Colombo: Lightweight middleware for service-oriented computing

F. Curbera
M. J. Duftler
R. Khalaf
W. A. Nagy
N. Mukhi
S. Weerawarana

Colombo is a lightweight platform for developing, deploying, and executing service-oriented applications. It provides optimized, native runtime support for the service-oriented-computing model, as opposed to the approach of layering service-oriented applications on a legacy runtime. This approach allows Colombo to provide high runtime performance, a small footprint, and simplified application development and deployment models. The Colombo runtime natively supports the full Web Services (WS) stack, providing transactional, reliable, and secure interactions among services. It defines a multilanguage service programming model that supports, among others, Java™ and Business Process Execution Language for Web Services (BPEL4WS) service composition, and offers a deployment and discovery model fully based on declarative service descriptions (Web Service Description Language [WSDL] and WS-Policy). In this paper we describe these and other aspects of the architecture, design principles, and capabilities of the Colombo platform.

INTRODUCTION

Middleware and applications supporting Web Services specifications and standards are now being offered by every major software vendor. Solutions utilizing these technologies can be found throughout the Information Technology (IT) industry. As the framework of Web Services specifications[1,2] progresses toward consolidation and standardization, a new distributed computing paradigm is slowly being established. Web Services provide an XML (Extensible Markup Language) realization of the service-oriented computing (SOC) paradigm, and the progressive deployment of applications based on Web Services has the potential to make service-oriented architecture (SOA) the main architectural paradigm in the industry.

Service-oriented applications have specific characteristics that distinguish them from traditional distributed applications.[3] First, they must be able to operate in a natively cross-organizational setting; they interact with each other as peers, over bidirectional, stateful channels, following standard-

ized protocols that allow them to operate in highly heterogeneous environments. They declaratively define their functionality and quality-of-service (QoS) requirements and capabilities in agreed-upon, machine-readable formats in order to enable dynamic and automated service discovery. Finally, service-oriented applications are, in one way or another, created as compositions of services.[4]

The adoption of Web Services technologies does not necessarily mean that SOA principles are being embraced. At this stage, middleware interoperability; that is, the ability to support interoperation among incompatible middleware platforms, is probably the main factor driving Web Services adoption. Web Services allow Enterprise Java** and Microsoft .NET** to interact and proprietary messaging middleware to connect over bridges of standardized protocols. In this initial adoption stage, the architectural impact of Web Services is likely to be limited.

Extended interoperability contains the seed of wider changes, however. As more and more existing and new applications are Web-Services enabled, the prospect of quasi-universal interoperability becomes closer to reality, and with it, the ability to access any number of services deployed anywhere (inside or outside the enterprise). A business model built on service access and reuse (including in particular "pay per use") is a direct consequence of this increased level of interoperability. Even without assuming a significant transformation of the business model, application development is likely to be profoundly transformed by this new focus on code reuse through remote service access. Applications developed in this environment can take full advantage of Web Services composition models, dynamic service discovery, explicit interaction contracts encoded in standards-based dialects, and so forth. Moreover, the componentization brought about by SOA advances will likely bring the design of software components much closer to the business components they are intended to support, helping bridge the gap between business knowledge and IT. Thus, one may expect that, in time, service-oriented principles will naturally follow the Web Services framework in an unstoppable advance through the IT industry.

This assumption raises the question of whether the architecture and the programming model associated with today's middleware platforms is always the most appropriate for supporting applications executing in a service-oriented environment. Today's most successful Web Services platforms are fundamentally built by layering a veneer of Web Services protocols on top of existing middleware (consider, for example, Microsoft's Web Services Enhancements, WSE 2.0, and the suite of Java Specification Requests [JSRs] supporting the use of Web Services from a Java 2 Enterprise Edition [J2EE**] platform). The eventual success of this approach is not in question because it provides proven reliability and scalability with a new protocol suite, and it is also a progressive approach that ensures quick adoption with little disruption of existing projects and applications. Nonetheless, the question is whether this layering model is always appropriate as a long-term approach to service orientation.

The Colombo project at IBM Research is an attempt to understand the consequences of the service-oriented paradigm on the programming model and the runtime and the type of competitive advantage that a platform based natively on SOC principles can provide in terms of simplicity, performance, and developer productivity. With the Colombo approach, an experimental SOA platform is built from first principles, SOC principles, and its characteristics are evaluated with respect to performance, scalability, the process of development and deployment, application management and maintenance, and other aspects.

The Colombo project is a Web-Services-centric, lightweight approach to service orientation. Web Services are, after all, the only realization of SOC with a sufficient level of adoption to support realistic field tests and comparisons. The Colombo "platform" is still under development, but it already provides a very lightweight implementation of most of the Web Services (WS) stack, with the notable exception of service and meta-data discovery specifications. Thus, the Colombo runtime natively provides transactional, reliable, and secure interactions among services based on SOAP (Simple Object Access Protocol).[5] Colombo defines a multi-language service programming model that supports, among others, Java and Business Process Execution Language for Web Services (BPEL4WS) service composition, and offers a radically simplified development and deployment model based on

standards (Web Services Description Language [WSDL][3] and WS-Policy[6]).

In this paper we describe the design principles behind Colombo, focusing on its service-centric programming model (the "servicelet" model) and the architecture of its runtime. The rest of this paper is organized as follows. The next section briefly describes the principles of SOAs that have driven the design of the Colombo runtime and programming model. The third section introduces the Colombo programming model in detail, including deployment, development models, and the realization of that model in Java and BPEL. The fourth section describes the Colombo runtime, focusing on the Colombo message-processing engine, the component manager architecture that supports the servicelet multilanguage programming model, and the QoS policy framework. We conclude in the fifth and sixth sections with a discussion of future work and a summary of the paper.

SOA PRINCIPLES AND REQUIREMENTS IN COLOMBO

The Colombo project is directed at exploring the technical characteristics that differentiate SOA middleware. There are in fact many distinguishing aspects of SOAs that might be expected to have an effect on the design of SOA-centric middleware. Because the Web Services specification framework defines the only protocol stack built on SOA principles to date, the Colombo effort has focused on providing a faithful realization of the Web Services stack. The discussion in this section in particular is thus very much derived from the key characteristics of the Web Services stack.

The main components of the Web Services framework are depicted in ***Figure 1***. (See Reference 3 for an in-depth discussion of these components.)

The Colombo platform described in this paper supports the complete stack shown here except for the mechanisms for discovery of services and negotiation. This implies in particular that there is no support in Colombo for Universal Description, Discovery, and Integration (UDDI)[7] or WS-MetadataExchange.[8]

As in all middleware architectures, the interaction model, runtime architecture, and programming model are intimately connected in SOAs. The

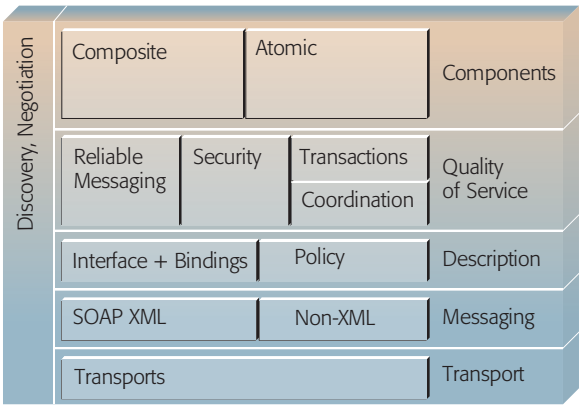

Figure 1
The Web Services stack

interaction model in Web Services is built around the SOAP messaging model, and this has two main implications: the centrality of XML as the data serialization model and a natively asynchronous interaction model that can also support synchronous interactions. The WS-Addressing specification[9] in particular introduces a set of SOAP headers and a simple processing model to seamlessly support both synchronous and asynchronous exchanges of SOAP messages. The runtime architecture in Colombo is thus built around an efficient XML parser supporting a SOAP processor that can deal with synchronous and asynchronous message exchanges.

A second characteristic of Web Services interactions is the central role of QoS requirements. In an SOA, QoS requirements are explicitly stated in machine-readable format and become part of the interaction contract between providers and requestors. QoS requirements are used to configure the interaction channel between service partners, and they are encoded in the form of explicit service policies, in the case of Colombo using the WS-Policy language. WSDL service descriptions published and consumed by Colombo applications are annotated with policies stating those requirements. Correspondingly, the Colombo runtime provides a policy handler framework on top of the message-processing engine, supporting the three QoS specifications currently in the Web Services stack: atomic transactions (WS-AtomicTransactions[10]), reliable messaging (WS-ReliableMessaging[11]), and the suite of Web Services security mechanisms defined by the Web Services security standard.[12]

Explicit meta-data in the form of machine-readable service descriptions, including functional and non-functional QoS characteristics, is a central aspect of Web Services.[13] The Colombo development and deployment models are completely driven by this meta-data. Code generation, deployment, and runtime configuration are all driven by the WSDL and WS-Policy descriptions of the services being authored or consumed. The goal is to avoid generating a gap between the internal representation of service capabilities and the external, interoperable service view that is defined by the service contract.

SOAs in general and Web Services in particular have a distinct component orientation.[4,14] Services are software components that allow remote access over standard protocols and provide declarative descriptions of their requirements and capabilities. In this component-centric environment, the main task of the application developer is the integration of service components into new applications; that is, service composition is the distinctive characteristic of development in SOAs. The programming model for SOA middleware needs to provide native support for service creation and composition. The Colombo programming model defines primitives for providing and reusing services in a language-independent manner, which allows Colombo to support a variety of composition models. Java and BPEL[15] are the two languages currently supported for composing services.

We can summarize the preceding discussion by enumerating the main aspects of the service-oriented model that drives the design of Colombo:

- Native support for a SOAP asynchronous interaction model
- Policy-enabled interactions supporting the business-enabling QoS requirements: transactions, reliable messaging, and security
- Meta-data-driven development and deployment models
- First-class support for service composition

The details of how the Colombo programming model and runtime support these requirements are discussed in the following sections.

THE COLOMBO PROGRAMMING MODEL

In this section, we discuss the Colombo programming model. Given that the term "programming model" can have many meanings, we begin by defining what we mean by a programming model. A *programming model* is the set of abstractions, conventions, and contracts to which the programmer must adhere when writing applications. It also includes the set of services that the system provides to the programmer.

The SOA programming model

A distinguishing characteristic of service-oriented applications is that of composition: services are often built by taking existing services and combining them with some compositional logic. As such, we identify four critical characteristics of a minimal SOA programming model. There must be a mechanism

1. to access other services,
2. to encode the compositional logic,
3. to encapsulate the composition as a new service, and
4. to state the QoS characteristics that should be followed when interacting with other services.

We illustrate this model in ***Figure 2***. As indicated in the figure, the composition logic is the centerpiece and serves to orchestrate, and add value to, the functionality provided by the "Used" services to offer a set of "Provided" services to its clients. The terms and conditions under which the services are used are indicated by policies associated with interactions between the composition and external services, whether they are used or provided services. Policies are declarations of expected or offered QoS characteristics, as described later. Note that interactions between the composition logic and "built-in" or "system" services are not special—they too are modeled as service interactions.

The servicelet programming model

In keeping with the simplicity objectives of Colombo, the programming model was designed to be as simple as possible yet powerful enough to write service-oriented applications. In Colombo, we define "service" as follows: A *service* is a stateless message processor supporting exactly one interface and with associated interaction-specific information, said to be "context." Thus while the service logic itself is stateless, it does have access to a context that may contain state associated with a particular interaction between the service and a client or partner service.

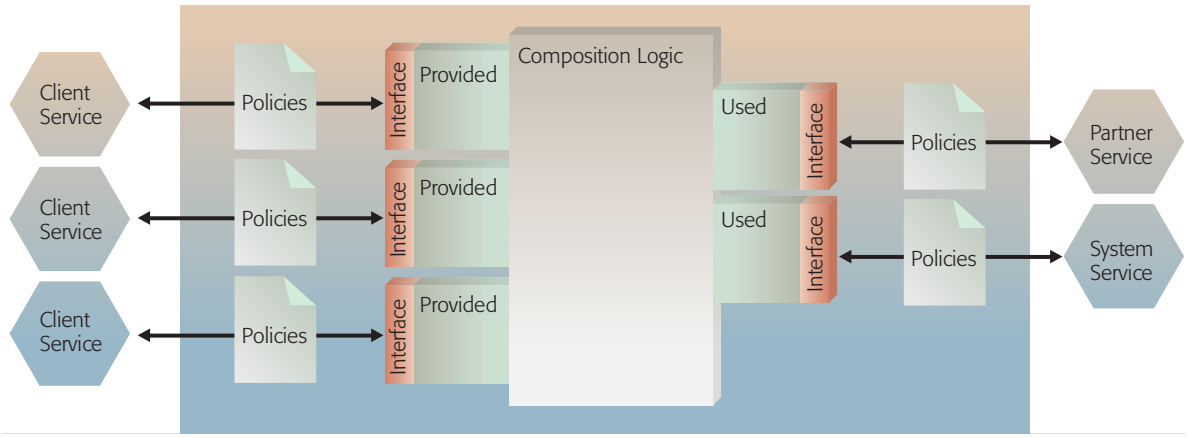

Figure 2
SOA applications as service compositions

The unit of development and deployment in Colombo is called a *servicelet*. As ***Figure 3*** shows, the servicelet concept is a direct mapping of the SOA programming model illustrated in Figure 2 with state represented as the servicelet context.

A servicelet provides one or more services and may use more services to implement the provided services' logic. We call the implementation of the provided services "business logic." Although the business logic itself is stateless, each interaction between any service of the servicelet and a client or partner results in the creation of a servicelet context associated with, and shared by, the entire servicelet. Interactions between the servicelet and other services (whether they are with a partner service by means of a stub or with a client service by means of an adapter) represent conversations with their own policies as well as a conversation context local to that conversation.

We model interactions between an application servicelet and the Colombo runtime also as service interactions. This uniform approach for all inter-

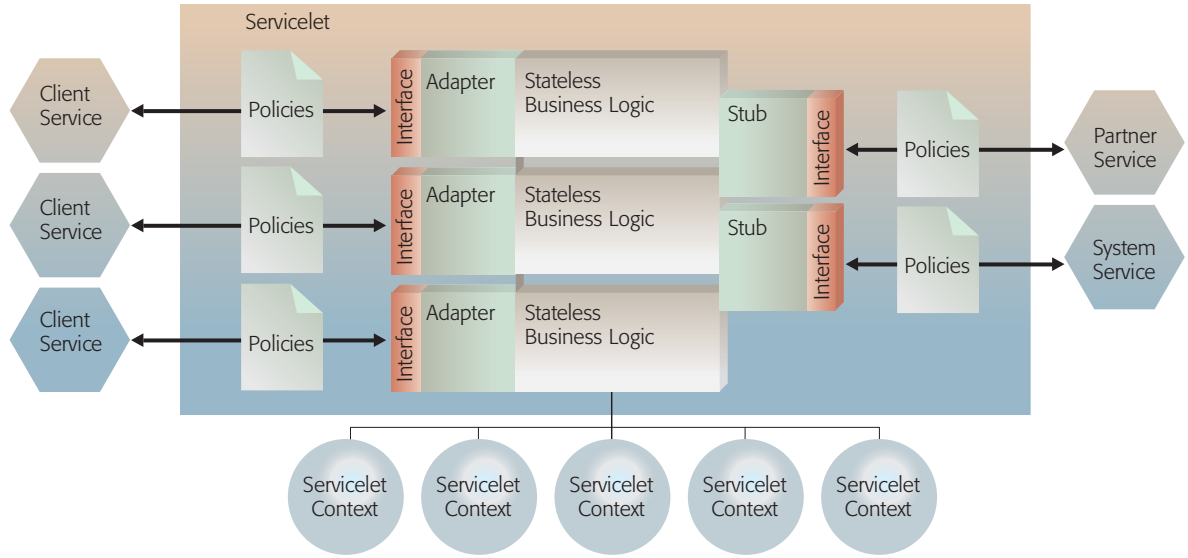

Figure 3
Servicelets

action with a servicelet simplifies the programming model by not forcing the programmer to learn two different sets of abstractions.

All the services in a single servicelet share a common servicelet context for each "instance" of the servicelet. However, we note that with our definition of service, there is really no concept of a service instance. Instead, there are stateful conversations with a service. Thus, every usage or sequence of usages (a session) with a service results in one stateful conversation. A stateful conversation with any of the services of the servicelet results in the creation of a single servicelet context that is shared by the entire servicelet. Thus, in effect, the servicelet context instance represents the "instance" of the servicelet.

Servicelet context data is not persisted by the runtime system. The decision in Colombo that the system would not provide any kind of automatic persistence was carefully considered, as our experience indicated that such automatic persistence comes at a heavy price with questionable benefit. Thus, if any application-sensitive data is stored in the servicelet context, the application is responsible for treating such data only as a cache and for storing it persistently by using the system data service.

Thus, a servicelet can migrate in between invocations to a different location by migrating the content of the servicelet context.

When implementing the actual business logic, programmers are offered a set of system services that they can rely on (the system services currently provide access to stubs for invoking services, creating user-managed transactions, access to the data service, and logging). As these services are modeled as Web services, the business logic can be implemented with any programming paradigm that supports Web-service interactions. Thus, we do not make any assumption about how the servicelet business logic itself is implemented; that is, the servicelet programming model is designed to allow a servicelet to be implemented as a single Java class, a collection of Java classes, a BPEL script, a collection of XSLT scripts, and so forth. The servicelet programming model is intended to be an abstract programming model that can be mapped to specific implementation approaches for servicelets.

Stubs and adapters

In Figure 3 we refer to stubs and adapters. When the service is invoking another service (i.e., when it acts as a client to another service), we say that the interaction is through a stub. This is a logical concept; that is, there is no requirement that all service interactions be through a statically type-mapped interface (as is required in JAX-RPC[16]). The stub exists to bridge between the business logic and the Colombo runtime, which is actually responsible for delivering the message to the called service.

The adapter concept is similarly logical. An adapter is simply the bridge between the incoming message and the actual business logic. Depending on the implementation platform of the business logic, the stub may do anything, ranging from nothing to completely static type mapping.

Servicelet life cycle

The life cycle of servicelets is defined by three states: initialize, process, and destroy. The servicelet is initialized as the first message arrives to the servicelet, or equivalently, when the servicelet context is created. This initialization step allows the servicelet to cache any useful information into the servicelet context, for example.

After initialization, when messages arrive via any of the published service interfaces, they are delivered to the servicelet, along with the appropriate servicelet context, for processing. The servicelet context associated with a message is determined by using information in the incoming message (e.g., using WS-Addressing reference properties).

During the execution of the servicelet, a transient state may be stored in the servicelet context. However, any persistent state must explicitly be stored by the application by using the system data service.

When the servicelet is deemed to be complete or finished, it is destroyed by destroying the associated servicelet context. Note that due to the difficulty of identifying when a servicelet is "finished" in general, there is no assurance that a servicelet context will ever be destroyed.

Conversations

A *conversation* is an interaction between a servicelet and its client or partner services. Note that by

definition every servicelet instance participates in at least one conversation with the sender of the message that started the interaction. Each servicelet "instance" may hold only a single conversation with any given partner at a time (a property of the Colombo programming model inherited from BPEL4WS).

The conversation concept is abstracted in Colombo because of the need to compute QoS parameters on a per-conversation basis. That is, at the start of a conversation the policy that will apply to the conversation is determined, possibly through a negotiation protocol (such as WS-MetadataExchange), and then that policy is used for the duration of that conversation.

Conversations are characterized by a pair of communicating endpoints and have context associated with the conversation. (In fact the conversation policy is part of that context.) However, the programming model does not expose this conversation context to the application programmer. The conversation context is maintained by the runtime, and the application affects the behavior of the conversation by asserting specific policies.

Policies

As just mentioned, all aspects of a conversation are governed by policies, including whether the conversation can take place at all. Colombo's policy infrastructure uses the WS-Policy family of specifications.

Policies can be applied at different levels. When a Colombo system is deployed, system-wide policies may be defined. When a servicelet is deployed, servicelet-wide policies may be defined as well. Similarly each service of a servicelet may have its own policies.

The policies that are applicable to a particular conversation are determined at runtime by combining all the system-wide policies, the servicelet-wide policies, and service-specific policies. This computation is performed at runtime, at the start of a conversation, as explained in the previous section. Policy granularity and calculation of effective policy is explained in detail in the section "Policy framework."

It is also useful to mention that the enforcement of certain policies applied to a conversation may not necessarily be reflected in the actual messages exchanged. In particular, privacy is an example of a policy that does not have an on-the-wire (explicit message) representation.

The system services currently provide access to stubs for invoking services, creating user-managed transactions, getting access to the data service, and logging.

Servicelet packaging and deployment

Before a servicelet can be deployed into a Colombo system, the constituent parts of the servicelet must be assembled and packaged together. Colombo uses the familiar JAR (Java archive) file mechanism as a packaging mechanism, yet does not force the use of it; that is, Columbo expects that its defined conventions have been followed rather than that everything has actually been deployed as a single JAR.

The base servicelet package has the following structure:

```
example.car/
  META-INF/
    servicelet.xml
    policies.xml
    *.wsdl for service definition
    certificates.ks
  classes/
    object and source files
  lib/
    shared library files
```

The basic objective of this structure is to simplify the developer experience by providing a relatively familiar structure (something similar to the Java Web application archive format) and one that is intuitively explainable. In order to deploy a servicelet one clearly needs some meta-data—the description of the servicelet itself (e.g., which code artifacts constitute the implementation of the business logic), the policies that must be applied, and the WSDL descriptions of the interfaces of the servicelet. If there are security aspects, then a key store is needed to record the keys.

In addition to this meta-data, one also needs to deploy the code artifacts located in `classes` and `lib`. The classification of code artifacts between `classes` and `lib` is a logical one. For example, if the service were implemented in PHP,[17] then the PHP

```
<servicelet name="servicelet://localhost/path-to-ser
    <provides>
        <service name="xs:NCName" interface="xs:QName">
            <policies> ... </policies>?
            ... Extensibility; e.g., java:implementation...
        </service>+
    </provides>
    <requires>
        <service name="xs:NCName" interface="xs:QName">
            <locator type="static|mobility|...">
                EPR or whatever
            </locator>
            ... Extensibility; e.g. java:stub...
        </service>*
    </requires>?
    <policies> ... </policies>?
    ... Extensibility; e.g., servicelet-wide impl like BPEL ...
</servicelet>
```

Figure 4
A servicelet.xml file

files would be inside the `classes` directory, even though strictly speaking PHP files are source and not compiled classes. The `lib` directory exists to contain dependent libraries.

Note that Colombo does not require that every one of these parts be present. The objective is to require absolutely minimal information and to expect the runtime to deduce, derive, or compute whatever is possible to be inferred. Furthermore, the author is allowed to deploy the packaged file or the complete directory structure; thus allowing developers to run the system directly from their working environment without having to copy any files.

The servicelet descriptor itself is also designed to be intuitively obvious once the developer understands the servicelet programming model concept. The structure of the `servicelet.xml` file is given in *Figure 4*.

Every servicelet has a name provided by the application developer. The name is used to generate the uniform resource identifier (URI) where the Colombo runtime makes the servicelet available by means of some access protocol (such as HTTP). For example, if the servicelet is published by the Colombo runtime by means of HTTP, then the URI address of the servicelet is `http://hostname/serviceletname`. Thus, the name represents a transport-independent name for the entire servicelet.

For each service that the servicelet publishes, there is a single ⟨`provides`⟩ element describing that service. At a minimum, each service has a name (which can be concatenated to the end of the servicelet name URI to get the full address of the specific service) and some way to indicate how the service is implemented. The specific implementation technology used will select exactly how the code is executed (see the section "Runtime architecture").

Servicelet development

Typical development in Colombo begins with a WSDL document (Colombo supports only the WSDL document/literal style, believing this is the direction in which the industry is headed and this is the most consistent model from a Web Services architecture perspective). Data types are described by using XML Schema, and WSDL port types represent service interfaces.

Colombo tools generate mappings from XML Schema data types into Java data types, generating custom classes for reading, writing, and manipulating the data from the service implementation (Colombo "structs") to enable future support of alternative data definition languages (such as RELAXNG schema language for XML for example). Colombo does not strictly follow any of the existing conventions for mapping Java and XML Schema types because strictly following any of the conventions would significantly detract from the simplicity of the mapping of Colombo "structs" to Java types. The serialization and deserialization code is built on top of a pull parser[18] interface.

In addition to mapping data types, Colombo tools generate code artifacts that allow servicelets to provide or to consume Web Services interfaces (corresponding to the required and provided interfaces of the deployment descriptor). To provide a Web Services interface, a skeleton and an adapter are generated out of the WSDL port type: the skeleton defines the Java interface that must be implemented by the service author, and the adapter is the class that coordinates all the reading or writing of the message payload (not including the transport and protocol headers) and contains logic to invoke the desired method on the skeleton.

To invoke a service a stub is generated from the WSDL port type. The stub is similar to the adapter in that it coordinates the reading or writing of the payload; however, instead of invoking methods

directly on an interface, the stub makes use of a servicelet manager to make the calls.

Policies may be attached to the binding sections of the WSDL documents for required and provided services, indicating QoS policies that are required for the interactions. At deployment, these policies are extracted and factored into the computation of the effective policy of the servicelet. (See the section "Policy framework" for additional details.)

Servicelets in Java

Servicelets defined in Java currently support only one provided system service and may support one or more required services. A servicelet implementation consists of regular Java code, usually implementing a generated skeleton. The skeleton may provide access to the payload of incoming messages in one of three possible ways: (1) raw incoming XML stream as pull parser events, (2) XML information set (infoset) preserving a form such as the document object model (DOM), or (3) statically mapped Java data types. Whatever the choice, a corresponding generated adapter provides the link between the servicelet implementation and the rest of the platform. The return value of each method constitutes the response to the operation.

Because the servicelet programming model itself is stateless, every method in a servicelet implementation takes a servicelet context as input in addition to the payload of the incoming message. By the time a call arrives at the implementation code, the system has already looked up its servicelet context and deserialized the payload into whatever format is required by the skeleton. The servicelet context has access to any instance-specific data values. In addition, it looks up the stubs of both required services and system services. The servicelet implementation can then simply invoke these services by calling the corresponding Java operations directly on the stubs.

Consider a servicelet that offers an address book service address by means of an `addAddress` operation. Implementation of this operation invokes another service that manages a directory of phone numbers. The corresponding code snippet is shown in *Figure 5*.

The endpoint of the invoked service is specific to the context passed in. Therefore, different calls of this method (by different servicelet instances) may send the phone message to different implementations of the phone service, with different policies applied. We will discuss endpoint management, policies, and routing in subsequent sections of this paper.

A Java servicelet's deployment descriptor extends the basic one described earlier. It adds the Java class containing the implementation and the class containing the adapter to the provided service. It adds the Java class containing the stub to each required service. Note that the adapter is not relevant to the developer.

Servicelets in BPEL

Web Services can be authored using the Business Process Execution Language for Web Services (BPEL4WS, to be renamed WS-BPEL by the OASIS e-business standards consortium, and usually referred to as "BPEL"). A BPEL process contains a set of "partnerLinks" that model bidirectional conversations between the process and other services. A partnerLink can specify two WSDL portTypes, the first provided by the process and a second provided by the partner service. The business process is encoded by providing control logic that uses a set of primitive activities for message exchange, data manipulation, and so forth.

A BPEL process embodies some of the prominent characteristics of servicelets: it is inherently conversational due to partnerLinks, and its composition model is based purely on portTypes instead of actual physical endpoints. It therefore allows for specialization of particular instances through the characteristics (policies, endpoints) of their conversations.

The "implementation" of the servicelet is the BPEL process itself, and its runtime consists of a BPEL processing engine. The state of a BPEL servicelet instance consists of the values of its variables and the state of the process navigation. The conversations of a BPEL servicelet are defined by partnerLinks. A one-sided partnerLink becomes an inbound-only or outbound-only conversation, and a two-sided partnerLink becomes an inbound-and-outbound conversation. The portTypes provided by the process are all those whose partnerLinks have an inbound component. Conversely, the services used by the process are all those whose partnerLinks have an outbound component. As in the Java case,

```
public void addAddress(ComponentContext myContext,
                       NameAndAddress nameAddr) {
    …
    PhoneNumberServiceStub  pns = (PhoneNumberServiceStub)myContext.lookup("phoneService");
    PhoneAdditionInput pai = new PhoneAdditionInput ( ) ;
    pai.setName(nameAddr.getName( ) );
    pai.setNumber(nameAddr.getNumber ( ) );
    PhoneAdditionReturn par = pns.addPhoneNumber(pai);
    …
}
```

Figure 5
Address book service code

the outbound conversations must be provided with an endpoint reference before they can be invoked at runtime. There are multiple ways in which this can occur: (1) at deployment time, to be used by each created servicelet instance as a default endpoint; (2) on an instance-by-instance basis at some later time either by means of a `ReplyTo` in an incoming message or by means of the explicit copying of an endpoint in the BPEL process through the BPEL assign activity; or (3) a combination of the two in which the default endpoint for each outbound conversation is used as long as it is not overwritten in a particular instance using (2).

A BPEL process instance uses "correlation sets" to allow message exchange activities to correlate messages sent and received by specific servicelet instances. These are fields of application data present in the messages that maintain constant values during the interaction with a particular servicelet instance. This property allows the BPEL runtime to use those correlation set values for message routing. To clarify the interplay of correlation-based routing and WS-Addressing-based routing (using endpoint reference data), a set of precedence rules must be introduced. In Colombo, correlation set routing is used only if middleware-based routing is unable to find an instance. Correlation set values are verified and updated regardless of the mechanism used to route a message, because the BPEL specification requires that an application-level error be signaled if the instance found using middleware-based routing has different correlation values.

To summarize, a BPEL servicelet consists of multiple conversations that provide and consume a number of services, a single implementation consisting of the BPEL process definition, and any endpoints and policies for these conversations.

RUNTIME ARCHITECTURE

The Colombo runtime architecture consists of three major components: a SOAP-based message-processing engine, a set of servicelet managers, and a policy framework. The message-processing engine is the backbone of the Colombo runtime, providing the means by which data is transferred to and from servicelets and QoS policy is enforced. The servicelet managers connect the Colombo message engine to the individual servicelets, helping isolate the servicelet programming model from implementation artifacts and transport details. Finally, the policy framework is responsible for managing and enforcing QoS characteristics for services running inside of Colombo. The following subsections describe each of these components in more detail.

The message-processing engine

Like most other middleware platforms, Colombo is at its core a message processing engine. Messages flow into the system via a communications channel, and are examined and dispatched to the intended recipient according to a set of predefined rules.

Interactions are discussed in this section from the point of reference of the servicelet. The term "request" refers to a request directed to a Colombo-hosted service (a servicelet). The term "response" refers to the message sent by a servicelet to answer a request. The term "invocation" indicates a message (other than a response) sent by a servicelet to another service, while "invocation response" refers to the response message coming back to the servicelet, which is triggered by the invocation

message (we assume here the operation invoked was a request/response operation).

The term "channel," unless otherwise noted, refers to the messaging protocol and transport used to transmit Web Services messages, for example SOAP or HTTP. In the case of Colombo, there is an assumption that SOAP defines the messaging and serialization format, whereas typically HTTP carries the XML envelopes. Colombo allows replacement of the underlying communication protocol (replacing HTTP by TCP/IP for example), although the only communication protocol currently available is HTTP.

"Message context" refers to contextual information associated with a specific message, such as the originator of the message, and is not to be confused with the servicelet context that was mentioned earlier, although both are used during the processing of a message.

Colombo's message-processing pipeline implements a one-way messaging model. While Web services, in general, support other types of interaction patterns, such as synchronous or asynchronous request/response, the Colombo architecture does not assume any particular message exchange pattern (MEP). (See References 3 and 5 for typical Web-service exchange patterns.) On top of the one-way core message-processing engine, Colombo uses WS-Addressing[9] mechanisms to build support for the common request/response MEP. Other MEPs can be built on top of this framework once the appropriate support mechanisms (message headers typically) are provided.

Colombo partitions all exchanged messages into two sets and creates a separate execution path for each one of them. The first execution path processes incoming messages, including requests and invocation responses, while the second processes outgoing messages: responses and invocations. This factorization is sufficiently generic to potentially enable the execution of arbitrary MEPs, and supports the requirements of generic message-based QoS protocols, such as atomic transactions, reliable messaging, or security. The result is to decouple the programming-model-level decision of which MEPs are more suitable when using a particular programming language for a particular application from the architecture of the runtime message processor. The same benefit applies when we consider specific QoS protocols instead of MEPs.

As a result, the programming model exposed by each servicelet realization (e.g., Java or BPEL) remains consistent across communication mechanisms. For example, the current invocation pattern supported by Java servicelets fits naturally with the synchronous request/response characteristics of HTTP, but works consistently when asynchronous responses are demanded (according to the use of the `ReplyTo` header in WS-Addressing, for example). BPEL servicelets are able to support both synchronous and asynchronous request/response operations and can take explicit advantage of the flexibility of the platform (both are also, of course, able to perform one-way invocations, which is not natural to HTTP). The more complex interaction patterns are layered on top of the one-way messaging model, and the issues involved are isolated in the channels and in the servicelet managers. The channels are responsible for managing the details of the transport and protocol; for example the SOAP/HTTP channel is responsible for managing the fact that HTTP is a synchronous request/response communication protocol.

Inbound message processing

Figure 6 shows the message flow into a Colombo servicelet and represents the executing path followed when a new request or an invocation response is received. When one of these messages is received, (1) the incoming connection is handed off to an available worker thread or queued up if none is available. When a worker is available, the transport headers (e.g. HTTP headers) are stripped from the stream and processed and (2) a new instance of the XML parser representing the SOAP message is handed over to the SOAP header processor (the parser instance is passed along the message-processing path because it contains and represents the SOAP envelope). The header information is extracted and stored (3) into a message context structure by using preregistered object mappings. By mapping SOAP headers to Java objects in advance, each of the consumers of the header information is directly presented with this information in a well-defined, communication-protocol-independent format.

The SOAP message body is not read until the message is actually delivered to the servicelet, thus

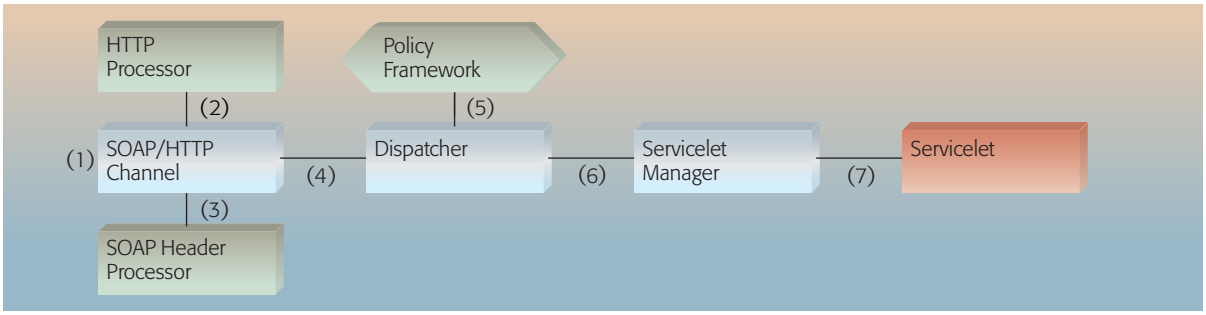

Figure 6
Inbound message flow in Colombo

avoiding unnecessary parsing and data mapping in cases when the message is not delivered to the application because of the action of intermediate QoS protocol handlers. In some instances, however, it is not possible to delay the complete parsing of the message because complete information is demanded by an intermediate policy processor, such as when digital signatures are validated.

After SOAP headers are extracted, the message is passed off to a dispatcher module (4) where routing and processing information is retrieved from WS-Addressing message information headers. The information retrieved from these headers is used to identify the target service instance and to retrieve the policy configuration that applies to the conversation.

Once the target service has been identified and the conversation details have been accessed, configured policies are enforced. The policy handlers (described in the subsection "Policy enforcement") may approve the delivery of a message, reject the delivery, or defer further processing. For example a reliable messaging handler may delay delivery of a message if earlier messages in the sequence have not yet been received. Policy handlers generally operate solely on the message context provided. If the message adheres to all of the applicable policies, it is delivered to the appropriate servicelet manager (6) for dispatching to the actual servicelet implementation and instance (7).

The servicelet manager may at this point decide that the connection to a channel is no longer required, based on the MEP within which the message has been exchanged; for example, if the operation is a one-way MEP or a response has already been transmitted. In this case the manager notifies the channel, which may decide whether to close down the outstanding connection or to continue processing further messages, such as in the case of a persistent HTTP connection.

Outbound message processing

Outgoing messages, responses (including most fault messages), and service invocations follow a similar, but reversed, path. This is shown in *Figure 7*.

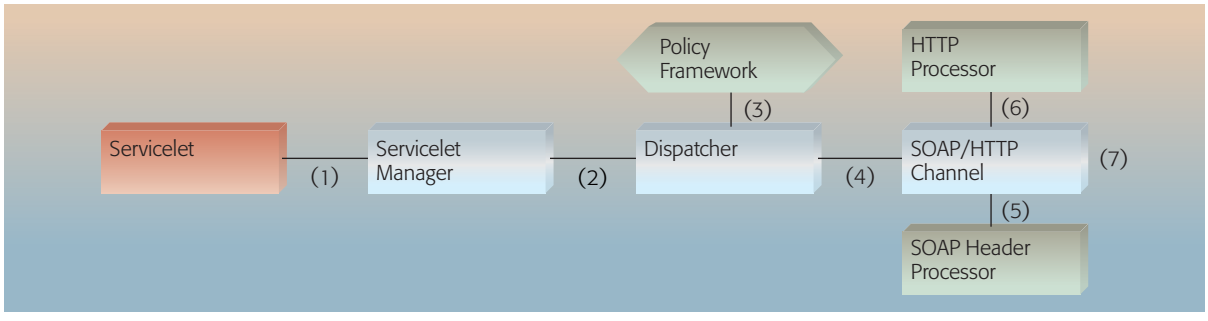

Figure 7
Outbound message flow in Colombo

Invocations and responses originating in a servicelet instance (1) are passed on to the outbound message pipeline by the servicelet manager (2). As with incoming messages, outgoing messages have message context associated with them, some of which may be derived from previously received messages, as is the case with responses, or from the configuration and state pertaining to the conversation on which the message was exchanged. Outbound messages undergo policy-enforcement-routine processing as defined by the channel configuration (3); just as in the incoming case, the policy handlers may interrupt the processing if appropriate. If the message passes inspection, it is sent over the appropriate channel for transmission (4). Information contained within the message context is written on the message (5); then the channel serializes and transmits the message (6 and 7).

Colombo support for WS-Addressing

Many of the difficulties that arose during development of the Colombo runtime centered on issues regarding the WS-Addressing specification.[9] Here we describe how Colombo supports WS-Addressing and the impact of this specification upon message routing. WS-Addressing defines a set of message information headers (SOAP headers) that provide message routing and correlation information. *Figure 8* shows a sample request message with WS-Addressing headers in a SOAP envelope (with an empty SOAP body).

For each request, Colombo requires the WS-Addressing `To` and `Action` headers to be present. As was mentioned earlier, Colombo uses WS-Addressing headers to identify the target of an incoming message. The `To` header is used to route the message to the appropriate service and hence to the correct servicelet. The `Action` header, in turn, identifies the operation being invoked on that service, and thus identifies the message exchange pattern (i.e., it indicates whether the message is one-way or part of a request/response exchange.) In addition, a subset of the SOAP headers present in the inbound message might correspond to WS-Addressing "reference properties" and is used to direct the message to a particular servicelet instance (or servicelet context instance).

Reference properties are specific SOAP headers that have been previously communicated to the service requestor with the endpoint reference of the target service instance (possibly sent in the `ReplyTo` field of an outbound message), and that the requestor must place on new messages directed to the servicelet instance. Colombo uses a single, well-defined, reference property (the "Colombo instance ID") to identify instances, but its value is otherwise opaque to requestors. The values of the headers of the `To` and reference properties are sufficient to identify the servicelet conversation and retrieve the corresponding configuration parameters (such as negotiated policy settings) or conversation state (recall that each servicelet instance can maintain only one simultaneous conversation with each service partner.)

Request/response operations, as indicated in the WSDL document for the service, also require that the `ReplyTo` header be present. The `ReplyTo` header indicates where the response message should be directed. A specific "anonymous" URI defined by the WS-Addressing specification is used to indicate that the delivery mechanism for the reply message was provided out-of-band. In particular, this covers the case of synchronous request/response interactions in which the response is delivered through the open connection as part of the HTTP response. Observe that the semantics of the `ReplyTo` header in WS-Addressing requires the runtime to dynamically determine, based on the inspection of the `ReplyTo` header, whether a response needs to be returned synchronously over the open HTTP channel or whether a new connection needs to be created to deliver that response to a different address. Through manipulation of the headers, WS-Addressing allows the service requestor to force an asynchronous message exchange over an otherwise synchronous communications transport.

When a request for a request/response operation is received by the Colombo message processor with an "anonymous" `ReplyTo`, the HTTP channel is kept open until the operation returns and the response message is sent back as part of the HTTP synchronous response. If, however, any other URI is present in the `ReplyTo` field, the HTTP connection is closed and a `200 OK` returned. Next, a new HTTP connection to the address specified in the `ReplyTo` is opened and used to deliver the response message returned by the servicelet, resulting in the asynchronous delivery of the response.

The use of `ReplyTo` in WS-Addressing is not strictly limited to request/response MEPs. When received as

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
          "http://www.w3.org/2003/05/soap-envelope"
          xmlns:WS-ADDR=
          "http://schemas.xmlsoap.org/ws/2004/03/addressing"
          xmlns:COLOMBO=
          "http://w3.opensource.ibm.com/projects/colombo">
    <SOAP-ENV:Header>
        <WS-ADDR:MessageID>
        uid:221ae5be:101c9b82fc7:-7ffb
        </addr:MessageID>
        <WS-ADDR:To>
       http://localhost:4321/http://www.research.ibm.com/shoppingCart
        </WS-ADDR:To>
        <WS-ADDR:Action>
       colombo:///shoppingCart_service/finishWithCartRequest
        </WS-ADDR:Action>
        <COLOMBO:InstanceID>
        uid:221ae5be:101c9b82fc7:-7fff</COLOMBO:InstanceID>
        <WS-ADDR:ReplyTo>
          <WS-ADDR:Address>
http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous
          </WS-ADDR:Address>
        </WS-ADDR:ReplyTo>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body />
</SOAP-ENV:Envelope>example.car/
```

Figure 8
Sample request message

part of an invocation response or a request (except in the case of the "anonymous" URI), the Colombo message engine uses the `ReplyTo` field to bind, for the duration of an extended message exchange, the endpoint address of a service partner. This is particularly relevant in the case of the interaction between a BPEL process instance and a service partner. The scope of the `ReplyTo` header is interpreted in this case more broadly and, in effect, provides an in-band mechanism to support conversation migration (because a new value of the `ReplyTo` address may be communicated with any incoming message). All future invocations to that partner will then be delivered using the new URI.

Colombo supports all WS-Addressing headers as defined in the WS-Addressing specification (`FaultTo` to direct faults, `MessageId` for message correlation, etc.), but we will not describe their operation here. Refer to the WS-Addressing specification for details on the semantics of these additional headers.[9]

Note that Colombo currently assumes that all of the URIs specified by means of WS-Addressing mechanisms are network routable (e.g., when performing an invocation, the `To` header contains enough information for the connection to be opened), although it would not be difficult to add a layer of indirection for URI resolution. However this may make interoperability with other systems more difficult.

The servicelet manager

A servicelet manager provides the connective layer between servicelet implementations and the rest of the system's infrastructure. Different implementation runtimes have different requirements and formats. Therefore, having different servicelet managers for each kind of servicelet enables the reuse of most of the common SOA infrastructure while still allowing specialization for different runtimes. Examples of servicelet types requiring different servicelet managers include Java servicelets and BPEL servicelets. In order to add support for additional servicelet types to the platform, a new servicelet manager that hooks the new servicelets and their runtimes into the rest of the Colombo machinery can be added. At the very least, a servicelet manager interfaces with the SOAP processing machinery and the policy handlers, serializes and deserializes message payloads into and out of the correct format, and manages instances (creation, destruction, and routing).

In cases such as Java, where the language itself does not have mechanisms to represent conversations, the developer needs programmatical access to partner representations and a language-specific interface that complies with the WSDL portType being implemented. To provide this capability and shield the developer's code from dependencies on the servicelet manager, stubs, skeletons, and adapters are generated. The stubs and skeletons provide the layer of abstraction for the developer code. The stubs and adapters also provide the glue-code that interfaces with the servicelet manager. These are then used by the servicelet manager to handle serialization and deserialization of data types.

The invocation sequence in the case of Java servicelets is as follows: the manager passes the servicelet context and the parser to the generated adapter, which uses a "read" method defined in the generated target Java class to parse the remainder of the message. The appropriate method is then invoked on the skeleton, which contains the business logic for the operation. Note that the adapter is stateless and does not maintain a reference to the skeleton. There is one adapter instance per adapter class, and each skeleton reference is passed to the adapter, along with the context and parser, at the time of invocation. After the skeleton method invocation is completed, the return value is written out by using the typed XML serializer.

BPEL, on the other hand, is natively conversational. It provides a representation for conversations based on WSDL portTypes (partnerLinks), as well as activities to interact with partners along these conversations. Therefore, there are no stubs and skeletons to expose to the BPEL servicelet developer. His or her interaction will always be strictly with standardized XML formats (BPEL, WSDL, XPath, XML Schema), especially BPEL activities. The BPEL servicelet manager therefore has the choice of whether to use static stubs and adapters internally to aid in reading or writing message payloads or to do the serialization and deserialization dynamically itself. The latter option is currently chosen in Colombo.

Handling service invocations

As noted earlier, the mechanism by which a servicelet invokes one of its required services depends on its implementation language, that is, either a call to a stub's operation in Java or through the activation of an invocation activity in BPEL, or through whatever other mechanism a newly supported implementation language would provide. Whichever way it occurs, a request for an invocation eventually reaches the relevant servicelet manager. The servicelet manager then creates a Colombo message. A new Colombo outgoing message is always assigned a unique identifier known as a "MessageID." The MessageID is serialized into a message header. If a response is expected, the servicelet manager registers the message in an outstanding-request registry, using the MessageID as a key. Finally, the servicelet manager hands the Colombo message to the message-processing machinery so that it can be sent to the partner.

In the Java case, the servicelet makes a request to its context to retrieve a stub. When a method is invoked on the stub, a sequence of events occurs, which are essentially the inverse of the adapter invocation described earlier. Typed XML serializers are used to write the argument types required by the operation being invoked.

The response to the invocation arrives separately as an incoming message to be dispatched to a waiting servicelet instance. When a message arrives at a servicelet manager, the latter has to find out whether it is a new request or the response to a previous invocation. The servicelet manager checks the `RelatesTo` field in the header. Following WS-Addressing, the value of this should be the MessageID from the invocation request message. Therefore, the servicelet manager can use this value to look up in its registry of outstanding invocations the invocation to which the MessageID belongs. Once found, the servicelet manager can dispatch the response appropriately.

Handling service requests

Upon deployment, each servicelet is registered with the relevant servicelet manager (Java, BPEL, etc.). Once a message comes into the system, the relevant servicelet manager is identified and handed the message. The servicelet manager, upon receiving an incoming request, has to find a servicelet instance to consume it or create one if none is found. The system will ask the servicelet manager for the instance of the conversation corresponding to the message. The system will also do any policy-related required work based on that information and eventually return to the servicelet manager with the

message so that it can be handed to and processed by the instance. Colombo uses a single, well-defined, reference property (the "Colombo instance ID") to identify instances, but the value of the reference property is otherwise opaque for requestors.

The default mechanism for looking up a conversation instance is to use the WS-Addressing headers. The uniform resource locator (URL) informs the servicelet manager which servicelet should consume the message. The servicelet manager then uses any available "reference property" headers to find out if there is an already existing instance to which this message should be routed. If there are no reference properties and the servicelet manager does not have any alternative routing schemes (such as BPEL correlation sets using the application data in the message), then the servicelet manager returns a new conversation instance for the appropriate servicelet. For optimization, the creation of the servicelet instance itself can be delayed until the message needs to be processed.

Once the system returns to the servicelet manager for the actual dispatching of the message to the instance, the servicelet manager simply passes it on. At this point it may create a new instance if it needs to. If a new instance is created, it is assigned a unique identifier ("Colombo instance ID") with which it is registered. The message is deserialized and handed to the implementation in the data format it expects either dynamically by the servicelet manager itself or by using an adapter if one is present. Currently we use the former in the BPEL servicelet manager and the latter in the Java servicelet manager.

In the case of a request/response operation, the response again goes through the servicelet manager. The servicelet manager creates a response message by using the destination endpoint reference that it obtains from the conversation context, the actual message content, and relevant information from the original request such as a `RelatesTo` header containing the request's message ID. A `ReplyTo` header is added containing the URL of the servicelet along with the unique identifier of the instance as a reference property. This is then handed to the outbound dispatcher along with information on any policies that need to be applied to the message on its way out.

Updating endpoint references

Once an instance is created by a servicelet manager, the endpoint references of its provided services (already containing a URL) are updated with a reference property consisting of the instance identifier. It is also registered with the servicelet manager by using the instance identifier.

The services that a servicelet invokes may also change their endpoint references at runtime. These changes occur when a request message arrives with an endpoint reference in the `ReplyTo` header that is different from the one saved in the servicelet instance information. In the section "Colombo support for WS-Addressing," we explained how this is done when there is a `ReplyTo` header in the message: the endpoint reference in the `ReplyTo` header overwrites the one saved in the conversation instance, provided that the former is not anonymous. A servicelet implementation may also ask that the endpoint reference of an active instance be updated. In this case, it needs to inform its servicelet manager so that the required endpoint reference updates take place. For example, in BPEL, one of the capabilities of the "assign" activity is to copy an endpoint reference passed by an application message into the endpoint reference to be used in subsequent interactions along one of its partnerLinks (conversations).

Instance deactivation

The servicelet manager deregisters servicelet instances once they are deactivated and are no longer able to consume messages. The cause of deactivation will depend on the type of servicelet (and its corresponding servicelet manager). For example, once all the activities in a BPEL process instance have been completed, that instance must be deregistered because it can no longer process messages. Another example is in the case of a Java servicelet, where a deactivate operation can be called at any time programmatically from within the implementation.

Policy framework

As described earlier, a key feature of SOAs is the explicit declaration of functional as well as QoS capabilities and requirements in agreed-upon, machine-readable formats. In the Web Services framework, policies are used to refer to meta-data for QoS attributes of services. The WS-Policy language[6] defines a base language and operators for policy

assertions; this is designed to be extended by domain-specific languages (such as languages for defining security or transactional policy assertions) to provide a rich framework for defining policies.

Policies are general statements about QoS and are attached to Web Services. A single service definition may have multiple policies attached to it simultaneously, at various levels of granularity (for example, to the service as a whole, a particular endpoint, an interface, an operation, or even a particular message type). In Reference 19, transaction policies are attached to BPEL processes. These policy attachments may be part of the service definition or may be externally specified. The WS-Policy Attachment specification[20] defines how policies are attached to Web services.

Colombo's policy subsystem is responsible for collecting, interpreting, and enforcing policies. Policies are enforced at the level of a conversation; that is, for each message entering or leaving Colombo, the policy subsystem needs to ensure that the message complies with the expected policy. Because policies are attached at different levels of granularity, interpreted policies need to be merged to compute the effective policy for each such message exchange.

Computing effective policy

The policies that the Colombo runtime applies to the interaction with a deployed service originate from three sources:

1. *Service policy*—The policy documents associated with a service that is made available using Colombo. These are attached to the service definition or a part of the service definition. Partner services that wish to use the functionality offered by such a service need to comply with the associated policies. As an example, a transaction policy may be associated with a service interface for an Internet banking service, because the semantics of the function offered through that interface require transactional behavior.
2. *Partner policy*—The policy documents associated with a partner service that is used by a Colombo service. These policies represent requirements for partners. For example, our example of an Internet banking service might offer a loan facility. This operation might make use of an external credit-check service. The banking service might require that the credit-check service follow the same transactional protocol so that transactions can be propagated and credit checks can occur within the same transactional context as a loan request transaction.
3. *Framework policy*—A policy document associated with a particular deployment of Colombo. It is used to enumerate the capabilities of the particular Colombo installation.

On deployment of a servicelet, the service policies and partner policies for each service supported by the servicelet are checked against the framework policy to ensure that each of the services does not have requirements that go beyond the platform capabilities. If any service requires a policy not supported by the platform, deployment of the servicelet fails. Note that partner policies represent requirements for partners; therefore, before partners are assigned endpoints (i.e., before the abstract partner interface becomes fully specified), the service policy of one partner must be determined to be compatible with the partner policy that is desired by the other partner. In Colombo, the abstract partner interface is fully specified at deployment. Thus, discovery of the offered policy of one partner and the checking of it against the desired partner policy of the other is assumed to have taken place before deployment. Doing this dynamically is a planned future area of research (Reference 21 offers one approach to arriving at negotiated policy). Based on this assumption, the semantics of the partner policies are that they represent the agreed-upon, negotiated policies for the interaction between the service and its partner, obtained by prior matching of the QoS requirements with partner offerings.

Next, effective policies need to be computed for each potential message destined to or originating from the services. This may be done at deployment or dynamically, when a message leaves or arrives at a service. In either case, the process is identical. The policies attached to each input and output message of a service endpoint are examined along with all higher scopes in the same hierarchy, such as policies attached to the service endpoint, interface definition, or I/O-message definition. These policies are merged to arrive at the effective policy for a particular input or output message, as shown in *Figure 9*. For example, the effective policy for input message of operation o1 in endpoint s1 is obtained

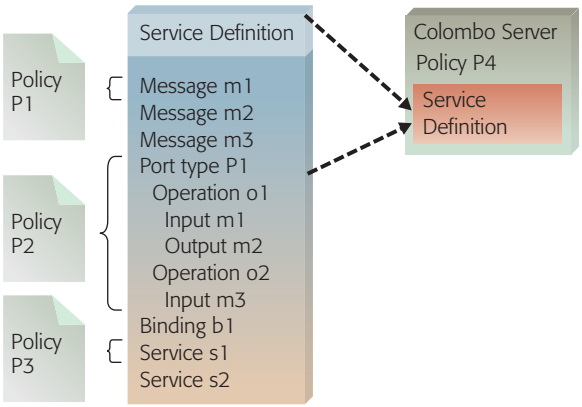

Figure 9
Calculating effective policy for operation o1

by merging policies P1 (attached to message type), P2 (attached to port type), P3 (attached to endpoint), and P4 (set of permissible configurations supported by platform).

The method for comparing and merging policy definitions is implied by the WS-Policy definition and described in some detail in References 22 and 23. Simply put, policies can be represented as Boolean expressions where policy operators are mapped to Boolean ones, and policy assertions form the terms being combined. Prior to merging or comparing policies, they are put into a normal form (analogous to disjunctive normal form (DNF) for Boolean expressions). In this representation, each of the terms in the DNF can be thought of as a permissible combination of QoS features. A merge of two policies can be obtained by defining a conjunction of the normalized policy expressions that each of the individual policies represents. The terms of normalized policy expressions can be compared. There are two caveats to these methods. The first one deals with policies that have different vocabularies. Consider two policies where one has an assertion relating to a security protocol (for example, a requirement that messages are signed using the RSA (Rivet Shamir Adlerman) algorithm, while the second policy has no assertions dealing with this particular security protocol. One solution would be to disallow such comparisons, but in practice they would likely occur often. We chose the solution of completing the policy where the term in question was missing, by adding its positive and negative form to each of the terms in the DNF form of the policy (thus doubling the number of terms). In other words, the policy would be extended to explicitly state that the security protocol was supported as well as not supported in each of the permissible configurations allowed by the policy (see policy-merge example later). Although this solution works for many practical cases, it does have inadequacies that we have described further in Reference 22. The second caveat involves dealing with effective policies that allow multiple configurations (i.e., the DNF of the merged policy has more than one term). Consider the situation where the incoming message for a particular operation offered by a particular endpoint of the service has a DNF with more than one term. For policy enforcement to take place correctly, when the message in question arrives, it has to be verified as following the policy correctly. This is possible only if there is prior agreement between the service and its partner as to which of the permissible configurations will be used. Some protocol is needed for making this determination. This is another area of work we plan to undertake in the future; currently, Colombo requires that all effective policies consist only of one term. This requirement is not a hindrance if QoS policies are defined narrowly.

Policy P1: `RSA_Signature (Subjects must follow RSA algorithm to sign messages)`

Policy P2: `AT_Mandatory (Subjects must participate in atomic transactions)`

Normalized forms:

```
P1n: (RSA_Signature ^ !AT_Mandatory) v
(RSA_Signature ^ AT_Mandatory)
P2n: (AT_Mandatory ^ RSA_Signature) v
(AT_Mandatory ^ !RSA_Signature)
```

Final merged policy: `ATMandatory ^ RSA_Signature`

Policy enforcement

After effective policies are computed, the enforcement mechanism for each effective policy can be put into place. As in the case of computation of effective policies, this can be done either at deployment or dynamically, when a relevant message is being processed by Colombo.

Information relevant for policy enforcement (such as transaction IDs or message digests) is carried along with messages as header information. This information is serialized or deserialized by code that

is specifically designed to deal with data following the requirements of a particular message channel (for example, according to SOAP rules). For incoming messages, deserialized header information can then be consumed by policy handlers to perform the function required by a particular policy, such as recording message arrival or departure in the case of a reliable messaging policy. For outgoing messages, the policy handler performs the function mandated by the protocol and adds any relevant header information (such as a message ID in the case of a reliable messaging protocol) to the message. This information is later serialized by the relevant header processor before the message's departure from the system.

The mechanism for policy enforcement is thus a linear chain of policy handlers, customized for a particular policy. For example, if a message has an associated policy requiring message reliability and participation in a 2PC (two phase commit) transaction protocol, Colombo has to ensure that relevant messages are routed to the transaction policy handler and reliable-messaging policy handler. This approach of using the interceptor pattern to handle QoS is well known.[21,24,25]

When is a handler chain computed? The procedure for doing this involves parsing the effective policy for a message to configure a handler chain appropriately. It can be performed when services are deployed in Colombo (this is the default), or at runtime. After a handler chain is computed, the configuration can be stored for use with future messages with the same effective policy. If this feature is disabled, handler chains are dynamically computed when the need to process a particular kind of message is identified, and then the chains are discarded. This offers the most flexibility as the system can adapt even to changes in the effective policy.

Policy handlers need access to services offered by the Colombo framework (such as access to a database), the ability to invoke external services (such as a transaction coordinator service) and features that affect the routing of messages within Colombo (such as the ability to interrupt or resume message delivery to a service endpoint). Many of these requirements are identical to the requirements of services hosted within Colombo. Policy handlers are thus designed as services hosted within a specialized container (the policy handler container), which offers access to the internal elements of the framework that are not available through other containers. Other than this important distinction and some specialized logic for routing messages to the policy handler container, policy handlers can conceptually be deployed and managed as regular services and can use a programming model identical to that used by servicelets. Currently though, Colombo comes packaged with a set of predeployed handlers supporting reliable messaging and transaction and security policies and does not allow additional handlers to be deployed.

A difficult problem in policy enforcement relates to the order in which messages are processed by handlers within a handler chain. Our design implicitly assumes that each policy handler works independently, and the order of message delivery is not relevant. This is usually not the case. Consider a policy that requires encryption along with reliable message delivery. When a message is encrypted, its headers, including any headers used by the reliable-messaging protocol (such as a Message ID), are also encrypted. Thus, decryption of the message has to take place before the message is handed to the processor or handler of the reliable messaging header. This is a dependency that comes out of the semantics of the policy of that message. We plan to explore solutions to this general problem of handler dependencies within the chain, but currently we treat security (in particular, encryption) as a special case within the policy subsystem and take care of that before message delivery to other policy handlers.

FUTURE WORK

Work in the Colombo project continues in three areas: completing the implementation of the runtime platform, extending support to developers, and testing and measuring Colombo as a compete platform.

Two major areas of the Web Services stack are still not supported by Colombo. One is discovery capabilities, either a registry-based discovery (UDDI) or dynamic meta-data exchange (WS-MetadataExchange). Colombo is already designed to take advantage of dynamic service discovery (e.g., through dynamic policy reconfiguration), but currently Colombo does not support service partner discovery other than static binding. The second area

is the use of the business activity protocol (WS-BusinessActivity[26]) to support loosely coupled business interactions, as opposed to the tightly coupled scenario where an atomic transaction protocol might be used.

Support for developers will be extended by supporting bottom up development strategies for Java and incorporating scripting languages (Javascript**, Perl) into the set of servicelet-supported languages. We are considering adding the capability to update an already deployed servicelet without stopping and restarting the Colombo server (hot update), to complement the existing hot deploy capability.

Finally, the ultimate goal of the Colombo project is to deepen our understanding of what kinds of middleware architectures are most appropriate to support SOC environments. We are planning to test Colombo's assumptions about the performance and simplicity yielded by a platform that natively supports the SOC paradigm through a set of performance benchmark tests and a field study focused on the developer's "ease of use" experiences.

SUMMARY

This paper has presented an overview of the Colombo platform, an experimental IBM Research project aimed at providing a deeper understanding of how middleware can address the requirements of the SOC paradigm. Colombo's approach is to build a native Web services runtime and programming model focused exclusively on the execution and development of service-oriented applications. Colombo thus does not address the problem of integrating legacy middleware and applications, except by assuming that they can be enabled for Web services by using the tools and adapters already available in the market.

The Colombo programming and deployment models achieve substantial simplification by extending the service paradigm to both application and system services and by concentrating on the composition problems. Colombo programming can be implemented in multiple languages; Java and BPEL realizations of this programming model are already available. The Colombo runtime is built on an asynchronous SOAP processing engine in which a policy-processing framework provides customizable QoS capabilities.

Colombo is an ongoing Research project. Work continues to provide support for service discovery standards, management interfaces, and extensions to the set of programming tools. The evaluation of the platform in the areas of performance, management, and developer's usability is now taking place, and is likely to yield important insight into the problem of designing and optimizing middleware to support service-oriented environments.

ACKNOWLEDGEMENTS

The Colombo project is a cross-departmental project. In addition to the authors of this paper, Tom Mikalsen, Stefan Tai, and Ignacio Silva-Lepe from the IBM Thomas J. Watson Research Center and Michiaki Tatsubori from the IBM Tokyo Research Center have been key contributors to the design and implementation of the Colombo platform.

CITED REFERENCES

1. F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing,* **6**, No. 2, 86–93 (2002).
2. F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana, "The Next Step in Web Services," *Communications of the ACM*, **46**, No. 10, 29–34 (2003).
3. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL 1.1)," W3C Note, http://www.w3.org/TR/wsdl, March 2001.
4. C. Szyperski, "Component Software: Beyond Object-Oriented Programming," *Addison-Wesley Professional*, Boston, MA, 1999.
5. D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple Object Access Protocol (SOAP) 1.1," http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.
6. D. Box, F. Curbera, D. Langworty, A. Nadalin, N. Nagaratnam, M. Nottingham, C. von Riegen, and J. Shewchuk, "Web Services Policy Framework (WS-Policy Framework)," 2002, http://www-128.ibm.com/developerworks/library/specification/ws-polfram/.
7. UDDI V3.0.1 specification, http://uddi.org/pubs/uddi-v3.0.1-20031014.htm.
8. F. Curbera and J. Schlimmer (Eds.), "Web Services Metadata Exchange (WS-MetadataExchange)" (September 2004), ftp://www6.software.ibm.com/software/developer/library/WS-MetadataExchange.pdf.
9. D. Box and F. Curbera (Eds.), "Web Services Addressing (WS-Addressing)," W3C Member Submission, August 10, 2004, http://www.w3.org/Submission/ws-addressing/.
10. L. Cabrera, G. Copeland, M. Feingold, T. Freund, J. Johnson, C. Kaler, J. Klein, D. Langworthy, A. Nadalin, D. Orchard, I. Robinson, T. Storey, and S. Thatte, "Web Services Atomic Transactions Specification" (September

2003), http://www-106.ibm.com/developerworks/library/specification/ws-tx/.

11. C. Ferris and D. Langworthy (Eds.) "Web Services Reliable Messaging Protocol (WS-ReliableMessaging)" (March 2004), http://www-106.ibm.com/developerworks/library/ws-rm/.
12. Web Services Security: SOAP Message Security 1.0 (March 2004), http://www.oasis-open.org/committees/wss/.
13. F. Curbera and N. Mukhi, "Metadata-Driven Middleware for Web Services," *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE 2003),* Rome, Italy, 2003, pp. 278–286.
14. R. Khalaf, N. K. Mukhi, and S. Weerawarana, "Service-Oriented Composition in BPEL4WS," *Proceedings of the 12th International World Wide Web Conference (WWW 2003) Alternate Track Papers and Posters*, Budapest, Hungary, May 2003, http://www2003.org/cdrom/papers/alternate/P768/choreo_html/p768-khalaf.htm.
15. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana, "Business Process Execution Language for Web Services (BPEL4WS) 1.0," http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.
16. Java API for XML-Based RPC (JAX-RPC), Sun Microsystems Inc., http://java.sun.com/xml/jaxrpc/.
17. PHP Hypertext Preprocessor scripting language, http://www.php.net/.
18. XML Pull Parser, http://www.extreme.indiana.edu/xgws/xsoap/xpp/.
19. S. Tai, R. Khalaf, and T. Mikalsen, "Composition of Coordinated Web Services," *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Toronto, Canada (2004), pp. 294–310.
20. S. Bajaj, D. Box, D. Chappell, F. Curbera, G. Daniels, P. Hallam-Baker, M. Hondo, C. Kaler, A. Malhotra, H. Maruyama, A. Nadalin, M. Nottingham, D. Orchard, H. Prafullchandra, C. von Riegen, J. Schlimmer, C. Sharp, and J. Shewchuk, "Web Services Policy Attachment (WS Policy Attachment)," ftp://www6.software.ibm.com/software/developer/library/ws-polat.pdf.
21. E. Wohlstadter, S. Tai, T. Mikalsen, I. Rouvellou, and P. Devanbu, "GlueQoS: Middleware to Sweeten Quality of Service Policy Interactions," *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, 2004, pp. 189–199.
22. N. Mukhi and P. Plebani, "Supporting Policy-Driven Behaviors in Web Services: Experiences and Issues," *Proceedings of the 2nd International Conference on Service-Oriented Computing* (ICSOC 2004), New York, USA (November 2004), pp. 322–328.
23. P. Nolan, "Understand WS-Policy Processing" (December 2004), http://www-106.ibm.com/developerworks/webservices/library/ws-policy.html.
24. P. Narasimhan, L. Moser, and P. Mellior-Smith, "Using Interceptors to enhance CORBA," *Computer* **32**, No. 7, 62–68, July 1999.
25. N. Mukhi, R. Konuru, and F. Curbera, "Cooperative Middleware Specialization for Service-Oriented Architectures," *Proceedings of the 13th International Conference on the World Wide Web—Alternate Track Papers and Posters* (2004), pp. 206–215.
26. Web Services Business Activity Specification, http://www.ibm.com/developerworks/webservices/library/ws-busact, January 2004.

Accepted for publication June 23, 2005.
Published online October 24, 2005.

Francisco Curbera
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York 10532 (curbera@us.ibm.com). Dr. Curbera is a research staff member and manager of the Component Systems Group at the Watson Research Center in Hawthorne, New York, where he has worked since 1993. He holds a Ph.D. degree in computer science from Columbia University. His current research interests are in the use of component-oriented software in distributed computing systems. In the past, he has worked in the design of algorithms and tools for processing XML documents and in the use of markup languages for automatic user interface (UI) generation. He has worked on different Web Services specifications since the initial Web services concept surfaced in late 1999, first as one of the original authors of the Apache SOAP implementation of SOAP 1.1, and then as co-author of WSDL 1.1, BPEL4WS, WS-Policy, and WS-PolicyAttachments, WS-Addressing, WS-MetadataExchange, and other Web Services specifications. He currently represents IBM in the Web Services Addressing working group, standardizing WS-Addressing at the W3C, and in the Web Services Business Process technical committee, standardizing BPEL4WS at OASIS.

Matthew J. Duftler
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York 10532 (duftler@us.ibm.com). Mr. Duftler is a software engineer in the Component Systems Group at the Watson Research Center. He was one of the original authors of Apache SOAP, is the co-lead of JSR110, Java APIs for WSDL, is a co-author of the IBM BPEL4WS engine, BPWS4J, and one of the main developers of Colombo.

Rania Khalaf
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York 10532 (rkhalaf@us.ibm.com). Ms. Khalaf is a software engineer in the Component Systems Group at the Watson Research Center. She received her Bachelor's and Master's degrees in computer science and electrical engineering from Massachusetts Institute of Technology in 2000 and 2001. Her interests include component-based software engineering, workflow, and service-oriented computing, Web services in particular. She is a co-developer and co-architect of the IBM BPEL4WS prototype implementation (BPWS4J) and the Java Record Object Model (JROM). She has published a number of papers on service-oriented computing and has served on the program committees of conferences and workshops in the field.

William A. Nagy
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York 10532 (wnagy@us.ibm.com). Mr. Nagy is a senior software engineer in the Component Systems Group at the Watson Research Center. He received a B.S. degree in mathematics/computer science from Carnegie Mellon University in 1994, and a M.S. degree in computer science from Columbia University in 2000. His research interests include the design and development of wide area distributed systems, including their software architecture, development and programming models, and supporting infrastructure. His most recent efforts have been focused in the area of Web services.

Nirmal Mukhi
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York 10532

(nmukhi@us.ibm.com). Mr. Mukhi is a software engineer in the Component Systems Group at the Watson Research Center. He received an M.S. degree in computer science from Indiana University in 1999 and is currently working on a Ph.D. degree. He has been studying programming models for Web services, service composition, orchestration, and related problems. His current area of work is in developing policy-driven, reconfigurable middleware for Web services.

Sanjiva Weerawarana
(sanjiva@openource.lk) Dr. Weerawarana received a Ph.D. degree in computer science from Purdue University in 1994. After a few years at Purdue as visiting faculty, he joined IBM Research in 1997, where he worked as a research staff member in the Component Systems Group. While at IBM, he was elected a member of the IBM Academy of Technology. His research interests are in component-oriented programming, specifically component-oriented distributed computing architectures. He has been an active contributor to IBM's technical strategy for Web services and is an author of many Web Services specifications including WSDL, WS-Addressing, WS-MetadataExchange, BPEL4WS, and WS-Resource Framework. Dr. Weerawarana has implemented many of those specifications and has contributed to multiple IBM Web services products and offerings. He left IBM in April 2005. He is a very active member of the open source community, as a member of the Apache Software Foundation and as co-founder of the Lanka Software Foundation, an open source foundation in Sri Lanka. In his leisure time, he teaches at the University of Moratuwa, Sri Lanka, where he lives. ■