging Service destination is a web service.

5. A system for invocation of a web service by a web services client using Java Messaging Service protocol comprising:

a web container that provides an end point for a Java Messaging Service protocol based web service;

a Java Messaging Service invocation handler that receives a web service request from the web container, wherein said web service request includes a SOAP message; and,

a Java Messaging Service destination that receives a body of the web service request's SOAP message dispatched by the invocation handler, wherein the Java Messaging Service destination provides a web service response for the web service client via the Java Messaging Service invocation handler and the web container.

6. The system of claim 5, wherein the Java Messaging Service invocation handler is in the web container.

7. The system of claim 5, wherein the Java Messaging Service destination is in the web container.

8. The system of claim 5, wherein the Java Messaging Service destination is a web service.

9. A web service system wherein messages to the web service use Java Messaging Service to asynchronously communicate with a service requester.

10. The web service system of claim 9, wherein the system includes a web container.

11. The system of claim 10, wherein the web container contains a Java Messaging Service invocation handler.

12. The system of claim 10, wherein, the web container contains a Java Messaging Service destination.

13. A web service system wherein messages to the web service from a service requester are asynchronous.

14. The web service system of claim 13, wherein the system uses Java Messaging Service to send the messages.

15. The web service system of claim 14, wherein the system includes a web container.

16. The web service system of claim 15, wherein the web container contains a Java Messaging Service invocation handler.

17. The web service system of claim 15, wherein, the web container contains a Java Messaging Service destination.

* * * * *