Cooperative Middleware Specialization for Service Oriented Architectures

Nirmal K Mukhi
IBM T J Watson Research Center
P O Box 704
Yorktown Heights, NY 10598
nmukhi@us.ibm.com

Ravi Konuru
IBM T J Watson Research Center
P O Box 704
Yorktown Heights, NY 10598
rkonuru@us.ibm.com

Francisco Curbera
IBM T J Watson Research Center
P O Box 704
Yorktown Heights, NY 10598
curbera@us.ibm.com

ABSTRACT

Service-oriented architectures (SOA) will provide the basis of the next generation of distributed software systems, and have already gained enormous traction in the industry through an XML–based instantiation, Web services. A central aspect of SOAs is the looser coupling between applications (services) that is achieved when services publish their functional and non-functional behavioral characteristics in a standardized, machine readable format. In this paper we argue that in the basic SOA model access to metadata is too static and results in inflexible interactions between requesters and providers. We propose specific extensions to the SOA model to allow service providers and requestors to dynamically expose and negotiate their public behavior, resulting in the ability to specialize and optimize the middleware supporting an interaction. We introduce a middleware architecture supporting this extended SOA functionality, and describe a conformant implementation based on standard Web services middleware. Finally, we demonstrate the advantages of this approach with a detailed real world scenario.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques; D.2 [**Software**]: Software Engineering

Keywords

Service–oriented architecture, Web services, Metadata exchange, Middleware reconfiguration

1. INTRODUCTION

Service–oriented architectures (SOA) provide a promising way to address problems related to the integration of heterogeneous applications in a distributed environment. In an SOA environment, every application is assumed to be (potentially) under the control of independent service providers, external or internal to an organization. As a consequence, applications are required to declaratively define their functional and non–functional requirements and capabilities in an agreed, machine readable format, eliminating implicit and out–of–band assumptions about their behavior.

In the basic SOA model, service providers publish machine readable descriptions of their services in a publicly accessible registry; service requestors discover those services by querying the registry, and bind to the selected service dynamically. Automated service discovery, selection and binding become native capabilities of SOC middleware. Dynamic binding capability leads to looser coupling between applications and enables applications to efficiently adapt to a changing environment.

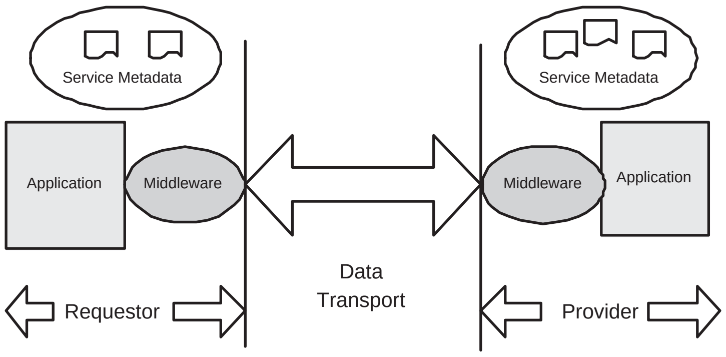


Figure 1: Service requestors and service providers

Services in a SOA environment may interact in a variety of ways which reflect the heterogeneity of supported applications, a consequence of the widespread interest that the Web services instantiation has created in almost every sector of the software industry. In the most general situation, services interact as peers, but traditional client server interaction are likely to be common as well. At each end of the interaction, services comprise three components on which we will focus our attention (see Figure 1): the *application* or *service implementation*, which provides the service business logic as well as its associated resources (databases, legacy systems resources, etc.;) the *service metadata*, which defines the "published view" of the application and is used by other parties to understand what the service does, and how to interact with it; finally, the supporting *SOA middleware* which provides the required interaction protocols and supports service discovery and binding. In Figure 1 the party initiating the interaction and the party receiving it have been labeled, respectively, *requestor* and *provider*.

Different SOA interaction models will be supported by different specializations of this configuration; that is, the symmetry of Figure 1 need not imply that in every SOA interaction both parties perform identical roles, or that identical middleware capabilities are at work at both ends. We are particularly interested here in situations in which the middleware function enabled at each end is dynamically determined at runtime. A typical example is that of a mobile application interacting with a service provider. By allowing the mobile application to assume middleware tasks otherwise performed by the service provider, "disconnected" operation can yield important performance gains at both ends. We argue here that the mechanisms available to retrieve service metadata can play a key role in supporting these sorts of scenarios.

Service metadata plays in central role in every aspect of SOA architectures; the reliance on machine readable metadata is probably one of the key defining aspects of SOAs. Metadata in the form of interface definitions, such as in CORBA's IDL definitions, has been present in distributed systems for a long time already [19]. Metadata in traditional systems has been mostly limited to functional aspects of the service, while non-functional and middleware interoperability characteristics have been treated as out-of band assumptions, or manually discovered at development or deployment time (for a discussion on this topic see [12]). The decoupling of implementations achieved by separating and publishing object interface definitions is generalized in SOA environments to include not just functional aspects of the service operation, but quality of service and middleware interoperability aspects as well. The impact of metadata in middleware architectures is discussed in [9] where it is argued that a new generation of middleware architectures is needed to leverage the pervasive use of metadata in SOAs.

In the standard SOA model, service providers publish in machine readable format all the information needed to access their services; a (requestor) party willing to use a certain type of service discovers services using a directory, and uses the published metadata to configure its application and middleware to access the selected service. The key point to stress is that the metadata published by the service drives not only the business logic of the requestor, but determines the middleware configuration it supports. In the basic publish, find, bind SOA model, however, metadata is fundamentally static and hard to tailor to the characteristics of specific users and usage scenarios.

The central idea of this paper is that service providers in an SOA should be capable of exporting different aspects of their functionality and operating environment, expressed in metadata, at runtime, based on the identity of the requestor and the execution environment. The view that a service requestor has of the service is thus customized. Based on this customized view, the requestor and provider can then specialize the middleware at each end, resulting in an optimized interaction. Different specializations may take place once the appropriate framework is available: specific communication protocols or data formats may be chosen, function may be offloaded to the requestor middleware, service behavior may be customized for the particular requestor or execution environment. We will use the Web services platform, an instantiation of the SOA concept, to illustrate how these ideas apply.

Solutions to this problem need to satisfy three major requirements in order to adequately fit in a SOA environment. First, they must avoid implicit architectural assumptions whenever possible so as to preserve loose–coupling between the service requestor and service provider; in particular, any assumptions made should be declaratively expressed by the service metadata. Second, middleware specializations should take place transparently to the applications, whenever functional aspects of the service are not affected. Finally, existing mechanisms for discovering and accessing services should be preserved; that is, operation without runtime specialization should be seamlessly supported.

This paper makes the following main contributions:

1. A proposal for a particular extension of SOA architectures wherein service providers selectively expose aspects of their functionality through metadata communicated at runtime to service requestors.

2. The concept of *cooperative SOA middleware*, which allows transparent specialization based on runtime access to service metadata.

3. An architecture supporting cooperative middleware specialization based on publicly available Web services middleware; we also describe a prototype implementation.

The rest of this paper is organized as follows. Section 2 reviews the potential benefits of middleware specialization. Section 3 introduces an extension to the basic SOA model supporting runtime customization of the service descriptions, and describes how cooperative specialization of middleware is enabled in this architecture. Section 4 describes the realization of this architecture as an extension to existing Web services framework, and Section 5 describes the implementation of the corresponding middleware support. In Section 6 we present the application of this architecture to a specific scenario. In Section 7 we review related work, and we conclude in Section 8.

2. BENEFITS OF SPECIALIZED MIDDLEWARE FOR WEB SERVICES

The Web services platform is an instantiation of the SOA concept. The platform consists of XML vocabularies to express protocols, service interfaces, registration of services, service discovery, policies, security, privacy and also the structure of service compositions. Some of the vocabularies (such as WSDL [8],[5], summarised below) are close to being standardized, other areas essential for business interactions (such as supporting Service Level Agreements) have not been addressed as yet. Even so, the platform has been met with wide–ranging industry support and already parts of the platform are being deployed in business applications.

A WSDL document describes a service in terms of four facets:

1. The data types, generally described using XML schema.

2. The interfaces, called *portTypes* in WSDL 1.1, which are collections of operations, each of which describes a particular pattern of message exchanges.

3. The protocol bindings, which describe how the functionality described in each of the interfaces can be accessed through a specific protocol. The dominant protocol binding used in the Web services platform is the SOAP messaging format with HTTP as the transport layer for messages.

4. A set of endpoints that collectively expose a service. Each endpoint refers to a particular binding and thus represents the availability of a specific interface through the described protocol, accessible at the location described in the endpoint.

The Web services framework is an instance of an SOA. As a result, service metadata is also a core concept in this framework. Think of a WSDL description of a service as the essential metadata of the service. This metadata is used by service providers to generate partial service implementations and configure supporting middleware and by service requestors to generate the necessary client paraphernalia to communicate with a service. Traditional distributed object systems such as CORBA rely on interface descriptions as being sufficient descriptions of a distributed object. Web services framework implementations have consciously or unconsciously adopted this notion for services and as a result rely on a WSDL document as being an adequate rendition of the service for requestors to use.

A service requestor, using an exported WSDL is able to use the locally available web services middleware to either generate a binding–specific proxy or a generic proxy and communicate with the web service by executing operations on the local proxy. The advantage of this approach is that it simplifies service development by completely factoring out the infrastructure available at a service requestor's end from the service provider. The service is cleanly encapsulated and isolated and offers a modular architecture for higher–level compositions. However, the problem is that the conceptual cleanliness of this approach is also employed in the execution of the web services middleware that support the interaction between service requestor and the service provider. Specifically, preserving this isolation between the provider and the requestor from the concept to the implementation comes with certain costs:

1. Since the proxy has no knowledge of the service beyond the service description, every call to the web service must necessarily perform a round trip to the provider. As a result, the proxy cannot leverage local resources (locally known Web services that provide better quality of service, available CPU cycles, etc.) when acting on behalf of the service.

2. Since all requestors are provided with the same coarse-grained view of the service, there is no direct support for customization of the service based on an individual requestor's execution context and its requirements.

These drawbacks are not much of a factor when placed in the context of the problems Web services is attempting to solve, viz. the integration of business applications. In this domain, information flow and updates must happen in a composite context of security, transactional spheres, processes, workflow, etc. The Web services framework is defining a unified architecture and normalized software infrastructure that can support the transformation of existing IT assets into services which businesses can use and integrate to perform effectively and efficiently in a heterogeneous, dynamic, distributed, multi–domain environment. The word *efficiency* is used in the sense of reducing an error–prone and costly manual process of communication that takes many days or weeks into something that can be performed in a period that is drastically less, such as day or a few hours. In other words, interactivity has not been the driving force or the main priority in the development of Web services.

However, the success of the Web services platform, its open standards and the multi–vendor support has made it attractive in non–traditional domains such in web portals that syndicate and aggregate remote content and services, high–performance computing carried out on Grids and pervasive computing. Clearly, responsiveness, mobility and customizability of applications are more important in these arenas than for traditional business applications that are not user–facing.

Optimizing interactions through dynamic middleware specialization thus has important implications for the use of Web services in non–traditional domains.

3. COOPERATIVE SPECIALIZATION OF SOA MIDDLEWARE

Here we discuss the use of SOA through an abstract model, the different classes of dynamism we are proposing, and then propose changes to traditional SOA platform designs to support the new model. This discussion is focused on three facets of SOA platform implementations: the way service metadata is handled, the structure of the middleware for service providers, and the structure of the middleware for service requestors. Finally we discuss how this model affects the sequence of events in interaction with a service in an SOA environment.

3.1 Service Metadata

Service metadata can be used in SOAs to describe functional as well as non-functional aspects of a service. Thus the service function (data types, interfaces), communication protocols used, details of other middleware aspects such as authentication and encryption protocols, transaction protocols, etc. can all be explicitly described using metadata. Collectively, the machine-readable metadata provides a potential requestor with enough information to have an interaction with the service automatically, as long as the metadata is based on accepted standards.

In our model we let $SM\{i_1 \ldots i_k\}$ be the k metadata elements that describe different aspects of the service provider. Traditionally, this is a that common subset of metadata that is relevant for the widest class of requestors and is sufficient for interacting with the service. The metadata thus generally includes the a description of the service functionality, data formats, and wire protocol information. This set is communicated to the service requestor (via the third piece in the architecture, the service broker, which is not central to our discussion). The view of the service available for the requestor is thus $S_{req} = SM\{i_1 \ldots i_k\}$. Note that traditionally this view is static.

We propose that the use of metadata to describe different aspects of the service provider needs to be extensive. Some of it may be relevant only for specific classes of requestors, kinds of communication or environmental conditions. It follows that some of this metadata might need to be kept private since it may be platform–dependent, expose essential portions of the business application's logic or server environment. So the recommended model advocates publishing, as before, a minimal subset of this metadata but makes an allowance for exchanging more details dynamically, once the service requestor's identity and environment is known. Thus, if the service metadata set is $SM\{i_1 \ldots i_p\}$, a requestor's first view of the service may be $S_{req} = S_{req\_initial} = SM\{i_1 \ldots i_l\}$ where $S_{req\_initial} \subset SM\{i_1 \ldots i_p\}$ but S_{req} can be refined at runtime, potentially to capture all p elements.

In order to achieve this, SOA implementations need to define standard ways for service requestors to refine their views of a service at runtime, based on the initial metadata they have available. Refinements may take many forms, such as exposing finer–grained aspects of interaction, and may also expose alternative realizations of a known aspect (such as alternatives to the previously known supported data communication protocols).

3.2 The Service Provider

The middleware on the service provider is configured depending on the metadata published. For example, the provider may be capable of understanding many different communication protocols, but is geared to handle just the ones published. Other aspects of the provider middleware are not affected by the metadata. For example, there may exist a data caching policy implemented by the middleware but this is not advertised through the metadata and this middleware piece thus remains unaffected.

We consider the middleware on the service provider as a set of *logical* software components $SP_{infrastructure} = SP\{j_1 \ldots j_v\}$, where each j_i is a logical piece that has independent responsibility of some middleware task, such as understanding wire protocols, marshalling and unmarshalling data according to some prescribed format, enforcing security, handle caching, and finally handing off data to and receiving data from the application. Each of these tasks is logically independent. In practice however, a single piece of software may perform more than one task and the fine-grained tuning needed by our model may not be possible. Often the application itself performs some of these tasks. This practical reality affects the extent to which our model can be adopted by real–world systems, a subject we will revisit at the end of this section.

In order to successfully enable interactions for a particular service, a subset of these logical components are required to be operating, let's say $SP_{interaction} = SP\{j_f \dots j_h\}$ where $SP_{interaction} \subseteq SP\{j_1 \dots j_v\}$. Traditionally, this set is static for a particular service.

In order for specialization on a per–requestor basis, we propose to dynamically allow $SP_{interaction}$ to be modified. We consider two classes of changes:

1. Adding to $SP_{interaction}$: Based on information exchanged at runtime, software components may need to be plugged in at runtime. For example, based on runtime negotiation, the requestor may advertise its knowledge of a high-performance protocol. The provider middleware may have the capability to use this protocol (i.e. the logical software component is a member of $SP_{infrastructure}$ but may have to be added to the active set $SP_{interaction}$).

2. Reducing $SP_{interaction}$: In other cases it may be necessary to cull members from the set. For example, the provider may have middleware that handles encryption and decryption of data to and from the requestor. If it is discovered at runtime that the requestor resides within the same corporate intranet as the provider, it is no longer necessary to protect the confidentiality of the data, so both sides may agree to turn off the encryption.

Thus the provider middleware for an interaction, $SP_{interaction}$ will remain a subset of $SP_{infrastructure}$ but may undergo change dynamically prior to and possibly during an interaction with a particular requestor. In order to achieve this degree of flexibility, the middleware for the service provider has to be dynamically reconfigurable.

3.3 Service Requestor

Similar arguments apply for service requestor middleware as those presented above for the service provider middleware. The software infrastructure available to the requestor can be represented by the set of components $SR_{infrastructure} = SR\{i_1 \dots i_w\}$, where each i_k is a logical software component. For a particular interaction, the set $SR_{interaction}$ is used, where $SR_{interaction} \subseteq SR_{infrastructure}$. Each member of $SR_{interaction}$ is responsible for managing a particular aspect of interactions with service providers. Specialization on a per-interaction basis requires that $SR_{interaction}$ change dynamically responding to the runtime negotiation between the requestor and provider. Thus dynamically reconfigurable middleware is a prerequisite for supporting this model.

3.4 Refining Service Interactions in SOA

Traditionally SOA implementation follow a three step mechanism for enabling interaction with a service. Service providers advertise information about a service (the *publish* step), requestors discover services through a broker or registry (the *find* step), they then connect to the service (the *bind* step) and make use of it. Our model requires the addition of two steps, described below:

1. The *find-more* step: This involves an interaction between the requestor and provider where the requestor asks the provider for more information about the service. The provider may require the requestor to prove its identity, or to provide some information about its environment before it can supply the metadata being requested. A service provider has to provide support for a standard *find-more* procedure through the service endpoint and such support must be advertised in the base metadata that was discovered by the requestor. If this is not done, the requestor will assume that dynamic middleware specialization is not possible for interactions with this service. Thus the proposed model is a clean extension to base SOA.

2. The *negotiate* step: A requestor and provider can negotiate to create a more optimized end–to–end interaction, based on the information they have about each other. Negotiation consists of a series of exchanges that can take various forms. For example, if a provider advertises its detailed data model with all the constraints and data relationships specified in a standard manner, the requestor might negotiate to enforce the conformance with the required data model on the data before it leaves the requestor. Server offloading is common in client–server architectures, and this would be its equivalent in SOA. As another example, a provider may publish its dependencies (other services it depends on) and the requestor can then negotiate to supply the provider with matching service instances, resulting in a more customized interaction. In each case, the set of message exchanges depends on the kind of negotiation taking place.

It must be noted that the *find-more* and *negotiate* steps can be used prior to any business interactions between applications to optimize the middleware. However, they may also take place at any time during an interaction with a service. The possibility of gaining more knowledge about a service dynamically results in potential benefits for long–running interactions.

Additionally, we have pointed out before that loose–coupling is one of the main benifits of SOA. Entering into detailed discovery and negotiation that is driven by an individual requestor and provider's environment and execution context therefore seems counterintuitive to SOA's aims. An observation here is that specialization that is performed dynamically on a per-interaction basis does not change the loose–coupling between the same service provider and other requestors. Also, we mandate that the *find-more* and *negotiate* steps be *optional*. Thus in the proposed model it is possible to maintain existing loose–coupling as well as enter into interaction–specific optimizations if they are supported by the interacting parties and it is determined that there is potential benefit that can be gained.

3.5 Designing dynamically reconfigurable middleware for SOA

We have already discussed the need for modifying the set of logical software components used in the middleware on both the service provider and service requestor ends of the interaction. In real–world systems, these logical tasks are often performed by a small number of independent software pieces which may be tightly coupled to the application code that implements the business logic. In such cases, reconfiguration on a per–interaction basis may be possible only to a limited extent, if at all.

We propose the creation of a middleware framework that is a first–class realization of the logical structure we described above. The middleware consists of composable pieces of software, each of which has independent responsibility for a part of the end-to-end interaction. The middleware for a particular interaction can be created by dynamic composition of such pieces. In the next section, we describe an implementation based on this design. Our prototype supports this model in the context of the Web services framework.

4. COOPERATIVE MIDDLEWARE FOR WEB SERVICES

4.1 Metadata representation

The metadata representation used in our prototype are XML documents that conform to the WS-Policy specification [3]. WS-Policy provides a general-purpose model to describe and communicate non-functional aspects of service information. WS-Policy is a domain-neutral framework, which is used to express domain-specific policies such as those for reliable messaging, security, and so on. It provides the grammar to express and compose both simple declarative assertions as well as conditional expressions.

```
<wsp:Policy xmlns:wsp="..." xmlns:wscache="...">
   <wsp:ExactlyOne>
      <wscache:Response maxSize='1MB' time='30m'/>
      <wscache:Response maxSize='10MB' time='5m'/>
   </wsp:ExactlyOne>
</wsp:policy>
```

The above policy describes the caching requirements for a particular application. One of two alternative policies can be supported. The first requires that responses upto 1MB in size be cached for a minimum of 30 minutes, while the second requires that data upto 10MB in size be cached for a minimum of 5 minutes. The WS-Policy specification defines the framework for the policy declaration (the XML elements with the `wsp` prefix) while domain-specific data such as the caching parameters defined with the `wscache` namespace prefix are separate.

Each policy that applies to a particular interaction will generally have a corresponding piece of middleware that has the responsibility for enforcing the policy. For example, consider a policy that declares that business data will be validated according to a specific schema. The enforcement of the policy would entail parsing business data to check conformance with the schema. A caching policy such as that shown above would be backed up by a piece of software that manages such a cache according to the prescribed rules, thus enforcing the policy.

4.2 Runtime Metadata Discovery Interface

We define a Web services port type that will be supported by all services allowing specialization. This port type allows the dynamic discovery of metadata. In this implementation, the provider must be presented with the requestor's security credential which it can authenticate prior to supplying the information. The port type supports two operations. The *getInformationTypes* operation returns a list of XML namespaces, each of identifies a type of metadata. Each metadata piece has associated semantics which the requestor may or may not understand. The requestor can then ask the provider to supply details on a type of metadata it recognizes using the *getInformation* operation. The WSDL port type definition is presented below.

```
<portType name="DynamicMetadata">
  <!-- 'credential' is a schema type for a    -->
  <!-- security credential                    -->
  <!-- 'metadata-IDs' is a schema type for a  -->
  <!-- list of URIs -->
  <operation name="getInformationTypes">
     <input name="credential" type="credential"/>
     <output name="information" type="metadata-IDs"/>
  </opration>
  <!-- metadataReq is a schema type           -->
  <!-- that encapsulates a URI and credential -->
  <operation name="getInformation">
     <input name="id" type="metadataReq"/>
     <output name="information" type="string"/>
  </operation>
</portType>
```

Figure 2: WSDL port type for dynamic metadata exchange

4.3 Runtime Negotiation Interfaces

We define a set of Web services port types for runtime negotiation that enable different kinds of optimizations in a service interaction. As we indicated in section 3 the exact messages exchanged depends on the kind of negotiation taking place. The key therefore is to allow the support of an extensible set of negotiation interfaces.

In our prototype we support two particular negotiation interfaces, described in detail in the subsections below. These interfaces enable dynamic negotiation to impact the responsiveness and customizability of the service. These were identified in [15] as being critical to the application of the Web services in non–traditional domains such as mobile computing. The first interface deals with offloading functionality. Some aspects of the service functionality will be amenable to offloading, and allowing this to be discovered and negotiated can have a positive impact on the response time in an interaction with the service. The second interface deals with configuring a service's dependencies. By allowing requestors to match a service's dependencies to third–party providers, we demonstrate customizability through dynamic negotiation.

4.3.1 Offloading negotiation

The first interface, called *Offloading-Negotiation* is designed to allow offloading of functionality from the provider to the requestor middleware. It defines one operation, *propose*. This interface supports a particular protocol for negotiating offloading. In this protocol, there is only one type of message: a set of metadata items, identified by URIs. Each metadata item identified in the message must imply the existence of some middleware functionality that can take place on either the provider or requestor ends without affecting the end–to–end interaction from the viewpoint of the applications. The data validation and caching policies we described above would fall into this category. The protocol is described below:

1. The requestor sends to the provider a set of URIs identifying a set of metadata. This represents a proposal from the requestor. It consists of service metadata items whose corresponding functionality it can take responsibility for. An example of a candidate metadata item is a detailed data model described using XML schema. This schema would be embedded within a standard data policy format that would include parameters relating to strictness of the validation. The semantics of this metadata are well–known and it is a candidate for use in an offloading negotiation since it implies the existence of some middleware (in this case, a schema validator). Moving schema validation from the provider to the requestor will result in round–trips being saved, as long as the requestor understands the data model and has the software support to validate data against this model.

2. The provider responds with a message of the same form. This is a counterproposal, and consists of a subset of the original proposed metadata items. The counterproposal may be identical to the proposal, in which case it is a communication of the acceptance of the requestor's proposal. Otherwise, it reflects a disagreement and represents a viable set of metadata the responsibility for which can be managed by the requestor safely and within the limitations of the provider's reconfigurability. The latter case may arise because the provider's

software may not be factored in a manner that allows the reconfiguration that would be required following the offloading. For example, if the requestor proposes to take the responsibility for tasks related to metadata items SM_{i1}, SM_{i2} and SM_{i3}, it may be that the software on the provider that manages tasks related to SM_{i1} (which need to be disabled if the functionality is to be offloaded) also takes care of tasks related to SM_{i4} (which needs to be kept operational), and that the software cannot be reconfigured to manage just one task but not the others.

3. The proposals and counterproposals may continue until:
   (a) A predefined number of maximum proposals is reached
   (b) A proposal and counter–proposal match (reflecting agreement) or
   (c) The requestor declines to make further proposals.
4. If a proposal is agreed upon, the requestor and provider middleware must reconfigure themselves to manage all future communication in a manner consistent with this agreement.

Figure 3 is the WSDL port type that represents this interface.

```
<portType name="Offloading-Negotiation">
<!-- ’proposalReq’ is a schema type that encapsulates -->
<!-- a list of metadata IDs (URIs) and a security     -->
<!-- credential                                       -->
<!-- ’proposal’ is a schema type that encapsulates    -->
<!-- a list of metadata IDs                           -->
<operation name="propose">
   <input name="initialProposal" type="proposalReq"/>
   <output name="counterproposal" type="proposal"/>
</operation>
</portType>
```

Figure 3: WSDL port type for offloading functionality from provider to requestor

4.3.2 Dependency negotiation

The second negotiation interface, called *Dependency-Fulfillment* is designed to allow requestors to configure a service's partners. Each service can advertise its dependencies, or partners, using a standard policy format. These partners then need to be mapped to service instances, or businesses that can actually fulfill those dependencies. The policy can declare the list of default partners that will be used in the absence of any negotiation. By allowing an upfront declaration of depdendencies using a standard policy format, we allow a requestor to configure the partners used by a service to make the service more customizable. For example, a travel reservation service might allow requestors to configure which car rental agency is used, and the requestor can pick the one that with which it has a previous relationship.

This interface has one operation, *setPartner*, which requires that the requestor send the provider a service reference for one of the partners listed in the service's dependency set. The provider is required to send the requestor a confirmation of the fulfillment of this dependency through the provided reference. This operation can be called repeatedly to fulfill more dependencies. The WSDL interface is presented in figure 4.

4.4 Protocol for service interactions

Our new interaction protocol took into account the possibility of the existence of port types that enabled cooperative specialization of the middleware at runtime. The steps in a service interaction following this protocol is as follows:

```
<portType name="DependencyFulfillment">
<!-- ’partnerData’ is a schema type that encapsulates -->
<!-- a security credential, partner name and a        -->
<!-- service reference                                -->
<operation name="setPartner">
   <input name="partnerRef" type="partnerData"/>
   <output name="confirmation" type="string"/>
</operation>
</portType>
```

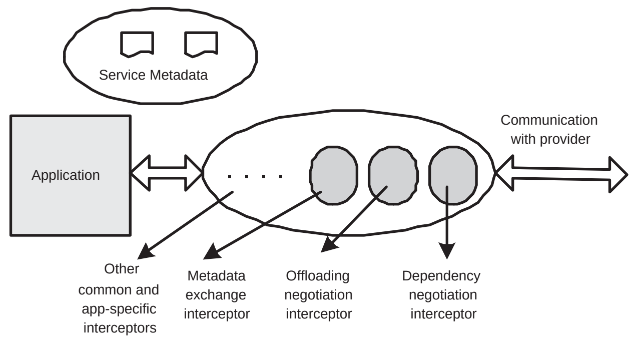
Figure 4: WSDL port type for fulfillment of service dependencies

1. *Exploration:* When the requestor application first makes use of a service, the requestor middleware determines if the predicted volume of interactions with this service is high enough to justify optimizing the middleware used. If this is *not* the case, we advance to step 5 of the protocol and the service interaction proceeds as usual, otherwise we enter the optimization protocol described in steps 2–4.
2. *View augmentation:* The requestor middleware examines the service description to see if the *Dynamic-Metadata* port type is available. If it is, the requestor invokes the *getInformationTypes* operation to get the list of metadata items available. For each metadata item the requestor understands the semantics of, it invokes the *getInformation* operation so it that the metadata is available to the requestor middleware. At the end of this step, the requestor's view of a service is complete to the extent possible.
3. *Negotiation 1 (Dependency Negotiation):* The requestor middleware checks if the service view includes a description of the service's partners. If so, it examines each partner definition to check if it is able to find a matching service instance it would like to use in place of the (random) choice made by the provider. For each such service instance, the requestor invokes the *setPartner* operation on the *Dependency-Fulfillment* port type of the service.
4. *Negotiation 2 (Offloading Negotiation):* The requestor middleware examines the service view to see if it includes metadata items that imply some service function that can be offloaded. Based on this examination and the reconfigurability of the requestor middleware, it enters into an offloading negotiation as described in 4.3.1.
5. *Standard Interaction:* The interaction between the applications takes place.

5. COOPERATIVE MIDDLEWARE IMPLEMENTATION

The Web services middleware we used in our prototype was based on the Web Services Invocation Framework (WSIF) [11], [17] and the Web Services Gateway [17]. WSIF is a framework for using WSDL–described services in a protocol independent way and was a starting point for our requestor–end middleware. The Web Services Gateway defines an endpoint on a server to which Web services can be deployed and is the basis for our provider–end middleware.

As we described in section 3, we aimed to create independent pieces of software we could compose in order to create the customized middleware pipe for a particular interaction. This was achieved through the use of message interceptors. Each interceptor is a module that performs a small well–defined task and is handed



Service Metadata
Application
Communication with provider
Other common and app-specific interceptors
Metadata exchange interceptor
Offloading negotiation interceptor
Dependency negotiation interceptor

Figure 5: Requestor middleware design

messages that flow to and from the application. Interceptors can be composed into linear chains and can be deployed on a server–wide basis. Interceptors are also programmable, for example it is possible to disable an interceptor so that it acts as a no–op when handed a message, and also re–enable it later.

The set of interceptors initially installed represents the pool of middleware components from which interaction–specific chains can be composed. This pool can be augmented by installing new interceptors dynamically. An initial set of interceptors are chained to manage the base case, just like static protocol stacks used in current middleware infrastructures. However, the possibility of specializing the chain on a per–interaction basis through the addition or removal of interceptors from the available pool differentiates our approach. This design principle is used on both the requestor and provider ends, as described below.

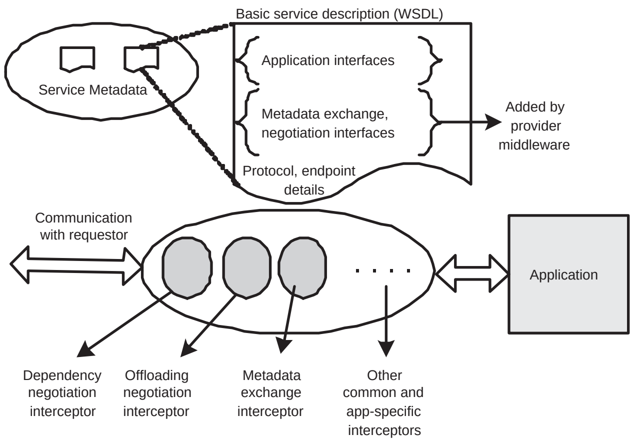
5.1 Requestor middleware design

WSIF provides requestors with a WSDL–based view of a service. The API for interacting with a service is driven off the abstract service description in the WSDL, and is independent of the actual service instance used. Software layers beneath the API can intelligently choose an appropriate binding (protocol) and a particular service instance. The WSIF framework allows customizations in the interaction protocol followed. This pluggable design allows us to introduce specific interceptors that enable applications to transparently follow the interaction protocol we designed. We introduce interceptors for managing metadata exchange, managing the offloading negotiation protocol and assigning service dependencies. These interceptors handle the requestor's role in steps 2–4 of the interaction protocol described in section 4.4. Figure 5 shows the design for our requestor middleware.

5.2 Provider middleware design

The Web Services Gateway is a pluggable framework that acts as a virtual endpoint for a set of services. Messages are routed to the correct service instance, and the gateway also supports interception of messages. A valuable feature of the gateway is that it is capable of allowing data to be exchanged through an extensible set of channels, each of which supports a particular communication protocol. The data arriving at a channel is normalized to a form that can be consumed by the service implementation. Thus a service can transparently augment its capability to understand a different protocols through the use of the gateway as a virtual endpoint.

For our design we modified the service deployment mechanism within the gateway so that each service description gets augmented with *Dynamic-Metadata*, *Dependency-Fulfillment* and *Offloading-Negotiation* port types. We designed three standard interceptors to



Basic service description (WSDL)
Service Metadata
Application interfaces
Metadata exchange, negotiation interfaces
Added by provider middleware
Protocol, endpoint details
Communication with requestor
Application
Dependency negotiation interceptor
Offloading negotiation interceptor
Metadata exchange interceptor
Other common and app-specific interceptors

Figure 6: Provider middleware design

support these functions. These interceptors handle the provider's role in steps 2–4 of the interaction protocol described in 4.4. In addition, a variety of interceptors responsible for enforcing the various policies associated with the application can be plugged in. Figure 6 illustrates the design of the provider middleware.

6. SCENARIO

Our example scenario describes an interaction between two applications: a data portal that provides information and access to common applications used by employees of a large company, and an employee database application, which allows employee records to be looked up. The employee database application is accessible to employees through the data portal, so that a subset of an employee's information (such as contact number and office location) is available to other employees. Through this scenario, we demonstrate how an optimized interaction is enabled through the use of the proposed architecture.

The employee database application is modeled as a Web service. It is described in a standard manner using WSDL. This WSDL description is deployed to our provider middleware. This application makes use of another application for converting data retrieved from the database in XML format into a form that can be rendered more easily, such as HTML. This converted data is then sent back to the requestor. The data translation service is thus a dependency for the employee database service. This dependency is declared in a standard manner using the BPEL4WS language [2] but this metadata is initially kept hidden from requestors. The data exchanged is validated against a prescribed schema, the parameters associated with this validation are laid down in a WS-Policy document which is not made public. A schema validator is among the pieces of infrastructure used for service interactions, and is implemented as an interceptor in our provider middleware.

The requestor end, consisting of the data portal application, accesses the employee database service using the information contained in the WSDL for the service. The portal application uses WSIF's API to make queries as directed by the end user. In addition to the interceptors for managing metadata exchange, dependency negotiation and offloading negotiation, a schema validation piece, implemented as an interceptor, also exists in the available requestor middleware infrastructure. Finally, the data portal detects when an end–user's device is a PDA or mobile phone and is aware of the location of a service capable of converting XML data to a renderable format for such devices.

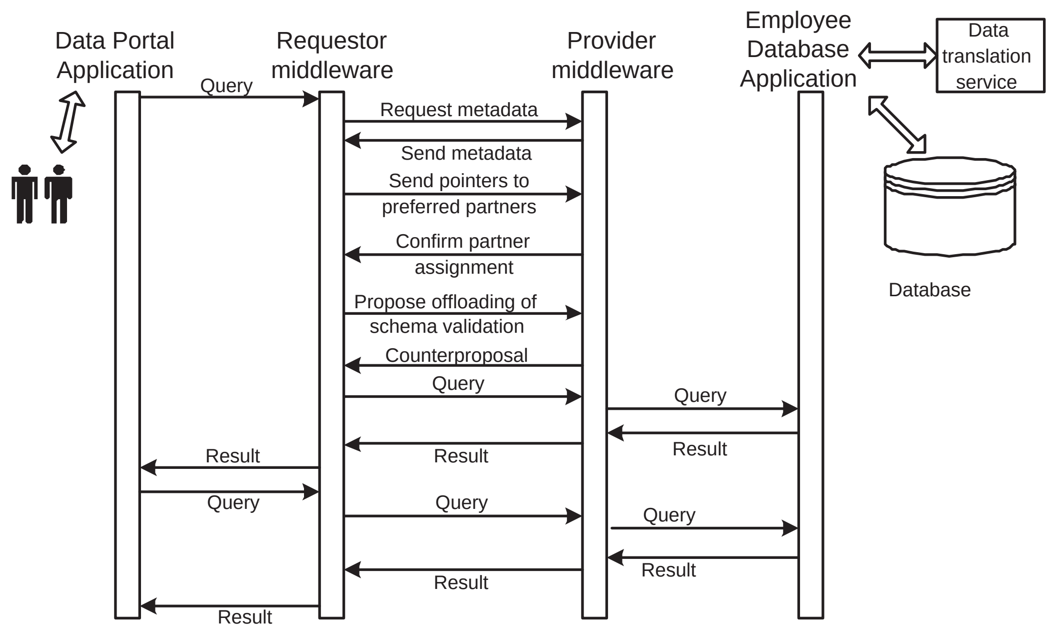


Figure 7: Message sequence chart for scenario

Following the protocol described in 4.4 the result of the initiation of an interaction between these services is:

1. The view of the employee database service available to the data portal is enhanced with the addition of the BPEL4WS document describing the partners used by the employee database application and the data validation policy document.

2. The instance of the data translation service used by the employee database application is set by the data portal so that the data is rendered in a form appropriate for the end user.

3. Schema validation is offloaded from the provider to the requestor middleware.

Figure 7 shows the sequence of message exchanges in this scenario.

7. RELATED WORK

The issue of specializing service interactions with respect to the context (device and environmental characteristics) of the requestor is a well–known one especially in mobile computing. [4] discusses the use of metadata to communicate non–functional aspects of an application, which when combined with a “reflective middleware” enables application customisation for each user in a mobile environment. [13] proposes an adaptation architecture that uses profile discovery, matching and finally application reconfiguration in order to customise application behavior for a particular environment. In both of these works middleware reconfiguration is triggered by the end-user as opposed to the transparent mechanism described in this paper. [6] discusses the use of “mobilets”, pieces of software that can be downloaded, pushed or migrated. They are managed by specialized middleware running on a moble client and the server with which it interacts. The latter two approaches rely on deployment of identical middleware. This differs from the model proposed here which does not demand any particular middleware implementation on the requestor and provider to enable optimization, as long as they support the standard port types for metadata exchange and negotiation.

The middleware framework we describe to support this model uses Web services standards in order to operate in a heterogeneous environment. Another popular standards–based middleware platform is CORBA, and it has been the proving ground for many related efforts. It is extended in DADO [20] to use reconfiguration aimed at easing development and deployment of cross–cutting features. The motivation for reconfiguration here is thus to adapt to a change, rather than for the purposes of optimizing an interaction as we attempted to do. Dynamic TAO [14] is an early work in the area of designing a CORBA ORB to enable application reconfiguration (in this case for the purposes of injecting new functionality), while OpenCOM [7] is an implementation of core parts of COM with the same end goal. [18] discusses enabling some classes of adaptation without having to resort to implementing an ORB, and uses an interceptor architecture similar in flavor to the one described in this paper. Smart CORBA proxies such as those described in [16] is another approach to allowing transparent optimization of an interaction.

Through our use of runtime negotiation to enable offloading, we attempted to address the problem of frequent round tripping to the server. This is recognized in the web application domain where programmers resort to using JavaScript to perform not only just validation but also several other functions such as sorting and simple computations that do not go back to the server. The XFORMs specification [1] also addresses this problem by supporting the definition of data models and corresponding constraints. An XFORMs compliant browser, on receiving an XFORM document, can perform a great deal of validation including at the level of instance values without requiring any code from the server or round trips to the server. Our approach advocates a similar expression of suitable metadata, going beyond the realm of presentation data alone. Traditional server offloading mechanisms are usually hardcoded into the server and client applications. This paper proposes discovering optimization possibilities dynamically, and is thus more flexible.

8. CONCLUSIONS AND FUTURE WORK

We conclude with some observations about the proposed model and our prototype implementation, and discuss possible areas for future work.

In our model, optimizations are performed with the constraints of the 'provider' and 'requestor' roles and are localized to a subset of the interactions between the two applications involved in these roles. In most practical cases, the same application both requires certain functionality (thus acting as a requestor) and provides certain functionality (this acting as a service provider) when interacting with another application. Optimizations can be performed in a more global context, with a recognition of all roles played by the applications in of their interactions.

Our prototype is a first attempt at realizing the proposed model and suffers from a number of limitations. Even though the *find–more* step in our model discusses two–way exchange of metadata, our Web services port type for metadata exchange, as designed, allows metadata flows from providers to requestors only. Ultimately a requestor makes the decision as to what metadata is useful and enters into negotiation protocols to enable optimizations. In many cases, a service provider can send requestors relevant metadata by discovering more about the requestor environment. Thus metadata needs to be a exchanged both ways in order to optimize interactions. Additionally, our prototype supports dynamic metadata exchange, negotiation and the reconfiguration as a one–time action that takes place at the beginning of an interaction between a requestor and provider. As we noted in section 3 long–running interactions can benefit from triggering metadata exchange, negotiation and reconfiguration even midway through an interaction, based on environmental changes, hardware events (such as major shifts in network loads) or user preferences. We plan to enhance our service interaction protocol to allow for this possibility, and re–engineer our prototype to support it.

Dynamic reconfiguration of middleware based on runtime information discovery has obvious performance implications. The cost of undertaking extra steps for discovery and negotiation have to be balanced against the gain from such optimizations. Additionally, the frequency with which such negotiation is triggered is another factor in the cost analysis. These topics need to be addressed in future work.

The steps we added to the interaction protocol introduce potential security loopholes in the transaction taking place, since not only application data but also policies that may impact the middleware flow on the wire. We discussed in section 4 the need for requestors to present security credentials when performing additional discovery. This is insufficient for real–world applications. A trust relationship has to exist between parties in order for such potentially dangerous information exchanges and negotiations to take place. New industry specifications such as WS-Trust [10] address some of the issues.

We have also discussed the transparency of dynamic change to applications as being one of our goals. That is why, in our prototype, the interceptors that manage metadata exchange, negotiation and the reconfiguration of other interceptors are all in the middleware, hidden from the application. In mobile applications in particular, this can be counterproductive in creating a customized interaction for an end–user, as indicated in [4]. Instead, allowing an application to 'steer' reconfiguration and negotiation is desirable. Taking care of this requirement within the bounds of a loosely–coupled architecture is another challenging problem we expect to explore in future work.

Acknowledgment

The authors would like to thank Rania Khalaf for useful comments during the writing of this paper.

9. REFERENCES

[1] XForms 1.0. Published on the World Wide Web by W3C, November 2002.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services Version 1.1. Published on the World Wide Web, May 2003.

[3] D. Box, F. Curbera, D. Langworthy, A. Nadalin, N. Nagaratnam, M. Nottingham, C. von Riegen, and J. Shewchuk. Web Services Policy Framework (WS-Policy Framework). Published online by IBM, BEA, and Microsoft at http://www-106.ibm.com/developerworks/webservices/library/ws-polfram, 2002.

[4] L. Capra, W. Emmerich, and C. Mascolo. Reflective Middleware solutions for context-aware applications. In *International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection)*, Kyoto, Japan, September 2001.

[5] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. Published on the World Wide Web, Mar 2001.

[6] S.-N. Chuang, A. T. S. Chan, J. Cao, and R. Cheung. Dynamic Service Reconfiguration for Wireless Web Access. In *12th International World Wide Web Conference (WWW 2003)*, Budapest, Hungary, May 2003.

[7] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzes. An efficient component model for the construction of adaptive middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, Heidelberg, Germany, November 2001.

[8] 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(2):86–93, Mar/Apr 2002.

[9] F. Curbera and N. K. Mukhi. Metadata-Driven Middleware for Web Services. In *To appear in the proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE 2003)*, Rome, Italy, December 2003.

[10] G. Della-Libera, B. Dixon, P. Garg, P. Hallam-Baker, M. Hondo, C. Kaler, H. Maruyama, A. Nadalin, N. Nagaratnam, A. Nash, R. Philpott, H. Prafullchandra, J. Shewchuk, D. Simon, E. Waingold, and R. Zolfonoon. Web Services Trust Language (WS-Trust). Published online by IBM, Verisign, Microsoft and RSA Security at http://www-106.ibm.com/developerworks/webservices/library/ws-trust, 2002.

[11] M. J. Duftler, N. K. Mukhi, A. Slominski, and S. Weerawarana. Web Services Invocation Framework (WSIF). In *OOPSLA 2001 Workshop on Object-Oriented Web Services*, October 2001.

[12] W. Emmerich. Software engineering and middleware: a roadmap. In *Proceedings of the conference on The Future of*

Software Engineering (ICSE 2000), pages 117–129, Limerick, Ireland, June 2000.

[13] N. Houssos, S. Pantazis, and A. Alonistioti. Generic adaptation mechanism for the support of context-aware service provision in 3G networks. In *IEEE 4th International Conference on Mobile Wireless Communication Networks (MWCN 2002)*, Stockholm, Sweden, September 2002.

[14] F. Kon, B. Gill, M. Anand, R. H. Campbell, and M. D. Mickunas. Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents. In *Proceedings of the IEEE Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA'2000)*, Zurich, September 2000.

[15] R. Konuru and N. K. Mukhi. Requestor Friendly Web Services. In *First European Workshop on Object Orientation and Web Services (EOOWS)*, Darmstadt, Germany, July 2003.

[16] T. J. Mowbray and R. C. Malveau. *CORBA Design Patterns*. John Wiley and Sons, 1997.

[17] N. K. Mukhi, R. Khalaf, and P. Fremantle. Multiprotocol Web Services for Enterprises and the Grid. In *Proceedings of the EuroWeb 2002 Conference on the Web and the Grid: From e-science to e-business*, Oxford, UK, December 2002.

[18] P. Narasimhan, L. Moser, and P. Mellior-Smith. Using Interceptors to enhance CORBA. *IEEE Computer*, 32(7):62–68, July 1999.

[19] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[20] E. Wohlstadter, S. Jackson, and P. T. Devanbu. DADO: Enhancing Middleware to Support Crosscutting Features in Distributed, Heterogeneous Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 174–186, 2003.