A Model-based Approach to Service-Oriented Computing

Jim Pruyne, Sharad Singhal

HP Laboratories
HPL-2008-174

Keyword(s):
Web Services, Middleware, Service Oriented Computing, Model-Driven Architecture

Abstract:
Web Services are the current best practice for developing distributed software and integrating disparate components across the Internet or within an enterprise. In this approach, services are characterized by their interface-what they can do, and their location-how they are accessed. We augment these characterizations with a service model-a definition of the state that a service exposes to the environment. The use of service models provides an attractive framework for describing services and leads to more structured service definitions, which in turn facilitates easier integration. Service models also provide a unique method for performing inter-service communication which is robust and resilient to failure because of its ability to guarantee consistency between run-time interchanges and the service model definition. We have prototyped the model-based system and validated the approach with both new and imported service definitions demonstrating the expressiveness and flexibility of the approach.

External Posting Date: October 29, 2008 [Fulltext] Approved for External Publication
Internal Posting Date: October 29, 2008 [Fulltext]

A Model-based Approach to Service-Oriented Computing

Jim Pruyne and Sharad Singhal, *Member, IEEE*

***Abstract*— Web Services are the current best practice for developing distributed software and integrating disparate components across the Internet or within an enterprise. In this approach, services are characterized by their interface—what they can do, and their location—how they are accessed. We augment these characterizations with a service model—a definition of the state that a service exposes to the environment. The use of service models provides an attractive framework for describing services and leads to more structured service definitions, which in turn facilitates easier integration. Service models also provide a unique method for performing inter-service communication which is robust and resilient to failure because of its ability to guarantee consistency between run-time interchanges and the service model definition. We have prototyped the model-based system and validated the approach with both new and imported service definitions demonstrating the expressiveness and flexibility of the approach.**

***Index Terms*— Web Services, Middleware, Service Oriented Computing, Model-Driven Architecture.**

I. Introduction

SERVICE oriented computing is based on the premise that all software or hardware resources can be accessed and utilized via a network. Each resource is considered a service, and clients only need to know the *interface* and *location* of the service to make use of it. The interface defines *what* the service can do, and usually takes the form of a set of operations and the mechanism for accessing the operations, such as standards based interface definitions (e.g., WSDL), or wire level input and output message formats. The location defines *how* the service is accessed, and typically takes the form of a network address and an identifier on the corresponding host that names the service. Commonly, the location takes the form of a URL.

The benefits of the service oriented computing are many. Services may be spread across multiple management domains, and can be augmented or improved without requiring changes from the clients as long as their interface is unchanged. Complex applications and composite services can be formed by orchestrating invocations to multiple services. Often, legacy applications can be fitted with service interfaces to bring them in to a service oriented environment. These advantages make service oriented approaches the best practice for building today's software, whether the clients span the entire Internet or a single enterprise's data center. However, the flexibility provided by web services can also lead to tremendous complexity. As new services are developed, they are often inconsistent with other existing services in their interface conventions. This, in turn, requires developers or service integrators to learn each service's conventions, and makes service composition more difficult and error prone.

We observe that systems management environments have faced similar integration problems. New devices, components, and concepts are routinely introduced in the environment, often after the management system has been developed and deployed. Model-based approaches provide a common solution to this problem of diversity. Examples include SNMP's MIBs and the Common Information Model (CIM) developed by the DMTF. In this approach, a management system defines a modeling methodology, and a model schema with a few core concepts. New elements can be introduced into the environment dynamically by creating additional models that follow the core concepts, typically without making significant changes to the management system itself.

We make two primary contributions in this paper. First, we describe how service descriptions can be augmented to include a service model that is exposed by the service. All services within a domain share a common base model and completely describe their external representation in the service model. The models extend the service interface by including the public state of the service. This state provides a common method of introspecting and interacting with all services, which aids in understanding and integrating new services as they are developed. Second, we extend the notion of a service end-point to a Service Access Point, a full service in its own right. We demonstrate that by delegating model management and inter-service communications to the Service Access Point, a more structured and powerful mechanism for service integration can be created in a service oriented environment.

The common shared service model as well example service models are provided in Section II. A model-based, inter-service communication pattern is described in Section III. In Section IV we describe the software architecture used in our implementation and Section V describes our experience using case studies. Related work is discussed in Section VI. We conclude with a summary and directions for future work in Section VII.

Manuscript received August 15, 2008.

Authors are with Hewlett Packard Laboratories, Palo Alto, CA 94304, USA. (Phone: 650-857-8310; email:pruyne@hp.com).

II. Model Based Concepts for Service Orientation

The purpose of modeling services is two-fold. First, models provide a clean and human readable notation for understanding the capabilities and purpose of a service. Second, once a concept is modeled, it is readily available for use by other services either as presented or via further customization or extension.

We build our service models using two existent and common methodologies: UML [1] and the modeling foundation developed for the Common Information Model (CIM) meta-model [2] by the DMTF. UML provides a visual, diagrammatic view of our models which makes them easy to quickly absorb and understand. The CIM meta-model provides a standards-based, robust modeling foundation including basic modeling concepts such as classes, properties, qualifiers, etc., as well as the Managed Object Format (MOF)—an easily readable and writeable syntax, and CIM-XML—an XML representation of the model useful for communicating and persisting the model.

A. The Core Model

We start by defining a few basic concepts used as the basis for all service models. These are shown in Figure 1. The root of the model is the class *Entity*. All other modeled elements inherit from *Entity*. *Entity* contains a unique Identifier (*Id*) such that every instance of *Entity* in the environment will be uniquely identified. *Entity* also contains the name and version of the class to which it conforms. Class definitions are captured in the *Type* class[1]. *Type* contains meta-information about the class including its name, author and version as well as the full class definition. We carry the class definition in its native format in the field *TypeDefinition* to support run-time introspection of the class definition by services that do not have prior information about the model exposed by a given service. For any class name and version pair, the definition is considered immutable.

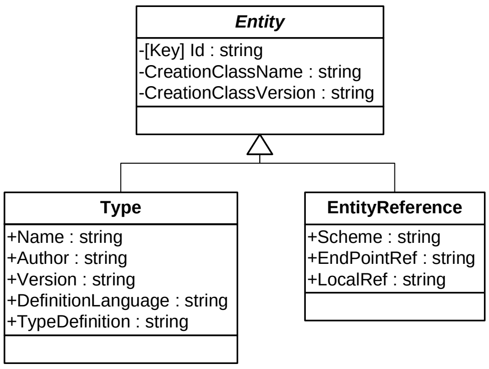

Figure 1 -- The Core Model

As discussed later in Section III, the run-time environment enforces that a class definition cannot be changed while existing instances of that class exist. This allows the system to evolve in predictable ways. Finally, *EntityReference* provides location information. It contains an access scheme or protocol (for example http), a network location (*EndPointRef*) and a local name (*LocalRef*) which is meaningful only at that particular location in the system. When services communicate about entities, they typically do so by reference by passing an *EntityReference* structure.

[1] The string "Class" is a reserved word in the MOF syntax, so our model uses "Type" to describe a class definition.

B. The Service Model

The core model applies to all entities in the service-oriented environment, but services themselves are further modeled as shown in Figure 2. A *Service* extends *Entity* (not shown), and defines a set of names by which it may commonly be referred. Thus, a single service can identify itself by as many names as may be needed to be understood or discovered by different clients. Services interact with the system by receiving messages via the *ReceiveIndication* operation. Each service is associated with zero or more instances of *Credential* and *IndicationFilter*. When the service is instantiated, credentials are associated with the service instance, for example, based on the credentials of the user or organization that owns the service. As the service interacts with other services, these credentials are passed in messages so that other services can perform authentication and authorization to ensure that operations initiated by this service are permitted. Every service also has associated with it a set of *IndicationFilter*s which specify what Indications or messages should be directed to that service. The *IndicationFilter* specifies the type of message, the operation, and the message target identity that this service wishes to receive. In this way, a single service can receive messages destined for many different modeled entities, and can act as an operational proxy for those entities. The complete handling of messages is discussed in more detail in the next section.

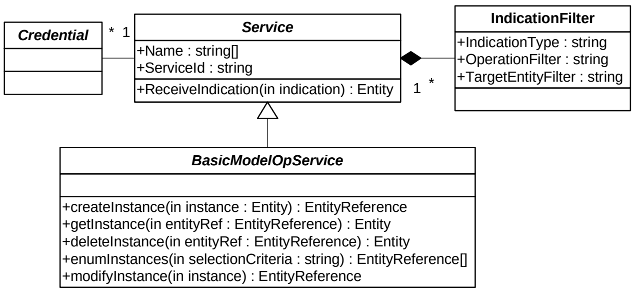
Credential
* 1
Service
+Name : string[]
+ServiceId : string
+ReceiveIndication(in indication) : Entity
1 *
IndicationFilter
+IndicationType : string
+OperationFilter : string
+TargetEntityFilter : string
BasicModelOpService
+createInstance(in instance : Entity) : EntityReference
+getInstance(in entityRef : EntityReference) : Entity
+deleteInstance(in entityRef : EntityReference) : Entity
+enumInstances(in selectionCriteria : string) : EntityReference[]
+modifyInstance(in instance) : EntityReference

Figure 2 -- The Service Model

The *BasicModelOpService* class encodes a common set of operations for manipulating modeled entities including creation, retrieval, deletion, discovery and modification. Note that because types are themselves defined as entities in the core model, they can be manipulated using the same operations. This enables new type definitions to be introduced dynamically at run-time. Because services expose their state using entities, it is a common pattern for them to support these or a sub-set of these operations on the entities which represents their state.

C. Mixing Multiple Service Interfaces

While we lean heavily on the CIM foundation, transitioning its use from systems management to service orientation has demonstrated a few shortcomings within it. Most notable has been its limited ability to support inheritance of service interfaces. Numerous service definition and programming languages have adopted the notion of interfaces as a first-class concept to enable a single service to mix-in pre-defined sets of operations. We accomplish this in CIM by introducing two new qualifiers: *Interface* and *Implements*. Qualifiers in a CIM model provide additional descriptive or meta-information about various parts of the model and new qualifiers may be defined as part of any model. Consider the following example in the MOF syntax:

```
[Interface, Abstract, Version(“1.0”)]
class Intf1 {
  boolean OperationA();
};
[Interface, Abstract, Version (“2.1”)]
class Intf2 {
  boolean OperationB();
};
[Implements(“Intf1:1.0, Intf2:2.1”),
 Version(“1.1”)]
class Impl1 : Service
{
};
```

Two classes, *Intf1* and *Intf2*, are defined with the *Interface* qualifier. To prevent direct instantiation of the classes, the qualifier *Abstract* is required for classes labeled with the *Interface* qualifier. Class *Impl1* defines a *Service* and has the *Implements* qualifier. It lists the names of the interface classes, and their corresponding version, separated by a colon. As noted above, the version is considered to be part of the class identifier, so it needs to be included in the *Implements* statement.

These qualifiers are used both at registration and during service interactions to assure proper operation of the service. At registration time, the existence of the interface classes is checked, and checks for operation name collision are performed. If two interface classes define operations with the same name, the class definition is rejected, because CIM forbids overloading of operations, even when parameters differ. As will be shown later, during message processing, we validate correspondence between messages and service definitions, and the implemented interfaces are included in a service’s available operations.

D. *Extending the Service Model*

An important goal of our approach to service orientation is to make concepts re-useable. One approach to this is to extend the models we provide with new concepts that can be applied to a variety of services. As an example, we consider a service that supports Service Level Agreements (SLAs) as shown in Figure 3.

SLABasedService extends *Service* (not shown), and implements two additional interfaces that are related to SLA management and negotiation. The *SLABasedServiceMgmtIF* provides operations for querying the service about SLAs currently in operation and the templates for SLAs that the service expects it may be able to support for new clients.

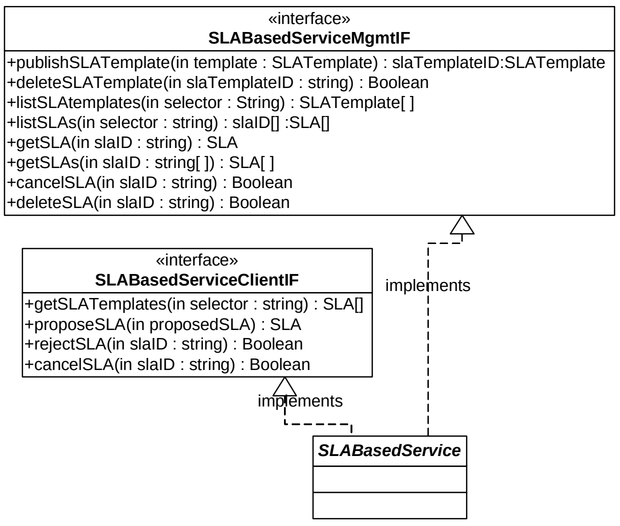
«interface»
SLABasedServiceMgmtIF
+publishSLATemplate(in template : SLATemplate) : slaTemplateID:SLATemplate
+deleteSLATemplate(in slaTemplateID : string) : Boolean
+listSLAtemplates(in selector : String) : SLATemplate[ ]
+listSLAs(in selector : string) : slaID[] :SLA[]
+getSLA(in slaID : string) : SLA
+getSLAs(in slaID : string[ ]) : SLA[ ]
+cancelSLA(in slaID : string) : Boolean
+deleteSLA(in slaID : string) : Boolean
«interface»
SLABasedServiceClientIF
+getSLATemplates(in selector : string) : SLA[]
+proposeSLA(in proposedSLA) : SLA
+rejectSLA(in slaID : string) : Boolean
+cancelSLA(in slaID : string) : Boolean
implements
implements
SLABasedService

Figure 3 -- The SLA Service Model

The *SLABasedServiceClientIF* is used by a client for negotiating and managing an instance of a SLA. Each of these operates on a modeled *Entity* called simply an *SLA*. The *SLA* is a complex structure which is identified by a single identifier, but which also contains a set of terms indicating what conditions the SLA imposes on all of the parties and what the payment policies and conditions are when the SLA is satisfied or violated. Further discussion of the particular SLA model is provided in [3]. We show the use of the SLA model in a case study in Section V. This example shows that common principles can be modeled once, and then used or extended to match the needs of new services.

E. *A Complete Service Example*

To provide an example of a complete service model, we describe one of our “foundation” services. In any distributed or service-oriented environment, there is a need for some basic, supporting services that are used by all services. The foundation services we have defined include a naming service for distributed service discovery, a certificate authority for issuing and validating credentials, and a dictionary service for distributed registration and lookup of model types. While the first two, naming and certificate authorities, are common to many web-services environments, the dictionary service is unique to our model based environment, so we focus on it here.

The service model for the dictionary service is shown in Figure 4. The model consists of two entities: a *TypeFilter* and the *DictionaryService* definition itself. The *DictionaryService* defines operations for registering, deleting and looking up instances of the class *Type*. The delete and lookup operations identify the *Type* classes of interest based on the name and version tuple. As discussed previously, we allow multiple versions of a single type to be present, so both values must be provided when identifying an instance of *Type* in the dictionary. Also, there is no update operation because once inserted into the dictionary, the *Type* definition is considered immutable. The *filteredLookup* operation provides finer grain access during lookups based on the *TypeFilter* class defined previously. This allows lookups to match any of the fields given in the provided *TypeFilter* instance, such as the Author. The service specification also allows for the values in the *TypeFilter* to include wildcards or regular expressions for more flexible matching. This gives an example of how a service can create supporting types which are accessible to clients and can be used for more complex interactions. The last operation, *evalCompatibility*, exposes the dictionary's ability to automatically determine compatibility between two types. Compatibility assessment is valuable as a system evolves because it determines when a service or type definition may safely be interchanged for another version. A complete description of the use of compatibility and how it is determined in the dictionary service is given in [4].

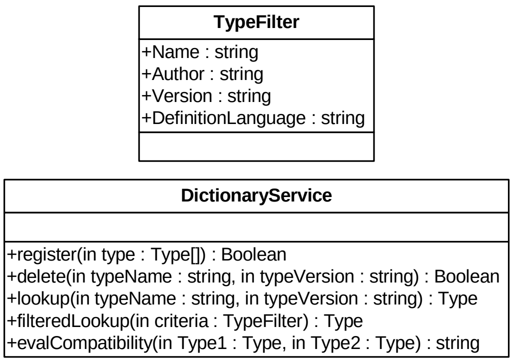
TypeFilter
+Name : string
+Author : string
+Version : string
+DefinitionLanguage : string
DictionaryService
+register(in type : Type[]) : Boolean
+delete(in typeName : string, in typeVersion : string) : Boolean
+lookup(in typeName : string, in typeVersion : string) : Type
+filteredLookup(in criteria : TypeFilter) : Type
+evalCompatibility(in Type1 : Type, in Type2 : Type) : string

Figure 4 -- The Dictionary Service Model

III. Service Interactions

To discuss how services interact with one another using service models, it is useful to start with a description of some of the components in the environment, and their roles. The components and their relationships are shown in Figure 5.

The focus of all activity in the environment is the *Service* as represented by the circles in the diagram. All communicating entities are services. Services may represent human interactions as shown in the top right corner, or may represent computational or data resources as shown in the diagram with the corresponding icons. As indicated in Section II.B, services may expose other entities (e.g., computers, virtual machines, service level agreements, or software) as part of their service models, and enable operations on those entities by clients.

Because these operations (and the corresponding models) can become very complex, we introduce a key service in the environment to manage these interactions on behalf of other services: the *Service Access Point* (abbreviated SAP in the diagram). The SAP service extends the notion of a web services end-point, and performs two essential roles. First, it acts as an intermediary for all model-related communication between services using a specific interaction pattern discussed below. In this role, it insulates all service implementations from the details of model representations, communication protocols, or message encodings necessary for transmitting models between services. Second, it provides a repository for all models exposed by services associated with it.

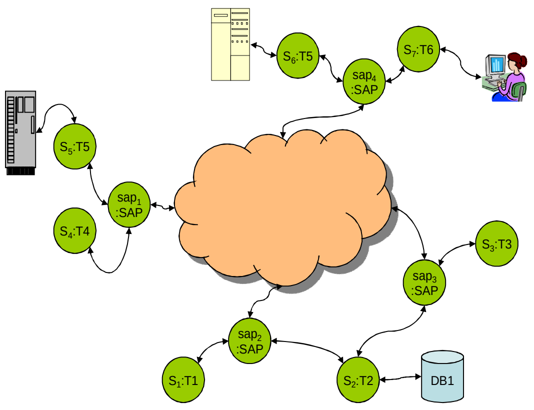

Figure 5 -- Actors in the Services Environment

As shown in the figure, a single Service Access Point may host many services, and a single service may communicate via multiple Service Access Points. Thus, services and their Service Access Points have a many-to-many relationship. This permits services to gain a degree of redundancy by registering in multiple locations, and allows the system to scale by hosting multiple services at a single Service Access Point sharing a single network endpoint and model repository.

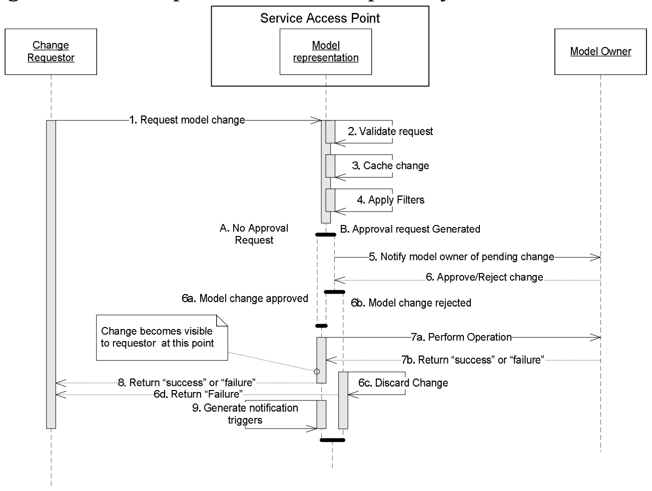
Service Access Point
Change Requestor
Model representation
Model Owner
1. Request model change
2. Validate request
3. Cache change
4. Apply Filters
A. No Approval Request
B. Approval request Generated
5. Notify model owner of pending change
6. Approve/Reject change
6a. Model change approved
6b. Model change rejected
Change becomes visible to requestor at this point
7a. Perform Operation
7b. Return "success" or "failure"
8. Return "success" or "failure"
6d. Return "Failure"
6c. Discard Change
9. Generate notification triggers

Figure 6 -- The model exchange pattern.

The Service Access Point mediates communication among services, and in so doing utilizes the service models to insure well-formed communication and resiliancy to failures. The communication pattern provided by the Service Access Point is encapsulated in what we call the model exchange pattern, which is shown in Figure 6.

We consider all service-to-service communications to be requests for changes in the service model of the target service. This is clear when we consider operations such as *modifyInstance* described in the last section, but the same pattern applies for an operation defined by any modeled entity because all operations ultimately may reflect a change in the exposed state of a service or its associated entities. In the Figure, we refer to the service client which initiates the request for the operation as the *Change Requestor*, and the *Model Owner* as the target service for carrying out the request.

As shown in the Figure, the request is mediated by the Service Access Point, which holds the service model of the target service. When the Service Access Point receives the operation request (1), it validates the request by comparing it against the type and state of the modeled entity (2). If the request is malformed, or refers to an operation or attribute in the model that is not defined, the request is rejected immediately. Next, the Service Access Point snapshots (or caches) the current state of the target model to allow roll-back in case of errors (3).

Filters are used to match the target of the operation with its corresponding owner (4). If the model owner wishes to perform explicit approval of an operation prior to the operation being performed, as denoted by an appropriate *IndicationFilter*, step (5) is performed, else the operation can be performed at step (7). The approval request comes to the Model Owner service via its *receiveIndication* operation defined earlier, but the payload indicates that this indication is for approval rather than execution.

The Service Access Point then forwards the request to the service (7a). As part of this interaction, the service performs the operation and updates the model at the Service Access Point. Message processing is completed by returning the result to the requestor (8) or rolling back and returning failure in the case of an error (6c) followed by completion events being sent (9) to any services requesting notification of model changes via an appropriate *IndicationFilter*.

This interaction pattern provides the following advantages over other protocols:

Message Validation: The pattern provides run-time type-checking, which is done prior to the target service being invoked, thus simplifying the handling of errors during service operations. In addition, the credentials of the change requestor are checked during validation to authenticate the identity of the requestor.

Failure Semantics: The pattern provides a well defined failure semantics in case of errors, and provides robustness by providing a mechanism to roll-back the service model to its original state in case of error. This avoids a common problem with model based systems, namely that the stored model state becomes inconsistent with the environment, particularly when operations fail.

Approvals: The pattern clearly separates out the procssing necessary to decide if a request has been approved from the actual handling of the operation itself. By separating approvals from performing the operation, the Service Access Point allows approvals and processing to be performed by separate services in a services oriented environment. Since the Model Owner is allowed to inspect the pending request as part of the approval, it can decide if approval should be granted based on the current state of the Service Model, the identity and/or the role of the requestor, and any other internal policies it has governing updates to its model.

Event Notifications: Using filters at steps (4) and (9), a publish-subscribe mechanism can be created that allows other services to be notified of changes occuring in service models within the services oriented environment.

Because the model owner (typically the service exposing the model) stays in control of the exposed model at all times, it can decide which model changes require approval and/or notification. It can also analyze pending changes to ensure model coherence. Finally, such a handshake provides assurance to the clients that any changes they make in the models are valid changes, and receive explicit notification when the changes fail.

IV. System Realization

The modeling infrastructure and particularly the model exchange pattern provide a highly functional environment with many desirable properties, but make a challenging target to realize due to the many steps involved in processing messages, and the requirement to manage communication and a model repository. In addition, our implementation delegates all network communication to the Service Access Point. Services view *EntityReferences* as opaque references, and rely upon the Service Access Point to determine how and where it should communicate to perform operations. Because the Service Access Point is an active entity in the system, it is dynamic in its interactions with other services. That is, as services start or stop, the Service Access Point is aware of these life-cycle based changes and accommodates them without itself having to shut-down or re-start. These challenges and requirements have led us to specific architecture and technology choices. The resulting system block diagram is shown in Figure 7.

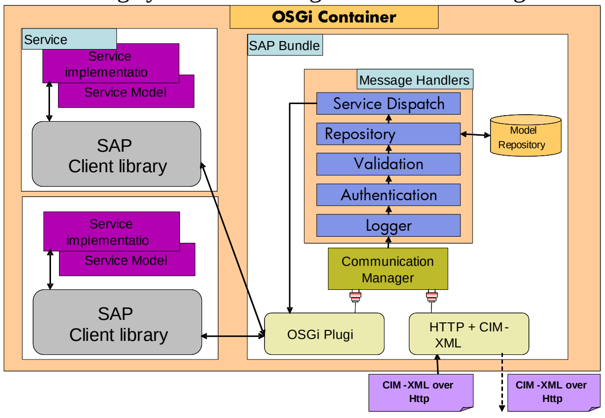

Figure 7 – System Block Diagram

A. *Technology Choices*

Our first choice in implementation was to use the Open Services Gateway Initiative (OSGi) [5] framework. OSGi provides a "bundle" abstraction for services: a self-contained collection of Java-code whose life-cycle can be managed independently of other bundles via a programmatic interface. In the Figure, we show a single Java Virtual Machine, labeled "OSGi Container" hosting both the Service Access Point bundle and multiple service bundles. OSGi allows bundles to selectively expose portions of their implementation to other bundles while providing isolation and security for other parts of the code. The Service Access Point uses these capabilities to export a client library which other services use for interacting with it. Command-line and web interfaces are available for managing bundles within an OSGi container, and we use these interfaces for installing, upgrading and removing services from operation without affecting the operation of other services. The creation of OSGi bundles is simplified by the use of developer tools found in the Eclipse plug-in development environment [6] which is also based on OSGi. Thus, service developers benefit from all wizards and other tools available to Eclipse plug-in developers. Use of this mature tooling greatly simplifies the process of packaging and deploying new services.

B. Communication and Message Handling

The number of steps involved in the model exchange pattern, and the desire to support multiple communication protocols led to a very modular approach to building the Service Access Point. The hub is the Communication Manager component. The Communication Manager has the responsibility of loading the various modules and orchestrating messages. Messages enter the Service Access Point through one of the communication plug-ins, which handle all in-bound and out-bound communication for the Service Access Point. When a plug-in receives a message, it passes it on to the Communication Manager for local processing, which inspects the destination *EntityReference* to determine whether the message should be handled locally or sent to a remote, peer Service Access Point, and if so, through which communication plug-in. The plug-ins to be installed are specified by a configuration file for the Service Access Point.

We have developed two plug-ins. The first handles communication within the OSGi container and provides the counterpart to the SAP client library used by services sharing the same OSGi container. The second plug-in uses the CIM-XML [7] format defined by the DMTF WBEM standard for all remote communication. This plug-in serializes and de-serializes messages in the CIM-XML format and sends and receives them using HTTP. The architecture permits other wire protocols (e.g., SOAP) to be handled similarly.

When a message is to be handled locally, the Communication Manager passes it through a series of Message Handlers. The set of message handlers is loaded dynamically as are the communication plug-ins. Each handler processes the message in sequence. In a typical installation, we have handlers which validate messages, perform authentication, generate model-related events and dispatch messages to service implementations using the steps defined in the model exchange pattern.

C. Model Repository

The final component of the Service Access Point is the model repository. The repository implements the operations defined in the *BasicModelOpService* described previously. Because each Entity to be stored or retrieved contains a unique identifier, we use a transactional key-value store system rather than a full relational database. For store and retrieve operations, this provides excellent performance with less implementation complexity because it does not require object-relational mappings. The transactional features of the store are used to perform the required caching, commit and rollback operations as defined in the model exchange pattern, as well as ensuring that concurrent accesses to the Service Access Point do not result in inconsistent results.

V. Experience

Two large services have been built on this environment, both in the area of provisioning. The first installs and configures a Microsoft Exchange e-mail server environment using bare-metal blades in an on-demand service environment. The provisioning service has been modeled to allow service users to request different sized installations of the Exchange server in terms of number of user mailboxes and mailbox sizes. The service supports Service Level Agreement (SLA) negotiation between the user and the service provider prior to deployment. The negotiation includes attributes such as mailbox size, number of users, server location, and price. Once the SLA agreement is reached, the service deploys the appropriate Exchange configuration and provides access to the requestor.

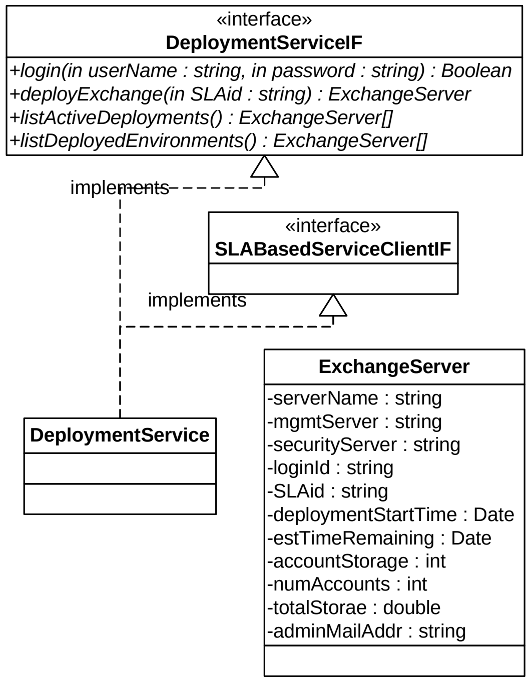

Figure 8 -- Exchange Deployment Service

The service model used in the service is shown in Figure 8. Note that the *DeploymentService* implements both the *SLABasedServiceClientIF* described previously and a new interface specific to this service. The service also models the state of a particular deployment in the type *ExchangeServer* and makes it available to the user. This type provides all of the deployment parameters as well as information about the progress of the deployment using the *deploymentStartTime* and *estTimeRemaining* properties. In this way, a client can query the state of their deployment by directly inspecting the model exposed by the Exchange service and without further interacting with the *DeploymentService*. Events are triggered when the deployment state reaches a desired value so that the user can be notified of completion. The service is available as a proof-of-concept [8] to HP's customers.

The second service provisions compute intensive jobs in to a batch scheduling system, and is part of an on-going initiative between the A*STAR Institute of High Performance Computing of Singapore and HP. The service models encapsulate all of the parameters necessary for submitting batch jobs to a high-performance computing cluster, and for monitoring the jobs throughout their lifecycle. After submitting a job, a user can monitor the state of the job by monitoring the *Entity* corresponding to the job, and ultimately receiving events directly from the Service Access Point as the job changes state. The use of events here has blended well with the service oriented approach, because the client does not need to poll the system to determine when a job completes. The events let them know as jobs are completed.

We also use the batch execution service as our benchmarking test. The service creates an SLA template that is composed of a complex structure, so it performs a large number of model interactions with the repository at the Service Access Point during service initialization. To understand the impact of the various steps in the model exchange pattern, we have enabled and disabled various steps using appropriate message handlers described previously. Results are shown in Table 1.

Table 1 -- Time to initialize batch processing service

Number of Operations	No Message signing	Messages signed and Validated	Access Control
1454	11 sec.	27 sec.	80 sec.

The table shows that during initialization, 1454 distinct service interactions take place, mostly to the Service Access Point within a single OSGi container. Typically, these require the repository to persist modeled entities that compose an SLA. The time increases as we introduce various authorization steps. In the first column, no authentication or authorization is possible because credential information and message signing is not performed. This provides the baseline for comparison. Column two shows the time required when messages are signed and authenticated by the Service Access Point. Column three adds authorization checks performed by a service outside the Service Access Point. We see that, overall with complete processing in place, we require about 55 milliseconds per service interaction within a local OSGi container. Much of this, about 36 milliseconds, occurs during access control checking, and this area may have opportunities for further optimization.

A. *Web Service Interoperability*

As part of developing these services, it became obvious that a major concern in such an environment is to make use of services that have already been created using other services infrastructures such as web-services. In both cases we experimented with, model-based communication was used to augment and expand the capabilities of other back-end services. In addition, adding model-based mechanisms to a services oriented environment requires us to inter-operate with the very large number of existing web-services. Our approach to addressing this has been two-fold. First, we must understand if our modeling capability is sufficient to capture all of the capability of other service definition approaches. To this end, we developed a translator that converts WSDL [9] representations of service interfaces in to service models specified in MOF. We found that the conversion is possible, but there are some significant differences between the expressiveness of WSDL and MOF. In particular, WSDL uses an XML document exchange pattern which encourages structures to be nested within one another, but does not easily support object oriented concepts such as inheritance and associations. Conversely, CIM-based modeling approach does not easily support nesting of data types. Instead, CIM encourages types to be defined separately and related via associations. However, we have successfully transformed the nesting style to one with many classes and corresponding associations. This has permitted us to proceed to the second phase of the experiment: building a bridge or proxy service which allows WSDL-based services to be automatically invoked from within our environment. This experiment is on-going at the time of writing, but because the translation has been completed, bridging appears feasible. Thus we are encouraged that the model based approach is at least as capable as the WSDL-based document approach which is commonly used today.

VI. Related Work

Service-oriented computing continues to evolve rapidly. New standards, specifications and middleware are being released frequently. In the standards community, the approach that most closely relates to ours is the WS-Resource Framework (WSRF) [10]. In WSRF, a service exposes state, referred to as a WS-Resource, in the form of an XML document. Operations can be directed to the resource to add, delete and change the set of properties on the resource. In contrast to our approach, WSRF does not specify message handling protocols to assure consistency of resource values, and does not provide an infrastructure with sufficient run-time information to perform the sort of message validation we do. Implementations of WSRF, such as Globus [11] often do provide built-in support for managing the resources to alleviate the burden on a developer, but they typically do not provide persistence or transactional behavior.

Another service framework is e-speak. E-Speak describes services using vocabularies and contracts which are similar to our models. Vocabularies represent a set of names and typed properties for any entity in the system, and are used to describe a service interface in a contract. Like the Service Access Point in our architecture, E-Speak also creates an intermediary for communication among services, and provides support for events based on service state change. However, E-Speak does not have any form of model-exchange pattern to ensure robustness of interactions and the descriptiveness of the vocabularies is limited compared to the full object-oriented modeling we inherit from CIM.

The CIM standards provide a protocol for remotely accessing models via the WBEM interface. These can be accessed in a REST model [13] or using SOAP messages as found in other web services specifications [14]. We borrow heavily from the WBEM specifications, particularly in terms of the XML serialized representation of CIM objects. However, the WBEM specifications stop short of defining a general, service-oriented approach, and provide only a means of accessing and updating models stored at an object manager. There is no specific notion of services as directly communicating entities.

Another class of systems which has some similarities to our work is the Enterprise Service Bus (ESB). There are a large number of ESB systems including open source systems like ServiceMix [15], Mule [16] or JBossESB [17] to long-standing commercial offerings such as Tibco [18]. These systems typically focus on the communication aspect of a service oriented environment. That is, isolating service implementations from the variety of protocols and transport layers. In this sense, they may provide a useful foundation for further development of our system, though none currently provide direct support for the CIM-XML transport we prefer. At higher layers, these systems often provide support for workflow, choreography and business process modeling. We do not explicitly support work in these areas as we're striving to demonstrate the implicit choreography achievable by generating events on service models as they change.

VII. Conclusions and Future Work

Service oriented computing is still an emerging paradigm though it is backed by years of work on distributed system middleware. Most of today's approaches to service orientation have built upon existent web technologies, protocols and software stacks to accommodate service-to-service interactions. We extend the foundation technologies to include those used in building model-based management and automation systems by adding the notion of modeled elements to the representation of each service. This in turn permits us to have highly descriptive representations of the service, and develop more robust communication protocols without significantly burdening the service developer. Our experience has shown that this approach can be applied to many types of services, including those currently using other service interface specifications.

One of the most significant barriers to success for service orientation is for services to be easily discoverable, and to present abstractions in their interfaces desired by service consumers. Service models provide a very clear definition of services, but the issue of suitability for services remains. To alleviate this issue, we are working to leverage today's popular collaboration environments to include service definition aspects so that service definition can be a group effort leading to a more usable service. We are also working to further the transparent distribution of models. The model based approach appears to provide a means of breaking the link to physical locations resulting in a more dynamic, robust and scalable system.

Acknowledgements

The work described here is the result of contributions of a great many people in development of the platform and the services that run on it. Contributors to this work included Karin Becker, Andre Lopes, Subu Iyer, John Wilkes, Arthur Fong, Rully Santosa, Hao Ming Zhong, Datta Hassan, Ravi Gullapalli, Nandan Reddy of HP and Yong Siang Foo of Singapore A*STAR.

References

[1] J. Rumbaugh, I. Jacobson and G. Booch, "The Unified Modeling Language Reference Manual", Addison-Wesley Professional (January 2, 1999).

[2] Distributed Management Task Force, "Common Information Model (CIM) Infrastructure Specification", DSP0004, version 2.3 Final, October 4, 2005.

[3] S. Iyer et. al., "SLA Based Service Module and SLA Module," Hewlett-Packard Laboratories Technical Report.

[4] K. Becker et. al., "Automatically Determining Compatibility of Evolving Services," to appear in *International Conference on Web Services (ICWS)*, September 2008.

[5] OSGi Alliance, "About the OSGi Service Platform," http://osgi.org/documents/collateral/OSGiTechnicalWhitePaper.pdf.

[6] W. Melhem and D. Glozic, "PDE Does Plug-Ins," http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html.

[7] Distributed Management Task Force, "Representation of CIM in XML", DSP0201, version 2.2 Final, February 8, 2007.

[8] Hewlett-Packard Company, "See AI in action," http://h20324.www2.hp.com/hpsdp/ib/IB_Entry.jsp?company_id=5040462.

[9] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, "Web Services Description Language (WSDL) 1.1," W3C Note 15 March 2001.

[10] K. Czajkowski, et. al., "The WS-Resource Framework", March 5, 2004. Available from http://www.globus.org/wsrf/specs/ws-wsrf.pdf.

[11] I. Foster**, "**Globus Toolkit Version 4: Software for Service-Oriented Systems," in *IFIP International Conference on Network and Parallel Computing*, Springer-Verlag LNCS 3779, pp 2-13, 2006.

[12] Hewlett-Packard Company, "E-speak Archectural Specification," Release A.0, January 2001.

[13] Distributed Management Task Force, "CIM Operations over HTTP", DSP0200, version 1.2 Final, February 8, 2007.

[14] Distributed Management Task Force, "Web Services for Management", DSP0226, version 1.0.0 Final, February 12, 2008.

[15] Apache Software Foundation, "Apache ServiceMix 3.x Users' Guide," http://servicemix.apache.org/users-guide.html.

[16] Mule Development Team, "Mule 2.x User Guide,"http://www.mulesource.org/display/MULE2USER/Home.

[17] JBoss ESB, http://www.jboss.org/jbossesb.

[18] Tibco Company, "TIBCO ActiveMatrix ServiceBus," http://www.tibco.com/software/soa/activematrix_service_bus/default.jsp.