Web Services Architecture

W3C Working Group Note 11 February 2004

This version:
http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/
Latest version:
http://www.w3.org/TR/ws-arch/
Previous version:
http://www.w3.org/TR/2003/WD-ws-arch-20030808/
Editors:
David Booth, W3C Fellow / Hewlett-Packard
Hugo Haas, W3C
Francis McCabe, Fujitsu Labs of America
Eric Newcomer (until October 2003), Iona
Michael Champion (until March 2003), Software AG
Chris Ferris (until March 2003), IBM
David Orchard (until March 2003), BEA Systems

This document is also available in these non-normative formats: PostScript version and PDF version.

Copyright © 2004 W3C® (MIT, ERCIM, Keio), All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply.

Abstract

This document defines the Web Services Architecture. It identifies the functional components and defines the relationships among those components to effect the desired properties of the overall architecture.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This is a public Working Group Note produced by the W3C Web Services Architecture Working Group, which is part of the W3C Web Services Activity. This publication as a Working Group Note coincides with the end of the Working Group's charter period, and represents the culmination of the group's work.

Discussion of this document is invited on the public mailing list www-ws-arch@w3.org (public archives). A list of remaining open issues is included in **4 Conclusions** [p.92] .

Patent disclosures relevant to this specification may be found on the Working Group's patent disclosure page.

Publication as a Working Group Note does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress. Other documents may supersede this document.

Table of Contents

Appendices

---

1 Introduction

1.1 Purpose of the Web Service Architecture

Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. This document (WSA) is intended to provide a common definition of a Web service, and define its place within a larger Web services framework to guide the community. The WSA provides a conceptual model and a context for understanding Web services and the relationships between the components of this model.

The architecture does not attempt to specify how Web services are implemented, and imposes no restriction on how Web services might be combined. The WSA describes both the minimal characteristics that are common to all Web services, and a number of characteristics that are needed by many, but not all, Web services.

The Web services architecture is an *interoperability* architecture: it identifies those global elements of the global Web services network that are required in order to ensure interoperability between Web services.

1.2 Intended Audience

This document is intended for a diverse audience. Expected readers include Web service specification authors, creators of Web service software, people making decisions about Web service technologies, and others.

1.3 Document Organization

This document has two main sections: a core concepts section (**2 Concepts and Relationships** [p.9]) and a stakeholder's perspectives section (**3 Stakeholder's Perspectives** [p.60]).

2 Concepts and Relationships [p.9] provides the bulk of the conceptual model on which conformance constraints could be based. For example, the resource [p.48] concept states that resources have identifiers (in fact they have URIs). Using this assertion as a basis, we can assess conformance to the architecture of a particular resource by looking for its identifier. If, in a given instance of this architecture, a resource has no identifier, then it is not a valid instance of the architecture.

While the concepts and relationships [p.9] represent an enumeration of the architecture, the stakeholders' perspectives [p.60] approaches from a different viewpoint: how the architecture meets the goals and requirements. In this section we elucidate the more global properties of the architecture and demonstrate how the concepts [p.12] actually achieve important objectives.

A primary goal of the Stakeholder's Perspectives [p.60] section is to provide a top-down view of the architecture from various perspectives. For example, in the **3.6 Web Services Security** [p.77] section we show how the security of Web services is addressed within the architecture. The aim here is to demonstrate that Web services can be made secure and indicate which key concepts and features of the architecture achieve that goal.

The key stakeholder's perspectives supported in this document reflect the major goals of the architecture itself: interopability, extensibility, security, Web integration, implementation and manageability.

1.4 What is a Web service?

For the purpose of this Working Group and this architecture, and without prejudice toward other definitions, we will use the following definition:

[Definition: A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.]

1.4.1 Agents and Services

A Web service [p.37] is an abstract notion that must be implemented by a concrete agent [p.31] . (See Figure 1-1 [p.9]) The agent is the concrete piece of software or hardware that sends and receives messages [p.19] , while the service is the resource characterized by the abstract set of functionality that is provided. To illustrate this distinction, you might implement a particular Web service using one agent one day (perhaps written in one programming language), and a different agent the next day (perhaps written in a different programming language) with the same functionality. Although the agent may have changed, the Web service remains the same.

1.4.2 Requesters and Providers

The purpose of a Web service is to provide some functionality on behalf of its owner -- a person or organization [p.55] , such as a business or an individual. The *provider entity* is the person or organization [p.55] that provides an appropriate agent to implement a particular service. (See Figure 1-1 [p.9] : Basic Architectural Roles.)

A *requester entity* is a person or organization [p.55] that wishes to make use of a provider entity's Web service. It will use a *requester agent* to exchange messages with the provider entity's *provider agent*.

(In most cases, the requester agent is the one to initiate this message exchange, though not always. Nonetheless, for consistency we still use the term "requester agent" for the agent that interacts with the provider agent, even in cases when the provider agent actually initiates the exchange.)

Note:

A word on terminology: Many documents use the term service provider to refer to the provider entity and/or provider agent. Similarly, they may use the term service requester to refer to the requester entity and/or requester agent. However, since these terms are ambiguous -- sometimes referring to the agent [p.31] and sometimes to the person or organization [p.55] that owns the agent -- this document prefers the terms *requester entity*, *provider entity*, *requester agent* and *provider agent*.

In order for this message exchange to be successful, the requester entity and the provider entity must first agree [p.67] on both the semantics and the mechanics of the message exchange. (This is a slight simplification that will be explained further in **3.3 Using Web Services** [p.66] .)

1.4.3 Service Description

The mechanics of the message exchange are documented in a Web service description [p.39] (WSD). (See Figure 1-1 [p.9]) The WSD is a machine-processable specification of the Web service's interface, written in WSDL. It defines the message formats, datatypes, transport protocols, and transport serialization formats that should be used between the requester agent and the provider agent. It also specifies one or more network locations at which a provider agent can be invoked, and may provide some information about the message exchange pattern that is expected. In essence, the service description represents an agreement [p.67] governing the mechanics of interacting with that service. (Again this is a slight simplification that will be explained further in **3.3 Using Web Services** [p.66] .)

1.4.4 Semantics

The semantics [p.42] of a Web service is the shared expectation about the behavior of the service, in particular in response to messages that are sent to it. In effect, this is the "contract" between the requester entity and the provider entity regarding the purpose and consequences of the interaction. Although this contract represents the overall agreement between the requester entity and the provider entity on how and why their respective agents will interact, it is not necessarily written or explicitly negotiated. It may be explicit or implicit, oral or written, machine processable or human oriented, and it may be a legal agreement or an informal (non-legal) agreement [p.67] . (Once again this is a slight simplification that will be explained further in **3.3 Using Web Services** [p.66] .)

While the service description represents a contract governing the mechanics of interacting with a particular service, the semantics represents a contract governing the meaning and purpose of that interaction. The dividing line between these two is not necessarily rigid. As more semantically rich languages are used to describe the mechanics of the interaction, more of the essential information may migrate from the informal semantics to the service description. As this migration occurs, more of the work required to achieve successful interaction can be automated.

1.4.5 Overview of Engaging a Web Service

There are many ways that a requester entity might engage and use a Web service. In general, the following broad steps are required, as illustrated in Figure 1-1 [p.9] : (1) the requester and provider entities become known to each other (or at least one becomes know to the other); (2) the requester and provider entities somehow agree [p.67] on the service description and semantics that will govern the interaction between the requester and provider agents; (3) the service description and semantics are realized by the requester and provider agents; and (4) the requester and provider agents exchange messages, thus performing some task on behalf of the requester and provider entities. (I.e., the exchange of messages with the provider agent represents the concrete manifestation of interacting with the provider entity's Web service.) These steps are explained in more detail in **3.4 Web Service Discovery** [p.68] . Some of these steps may be automated, others may be performed manually.

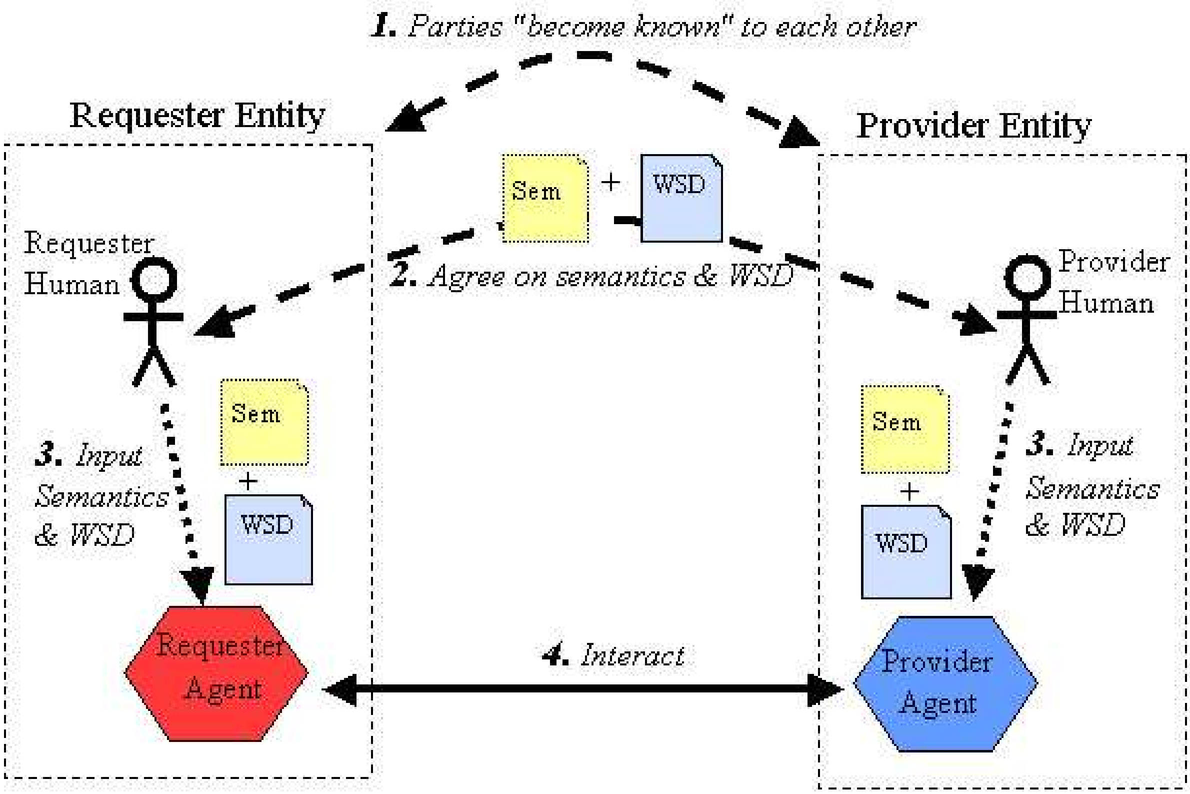

Figure 1-1. The General Process of Engaging a Web Service

1.5 Related Documents

The Working Group produced the following companion documents in the process of defining this architecture:

- Requirements Document [WSA Reqs] [p.96]
- Usage Scenarios [WSAUS] [p.96]
- Glossary [WS Glossary] [p.96]
- OWL Ontology [OWLO] [p.96]

2 Concepts and Relationships

2.1 Introduction

The formal core of the architecture is this enumeration of the concepts and relationships that are central to Web services' interoperability.

2.2 How to read this section

The architecture is described in terms of a few simple elements: concepts, relationships and models. Concepts are often noun-like in that they identify things or properties that we expect to see in realizations of the architecture, similarly relationships are normally linguistically verbs.

As with any large-scale effort, it is often necessary to structure the architecture itself. We do this with the larger-scale meta-concept of model [p.11] . A model is a coherent portion of the architecture that focuses on a particular theme or aspect of the architecture.

2.2.1 Concepts

A concept is expected to have some correspondence with any realizations of the architecture. For example, the message [p.19] concept identifies a class of object (not to be confused with Objects and Classes as are found in Object Oriented Programming languages) that we expect to be able to identify in any Web services context. The precise form of a message may be different in different realizations, but the message [p.19] concept tells us what to look for in a given concrete system rather than prescribing its precise form.

Not all concepts will have a realization in terms of data objects or structures occurring in computers or communications devices; for example the person or organization [p.55] refers to people and human organizations. Other concepts are more abstract still; for example, message reliability [p.85] denotes a property of the message transport service — a property that cannot be touched but nonetheless is important to Web services.

Each concept is presented in a regular, stylized way consisting of a short definition, an enumeration of the relationships with other concepts, and a slightly longer explanatory description. For example, the concept of agent [p.31] includes as relating concepts the fact that an agent is a [p.57] computational resource, has an identifier [p.59] and an owner. The description part of the agent [p.31] explains in more detail why agents are important to the archicture.

2.2.2 Relationships

Relationships denote associations between concepts. Grammatically, relationships are verbs; or more accurately, predicates. A statement of a relationship typically takes the form: concept predicate concept. For example, in agent [p.31] , we state that:

An agent is [p.57]

> a computational resource

This statement makes an assertion, in this case about the nature of agents. Many such statements are descriptive, others are definitive:

A message has [p.59]

a message sender [p.27]

Such a statement makes an assertion about valid instances of the architecture: we expect to be able to identify the message sender in any realization of the architecture. Conversely, any system for which we cannot identify the sender of a message is not conformant to the architecture. Even if a service is used anonymously, the sender has an identifier but it is not possible to associate this identifier with an actual person or organization.

2.2.3 Concept Maps

Many of the concepts in the architecture are illustrated with *concept maps*. A concept map is an informal, graphical way to illustrate key concepts and relationships. For example the diagram:

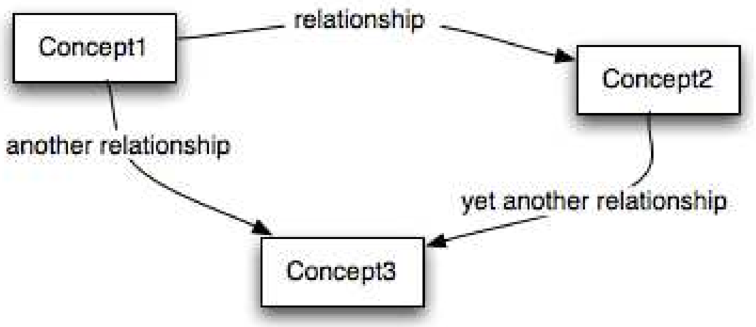

Figure 2-1. Concept Map

shows three concepts which are related in various ways. Each box represents a concept, and each arrow (or labeled arc) represents a relationship.

The merit of a concept map is that it allows rapid navigation of the key concepts and illustrates how they relate to each other. It should be stressed however that these diagrams are primarily navigational aids; the written text is the definitive source.

2.2.4 Model

A model is a coherent subset of the architecture that typically revolves around a particular aspect of the overall architecture. Although different models share concepts, it is usually from different points of view; the major role of a model is to explain and encapsulate a significant theme within the overall Web services architecture.

For example, the Message Oriented Model [p.17] focuses and explains Web services strictly from a message passing perspective. In particular, it does not attempt to relate messages to services provided. The Service Oriented Model [p.29] , however, lays on top of and extends the Message Oriented Model in order to explain the fundamental concepts involved in service - in effect to explain the purpose of the messages in the Message Oriented Model.

Each model is described separately below, in terms of the concepts and relationships inherent to the model. The ordering of the concepts in each model section is alphabetical; this should not be understood to imply any relative importance. For a more focused viewpoint the reader is directed to the Stakeholder's perspectives [p.60] section which examines the architecture from the perspective of key stakeholders of the architecture.

The reason for choosing an alphabetical ordering is that there is a large amount of cross-referencing between the concepts. As a result, it is very difficult, if not misleading, to choose a non-alphabetic ordering that reflects some sense of priority between the concepts. Furthermore, the optimal ordering depends very much on the point of view of the reader. Hence, we devote the Stakeholders perspectives [p.60] section to a number of prioriterized readings of the architecture.

2.2.5 Conformance

Unlike language specifications, or protocol specifications, conformance to an architecture is necessarily a somewhat imprecise art. However, the presence of a concept in this enumeration is a strong hint that, in any realization of the architecture, there should be a corresponding feature in the implementation. Furthermore, if a relationship is identified here, then there should be corresponding relationships in any realized architecture. The consequence of non-conformance is likely to be reduced interoperability: The absence of such a concrete feature may not prevent interoperability, but it is likely to make such interoperability more difficult.

A primary function of the Architecture's enumeration in terms of models, concepts and relationships is to give guidance about conformance to the architecture. For example, the architecture notes that a message [p.19] has [p.59] a message sender [p.27] ; any realization of this architecture that does not permit a message to be associated with its sender is not in conformance with the architecture. For example, SMTP could be used to transmit messages. However, since SMTP (at present) allows forgery of the sender's identity, SMTP by itself is not sufficient to discharge this responsibility.

2.3 The Architectural Models

This architecture has four models, illustrated in Figure 2-2 [p.12] . Each model in Figure 2-2 [p.12] is labeled with what may be viewed as the key concept of that model.

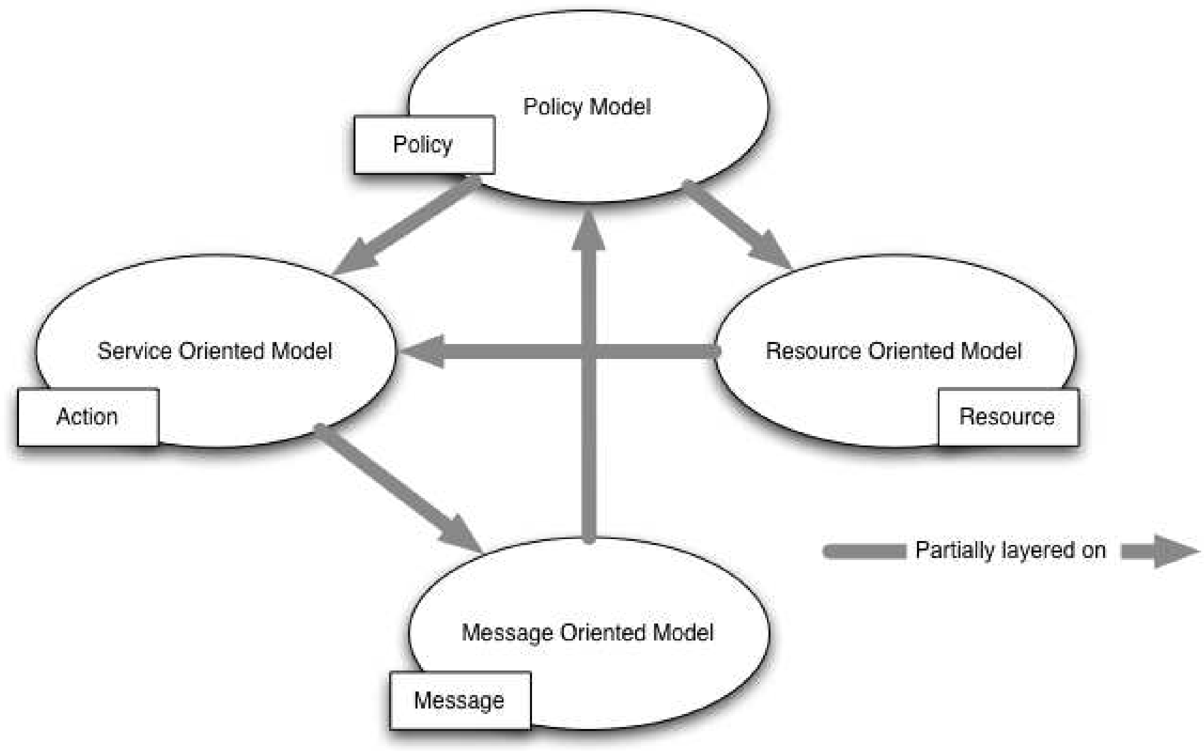

Figure 2-2. Meta Model of the Architecture

The four models are:

- The Message Oriented Model [p.17] focuses on messages, message structure, message transport and so on — without particular reference as to the reasons for the messages, nor to their significance.

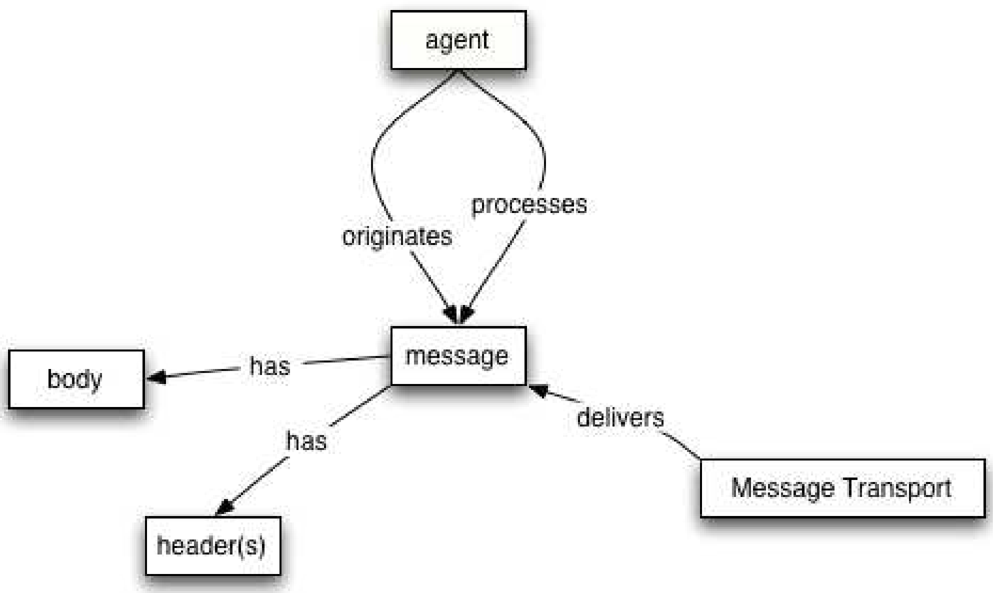

Figure 2-3. Simplified Message Oriented Model

The essence of the message model revolves around a few key concepts illustrated above: the agent [p.31] that sends and receives messages [p.19] , the structure of the message in terms of message headers [p.25] and bodies [p.21] and the mechanisms used to deliver messages. Of course, there are additional details to consider: the role of policies and how they govern the message level model. The abridged diagram shows the key concepts; the detailed diagram expands on this to include many more concepts and relationships.

- The Service Oriented Model [p.29] focuses on aspects of service [p.37] , action and so on. While clearly, in any distributed system, services cannot be adequately realized without some means of messaging, the converse is not the case: messages do not need to relate to services.

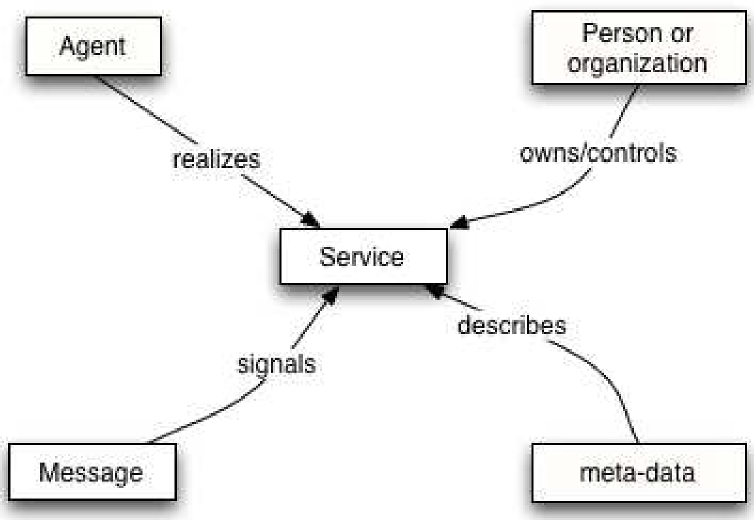

Figure 2-4. Simplified Service Oriented Model

The Service Oriented Model is the most complex of all the models in the architecture. However, it too revolves around a few key ideas. A service is realized by an agent and used by another agent. Services are mediated by means of the messages exchanged between requester agents and provider agents.

A very important aspect of services is their relationship to the real world: services are mostly deployed to offer functionality in the real world. We model this by elaborating on the concept of a service's owner — which, whether it is a person or an organization, has a real world responsibility for the service.

Finally, the Service Oriented Model makes use of meta-data, which, as described in **3.1 Service Oriented Architecture** [p.60] , is a key property of Service Oriented Architectures. This meta-data is used to document many aspects of services: from the details of the interface and transport binding to the semantics of the service and what policy restrictions there may be on the service. Providing rich descriptions is key to successful deployment and use of services across the Internet.

- The Resource Oriented Model [p.44] focuses on resources [p.48] that exist and have owners.

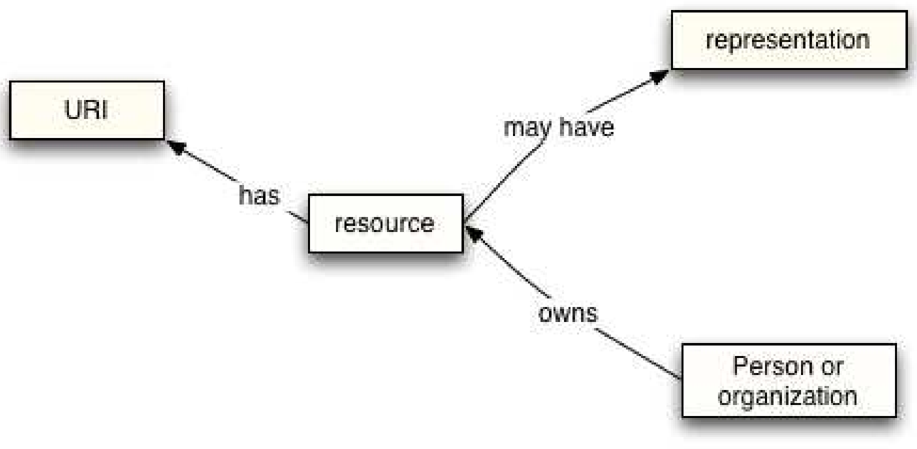

Figure 2-5. Simplified Resource Oriented Model

The resource model is adopted from the Web Architecture concept of resource. We expand on this to incorporate the relationships between resources and owners.

- The Policy Model [p.50] focuses on constraints on the behavior of agents and services. We generalize this to resources [p.48] since policies can apply equally to documents (such as descriptions of services) as well as active computational resources.

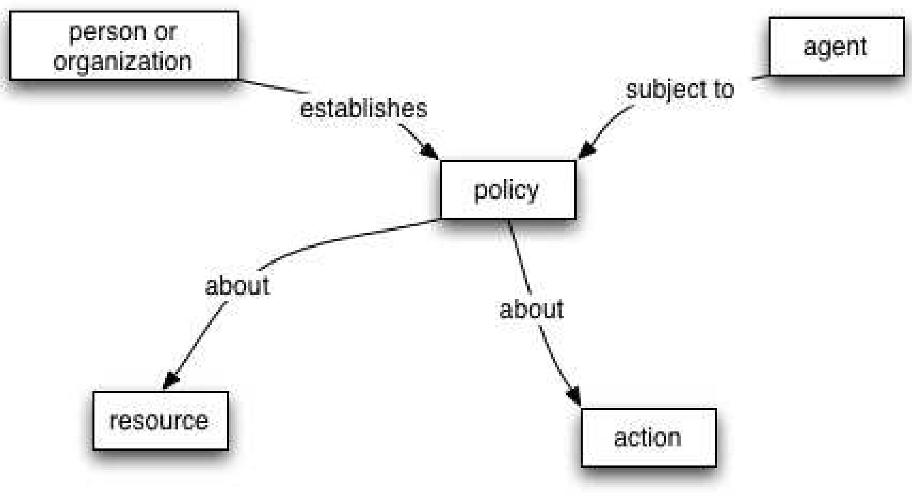

Figure 2-6. Simplified Policy Model

Policies are about resources [p.48] . They are applied to agents [p.31] that may attempt to access those resources, and are put in place, or established, by people [p.55] who have responsibility for the resource.

Policies may be enacted to represent security concerns, quality of service concerns, management concerns and application concerns.

2.3.1 Message Oriented Model

The Message Oriented Model focuses on those aspects of the architecture that relate to messages [p.19] and the processing of them. Specifically, in this model, we are not concerned with any semantic significance of the content of a message or its relationship to other messages. However, the MOM does focus on the structure of messages, on the relationship between message senders and receivers and how messages are transmitted.

The MOM is illustrated in the Figure 2-7 [p.17] :

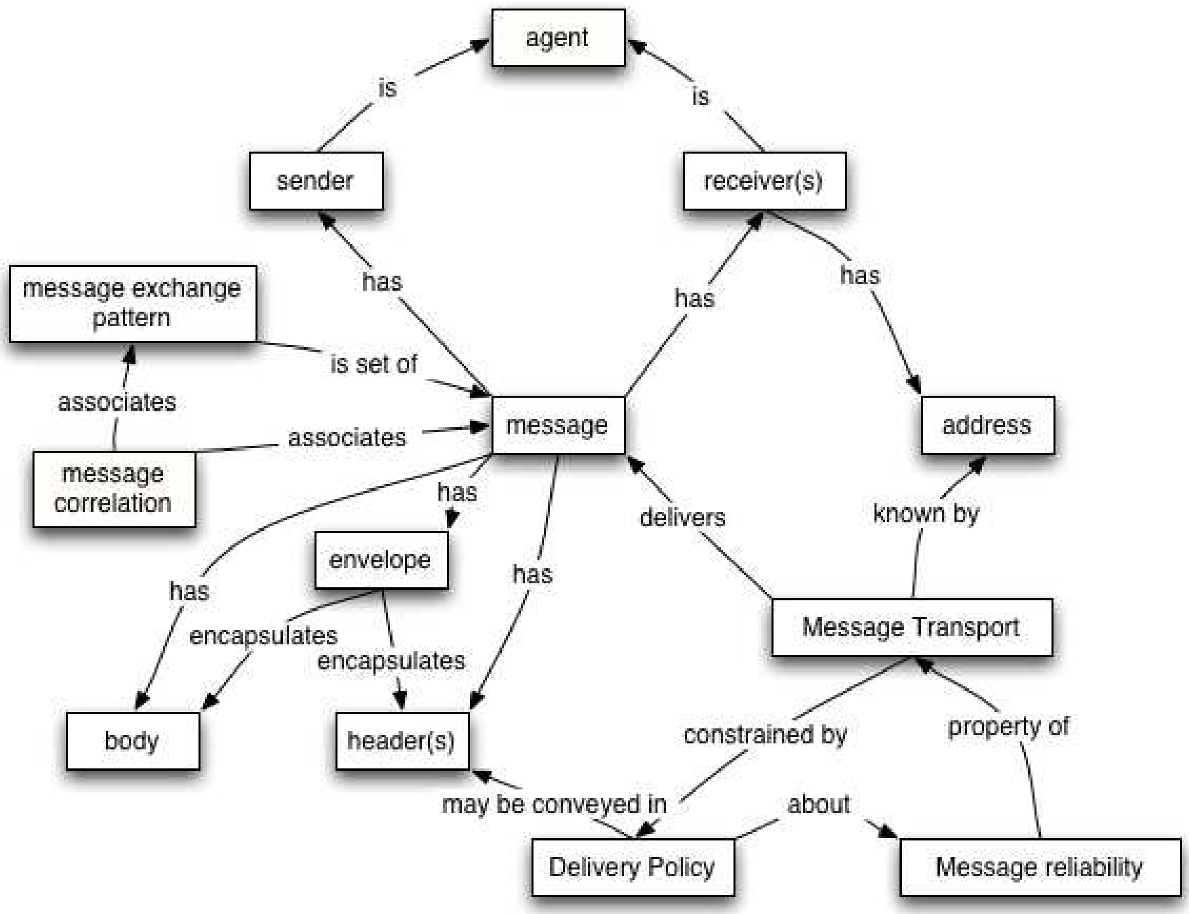

Figure 2-7. Message Oriented Model

2.3.1.1 Address

2.3.1.1.1 Definition

An address is that information required by a message transport mechanism in order to deliver a message appropriately.

2.3.1.1.2 Relationships to other elements

An address is [p.57]

> information used to describe how and where to deliver messages [p.19] .

An address may be [p.57]

> a URI.

An address is [p.57]

> typically transport mechanism [p.28] specific.

An address may be contained

> in the message envelope [p.22] .

2.3.1.1.3 Explanation

In order for message transport [p.28] mechanisms to function, it is normally necessary to provide information that allows messages to be delivered. This is called the address of the message receiver.

Typically, the form of the address information will depend of the particular message transport. In the case of an HTTP message transport, the address information will take the form of a URL.

The precise method that a message sender [p.27] uses to convey address information will also depend on the transport mechanism used. On occasion, the address information may be provided as additional arguments to the invoking procedure. Or the address information may be located within the message itself; typically in the message envelope.

2.3.1.2 Delivery Policy

2.3.1.2.1 Definition

A delivery policy is a policy that constrains the methods by which messages are delivered by the message transport.

2.3.1.2.2 Relationships to other elements

Delivery policy is [p.57]

> a policy [p.55]

Delivery policy constrains

> message transport [p.28]

Delivery policy may be expressed

> in a policy description language

Delivery policy may express

> the quality of service associated with delivering a message [p.19] by a message transport [p.28] mechanism

2.3.1.2.3 Explanation

Delivery policies are those policies [p.55] that relate to the delivery of messages.

Typically, a delivery policy applies to the combination of a particular message and a particular message transport [p.28] mechanism. The policies that apply, however, may originate from descriptions in the message itself, or be intrinsic to the transport mechanism, or both.

Examples of delivery policies include quality of service assurances — such as reliable versus best effort message delivery — and security assurances — such as encrypted versus unencrypted message transport. Another kind of delivery policy could take the form of assertions about recording an audit of how the message was delivered.

2.3.1.3 Message

2.3.1.3.1 Definition

A message is the basic unit of data sent from one Web services agent to another in the context of Web services.

2.3.1.3.2 Relationships to other elements

a message is [p.57]

> a unit of data sent from one agent [p.31] to another

a message may be [p.57] part of

> a message sequence [p.28]

a message may be described [p.58] using

a service description language

a message has [p.59]

a message sender [p.27]

a message has [p.59]

one or more message recipients [p.26]

a message may have [p.59]

an identifier [p.47]

a message has [p.59]

a message body

a message has [p.59]

zero or more message headers [p.25]

a message has [p.59]

a message envelope [p.22]

a message is delivered by

a message transport [p.28] system

a message may have [p.59]

a delivery policy [p.18] associated with it

2.3.1.3.3 Explanation

A message represents the data structure passed from its sender to its recipients. The structure of a message is defined in a service description.

The main parts of a message are its envelope, a set of zero or more headers, and the message body. The envelope serves to encapsulate the component parts of the message and it serves as a well-known location for message transport services to locate necessary addressing information. The header holds ancillary information about the message and facilitates modular processing. The body of the message contains the message content or URIs to the actual data resource.

A message can be as simple as an HTTP GET request, in which the HTTP headers are the headers and the parameters encoded in the URL are the content. Note that extended Web services functionality in this architecture is not supported in HTTP headers.

A message can also simply be a plain XML document. However, such messages do not support extended Web services functionality defined in this architecture.

A message can be a SOAP XML, in which the SOAP headers are the headers. Extended Web services functionality are supported in SOAP headers.

2.3.1.4 Message Body

2.3.1.4.1 Definition

A message body is the structure that represents the primary application-specific content that the message sender intends to deliver to the message recipient.

2.3.1.4.2 Relationships to other elements

a message body is contained by [p.59]

> the message envelope [p.22] .

a message body is [p.57]

> the application-specific content intended for the message recipient.

2.3.1.4.3 Explanation

The message body provides a mechanism for transmitting information to the recipient of the message. The form of the message body, and other constraints on the body, may be expressed as part of the service description.

In many cases, the precise interpretation of the message body will depend on the message headers [p.25] that are in the message.

2.3.1.5 Message Correlation

2.3.1.5.1 Definition

Message correlation is the association of a message with a context. Message correlation ensures that a requester agent [p.36] can match the reply with the request, especially when multiple replies may be possible.

2.3.1.5.2 Relationships to other elements

Message Correlation is [p.57]

> a means of associating a message [p.19] within a specific conversational context.

Message correlation may be realized [p.60] by including message identifiers to enable messages [p.19] to be identified.

2.3.1.5.3 Explanation

Message correlation allows a message to be associated with a particular purpose or context. In a conversation, it is important to be able to determine that an actual message that has been received is the expected message. Often this is implicit when conversations are relayed over stream-oriented message transports; but not all transports allow correlation to be established so implicitly.

For situations where correlation must be handled explicitly, one technique is to associate a message identifier with messages. The message identifier is an identifier that allows a received message to be correlated with the originating request. The sender may also add an identifier for a service, not necessarily the originating sender, who will be the recipient of the message (see asynchronous messaging).

Correlation may also be realized by the underlying protocol. For example, HTTP/1.1 allows one to correlate a request with its response.

2.3.1.6 Message Envelope

2.3.1.6.1 Definition

A message envelope is the structure that encapsulates the component parts of a message: the message body and the message headers.

2.3.1.6.2 Relationships to other elements

a message envelope may contain [p.59]

address information about the intended recipients [p.26] of its associated message [p.19]

a message envelope contains [p.59]

the message body [p.21] .

a message envelope contains [p.59]

the message headers [p.25] .

2.3.1.6.3 Explanation

Issue (message_with_address):

How is a message associated with its destination address?

There is an unresolved issue here. A message somehow must be associated with its destination address. This combination of the message with its destination address seems to be a significant architectural concept, yet SOAP does not require that the address be included in the message header.

Resolution:

None recorded.

The message envelope may contain information needed to actually deliver messages. If so, it must at least contain sufficient address information so that the message transport [p.28] can deliver the message. Typically this information is part of the service *binding* information found in a WSDL document.

Other metadata that may be present in an envelope includes security information to allow the message to be authenticated and quality of service information.

A correctly design message transport mechanism should be able to deliver a message based purely on the information in the envelope. For example, an encrypted message that fully protects the identities of the sender, recipient as well as the message content, may still be delivered using only the address information (and the encrypted data stream itself).

2.3.1.7 Message Exchange Pattern (MEP)

2.3.1.7.1 Definition

A Message Exchanage Pattern (MEP) is a template, devoid of application semantics, that describes a generic pattern for the exchange of messages between agents. It describes relationships (e.g., temporal, causal, sequential, etc.) of multiple messages exchanged in conformance with the pattern, as well as the normal and abnormal termination of any message exchange conforming to the pattern.

2.3.1.7.2 Relationships to other elements

a message exchange pattern describes [p.58]

> a generic pattern for the exchange of messages [p.19] between agents [p.31] .

a message exchange pattern should have [p.57]

> a unique identifier [p.47]

a message exchange pattern may realize [p.60]

> message correlation [p.21]

a message exchange pattern may describe [p.58]

> a service [p.37] invocation

2.3.1.7.3 Explanation

Distributed applications in a Web services architecture communicate via message exchanges. These message exchanges are logically factored into patterns that may be composed at different levels to form larger patterns. A Message Exchange Pattern (MEP) is a template, devoid of application semantics, that describes a generic pattern for the exchange of (one-way) messages between agents. The patterns can be described by state machines that define the flow of the messages, including the handling of faults that may arise, and the correlation of messages.

Issue (mep_vs_chor):

What is the difference between an MEP and a Choreography?

The precise difference between an MEP and a choreography is unresolved. Some view MEPs as being atomic patterns, and a choreography as including composition of patterns. Also, a choreography generally describes patterns that include application semantics (choreography = MEPs + application semantics), whereas an MEP is devoid of application semantics. Finally, there is usually a difference in scale between an MEP and a choregraphy: A choreography often makes use of MEPs as building blocks.

Resolution:

None recorded.

Messages that are instances of an MEP are correlated, either explicitly or implicitly. The exchanges may be synchronous or asynchronous.

In order to promote interoperability, it is useful to define common MEPs that are broadly adopted and unambiguously identified. When a MEP is described for the purpose of interoperability, it should be associated with a URI that will identify that MEP.

Some protocols may natively support certain MEPs, e.g., HTTP natively supports request-response. In other cases there is may be additional glue needed to map MEPs onto a protocol.

Web service description languages at the level of WSDL view MEPs from the perspective of a particular service actor. A simple request-reponse MEP, for example, appears as an incoming message which invokes an operation and an associated outgoing message with a reply.

An MEP is not necessarily limited to capturing only the inputs and outputs of a single service. Consider the pattern:

1. agent A uses an instance of an MEP (possibly request-response) to communicate initially with B.
2. agent B then uses a separate, but related instance of an MEP to communicate with C.
3. agent A uses another instance of an MEP to communicate with C but gets a reply only after C has processed (2).

This example makes it clear that the overall pattern cannot be described in terms of the inputs and outputs of any single interaction. The pattern involves constraints and relationships among the messages in the various MEP instances. It also illuminates the fact that exchange (1) is in in-out MEP from the perspective of actor B, and mirrored by an out-in MEP from the perspective of actor A. Finally, an actual application instantiates this communication pattern and completes the picture by adding computation at A, B and C to carry out application-specific operations.

It is instructive to consider the kinds of fault reporting that occur in such a layering. Consider a fault at the transport protocol level. This transport level may itself be able to manage certain faults (e.g., re-tries), but it may also simply report the fault to the binding level. Similarly the binding level may manage the fault (e.g., by re-initiating the underlying protocol) or may report a SOAP fault. The choreography and application layers may be intertwined or separated depending on how they are architected. There is also no rigid distinction between the choreography and binding layers; binding-level MEPs are essentially simple choreographies. Conceptually, the choreographic level can enforce constraints on message order, maintain state consistency, communicate choreographic faults to the application, etc. in ways that transcend particular bindings and transports.

2.3.1.8 Message Header

2.3.1.8.1 Definition

A message header is the part of the message that contains information about a specific aspect of the message.

2.3.1.8.2 Relationships to other elements

a message header is contained in

a message envelope [p.22]

a message header may be [p.57]

a specific well known types

Editorial note	
The "is-a" relationship here is used in a different way than elsewhere in the document.	

a message header may identify

a service role [p.41] , which denotes the kind of processing expected for the header.

a message header may be processed

independently of the message body [p.21]

2.3.1.8.3 Explanation

Message headers represent information about messages that is independently standardized (such as WS-Security) — and may have separate semantics -- from the message body. For example, there may be standard forms of message header that describe authentication of messages. The form of such headers is defined for all messages; although, of course, a given authentication header will be specific to the particular message.

The primary function of headers is to facilitate the modular processing of the message, although they can also be used to support routing and related aspects of message processing. The header part of a message can include information pertinent to extended Web services functionality, such as security, transaction context, orchestration information, message routing information, or management information.

Message headers may be processed independently of the message body, each message header may have an identifying service role that indicates the kind of processing that should be performed on messages with that header. Each message may have several headers, each potentially identifying a different service role.

Although many headers will relate to infrastructure facilities, such as security, routing, load balancing and so on; it is also possible that headers will be application specific. For example, a purchase order processing Web service may be structured into layers; corresponding to different functions within the organization. These stakeholders may process headers of different messages in standardized ways: the customer information may be captured in one standardized header, the stock items by a different standardized header and so on.

2.3.1.9 Message Receiver

2.3.1.9.1 Definition

A message receiver is an agent [p.31] that receives a message [p.19] .

2.3.1.9.2 Relationships to other elements

a message receiver is [p.57]

> a agent [p.31]

a message receiver is [p.57]

> the recipient of a message [p.19]

2.3.1.9.3 Explanation

The message receiver is an agent [p.31] that is intended to receive a message from the message sender [p.27] .

Messages may be passed through intermediaries [p.40] that process aspects of the message, typically by examining the message headers [p.25] . The message recipient may or may not be aware of processing by such intermediaries.

Often a specific message receiver, the ultimate recipient, is identified as the final recipient of a message. The ultimate recipient will be responsible for completing the processing of the message.

2.3.1.10 Message Reliability

2.3.1.10.1 Definition

Message reliability is the degree of certainty that a message will be delivered and that sender and receiver will both have the same understanding of the delivery status.

2.3.1.10.2 Relationships to other elements

message reliability is [p.57]

a property of message delivery.

message reliability may be realized [p.60] by

a combination of message acknowledgement and correlation [p.21] .

message reliability may be realized [p.60] by

a transport mechanism [p.28]

2.3.1.10.3 Explanation

The goal of reliable messaging is to both reduce the error frequency for messaging and to provide sufficient information about the status of a message delivery. Such information enables a participating agent to make a compensating decision when errors or less than desired results occur. High level correlation such as "two-phase commit" is needed if more than two agents are involved. Note that in a distributed system, it is theoretically not possible to guarantee correct notification of delivery; however, in practice, simple techniques can greatly increase the overall confidence in the message delivery.

It is important to note that a guarantee of the delivery of messages alone may not improve the overall reliability of a Web service due to the need for end-to-end reliability. (See "End-to-End Arguments in System Design".) It may, however, reduce the overall cost of a Web service.

Message reliability may be realized with a combination of message receipt acknowledgement and correlation. In the event that a message has not been properly received and acted upon, the sender may attempt a resend, or some other compensating action at the application level.

2.3.1.11 Message Sender

2.3.1.11.1 Definition

A message sender is the agent that transmits a message [p.19] .

2.3.1.11.2 Relationships to other elements

a message sender is [p.57]

an agent [p.31]

a message sender is [p.57]

the originator of a message [p.19]

2.3.1.11.3 Explanation

A message sender is an agent [p.31] that transmits a message [p.19] to another agent. Although every message has a sender, the identity of the sender may not be available to others in the case of anonymous interactions.

Messages may also be passed through intermediaries that process aspects of the message; typically by examining the message headers [p.25] . The sending agent may or may not be aware of such intermediaries [p.40] .

2.3.1.12 Message Sequence

2.3.1.12.1 Definition

A message sequence is a sequence of related messages.

2.3.1.12.2 Relationships to other elements

a message sequence is [p.57]

a sequence of related messages [p.19]

a message sequence may realize [p.60]

a documented message exchange pattern [p.19]

2.3.1.12.3 Explanation

A requester agent and a provider agent exchange a number of messages during an interaction. The ordered set of messages exchanged is a message sequence.

This sequence may be realizing a well-defined MEP [p.23] , usually identified by a URI.

2.3.1.13 Message Transport

2.3.1.13.1 Definition

A Message Transport is a mechanism that may be used by agents to deliver messages.

2.3.1.13.2 Relationships to other elements

a message transport is [p.57]

a mechanism that delivers messages [p.19]

a message transport has [p.59]

zero or more capabilities [p.33]

a message transport is constrained by

various delivery policies [p.55]

a message transport must know

sufficient address [p.18] information in order to deliver a message.

2.3.1.13.3 Explanation

The message transport is the actual mechanism used to deliver messages. Examples of message transport include HTTP over TCP, SMTP, message oriented middleware, and so on.

The responsibility of the message transport is to deliver a message from a sender to one or more recipient, i.e. transport a SOAP Infoset from one agent to another, possibly with some implied semantics (e.g. HTTP methods semantics).

Message transports may provide different features (e.g. message integrity, quality of service guaranties, etc.).

For a message transport to function, the sending agent must provide the address [p.18] of the recipient.

2.3.2 The Service Oriented Model

The Service Oriented Model (SOM) focuses on those aspects of the architecture that relate to service [p.37] and action [p.30] .

The primary purpose of the SOM is to explicate the relationships between an agent [p.31] and the services [p.37] it provides and requests. The SOM builds on the MOM, but its focus is on action [p.30] rather than message.

The concepts and relationships in the SOM are illustrated in Figure 2-8 [p.29] :

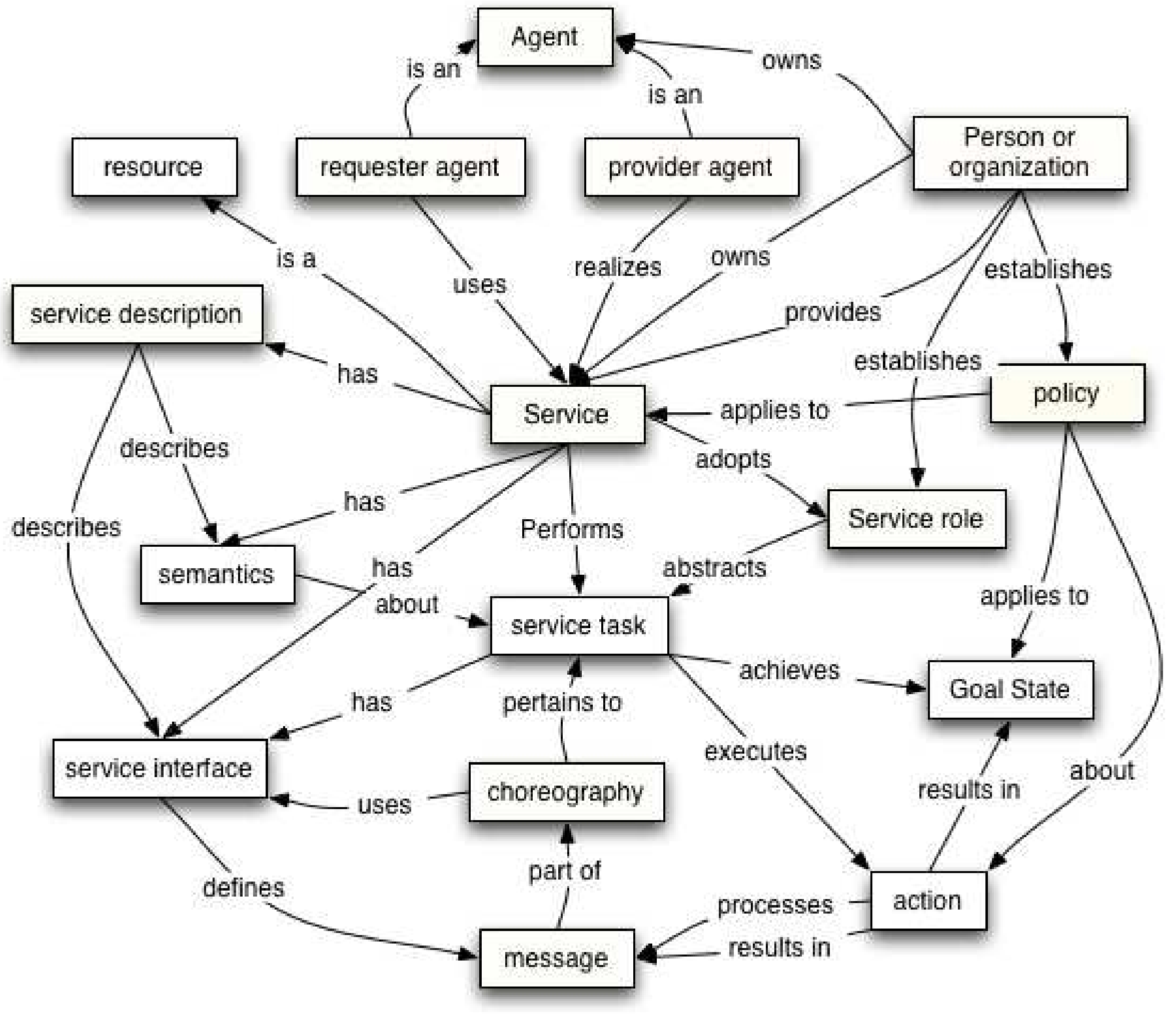

Figure 2-8. Service Oriented Model

2.3.2.1 Action

2.3.2.1.1 Definition

An action, for the purposes of this architecture, is any action that may be performed by an agent [p.31] , possibly as a result of receiving a message [p.19] , or which results in sending a message [p.19] or another observable state change.

2.3.2.1.2 Relationships to other elements

An action may result in [p.60]

a desired goal state [p.34]

An action may be [p.57]

the sending of a message [p.19]

An action may be [p.57]

the processing of a received message [p.19]

2.3.2.1.3 Explanation

At the core of the concept of service [p.37] is the notion of one party performing action(s) at the behest of another party. From the perspective of requester and provider agents, an action is typically performed by executing some fragment of a program.

In the WSA, the actions performed by requester and provider agents are largely out of scope, except in so far as they are the result of messages being sent or received. In effect, the programs that are executed by agents are not in scope of the architecture, however the resulting messages are in scope.

2.3.2.2 Agent

2.3.2.2.1 Definition

An agent [p.31] is a program acting on behalf of person or organization [p.55] . (This definition is a specialization of the definition in [Web Arch] [p.96] . It corresponds to the notion of *software agent* in [Web Arch] [p.96] .)

2.3.2.2.2 Relationships to other elements

An agent is [p.57]

a computational resource [p.48]

An agent has [p.59]

an owner that is a person or organization [p.55]

An agent may realize [p.60]

zero or more services [p.37]

An agent may request [p.36]

zero or more services [p.37]

2.3.2.2.3 Explanation

Agents are programs that engage in actions on behalf of someone or something else. For our purposes, agents realize and request Web services. In effect, software agents are the running programs that *drive* Web services — both to implement them and to access them.

Software agents are also *proxies* for the entities [p.55] that own them. This is important as many services involve the use of resources which also have owners with a definite interest in their disposition. For example, services [p.37] may involve the transfer of money and the incurring of legal obligations as a result.

We specifically avoid any attempt to govern the implementation of agents; we are only concerned with ensuring interopability between systems.

2.3.2.3 Choreography

2.3.2.3.1 Definition

A choreography defines the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state.

Editorial note	
This is a different level of abstraction from the definition used by the W3C Web Services Choreography Working Group.	

2.3.2.3.2 Relationships to other elements

A choreography uses

one or more service interfaces [p.40] .

A choreography defines

the pattern of possible interactions between a set of agents [p.31] .

A choreography may be expressed in

a choreography description language

A choreography pertains to

a given task [p.43]

A choreography defines

the relationship between exchanged messages [p.19] and tasks [p.43] of a service [p.37] .

2.3.2.3.3 Explanation

A choreography is a model of the sequence of operations, states, and conditions that control the interactions involved in the participating services. The interaction prescribed by a choreography results in the completion of some useful function. Examples include the placement of an order, information about its delivery and eventual payment, or putting the system into a well-defined error state.

A choreography can be distinguished from an orchestration. An orchestration defines the sequence and conditions in which *one* Web service invokes other Web services in order to realize some useful function.

A choreography may be described using a choreography description language. A choreography description language permits the description of how Web services can be composed, how service roles and associations in Web services can be established, and how the state, if any, of composed services is to be managed.

2.3.2.4 Capability

2.3.2.4.1 Definition

A capability is a named piece of functionality (or feature) that is declared as supported or requested by an agent [p.31] .

2.3.2.4.2 Relationships to other elements

a capability has a [p.59]

identifier [p.47] which is a URI

a capability has a [p.59]

a description of its semantics

a capability can be

advertised by an agent [p.31] that supports it

a capability can be

required by agent [p.31] that wishes to use it

a capability may be referenced by

a service description [p.39]

2.3.2.4.3 Explanation

Agents participating in an exchange may implement a wide variety of features. For example, there may be different ways to achieve the reliable delivery of a message, or there may be several mechanisms available to support security. A Web service may advertise that it supports a particular capability, and an agent requiring that capability might select the service on that basis. Or a Web service may indicate that it requires a particular capability of any requester agent that uses it, and a requester agent may select it or avoid it on that basis. There may also be a negotiation -- either manual or automatic -- about which capabilities to select.

The concept of capability encompasses SOAP features, but is broader.

2.3.2.5 Goal State

2.3.2.5.1 Definition

A goal state is a state of some service or resource that is desireable from some person or organization's [p.55] point of view.

2.3.2.5.2 Relationships to other elements

a goal state is [p.57]

> a state of the real world, which includes the state of relevant resources

a goal state is [p.57]

> desired by some person or organization [p.55] which has an interest in defining it.

a goal state may be characterized

> informally, or formally with a formal expression.

2.3.2.5.3 Explanation

Goal states are associated with tasks. Tasks are the unit of action associated with services [p.37] that have a measurable meaning. Typically measured from the perspective of the owner of a service, a goal state is characterized by a predicate that is true of that state — for example, a book selling service may have as its goal state that a book has been purchased by a legitimate customer.

It is difficult to be formal about vague properties such as desireable, however, it is also clear that services are deployed and used with an intention. An e-commerce service is oriented towards buying and selling, a stock ticker service is oriented towards giving timely information. A goal state is simply a way of being able to declare success when a task has completed sucessfully.

2.3.2.6 Provider Agent

2.3.2.6.1 Definition

A provider agent is an agent that is capable of and empowered to perform the actions associated with a service on behalf of its owner — the provider entity [p.35] .

2.3.2.6.2 Relationships to other elements

a provider agent is [p.57]

a Web service [p.37] software agent [p.31]

a provider agent realizes [p.60]

one or more services [p.37]

a provider agent performs, or causes to perform

the actions [p.30] associated with a task [p.43]

a provider agent acts on behalf of

a provider entity [p.35]

2.3.2.6.3 Explanation

The provider agent is the software agent that realizes a Web service by performing tasks on behalf of its owner — the provider entity [p.35] .

A given service may be offered by more than one agent, especially in the case of composite services, and a given provider agent may realize more than one Web service.

2.3.2.7 Provider Entity

2.3.2.7.1 Definition

The provider entity is the person or organization [p.55] that is providing a Web service.

2.3.2.7.2 Relationships to other elements

a provider entity

is a [p.57] person or organization [p.55]

a provider entity

offers a Web service [p.37]

a provider entity owns [p.59]

a provider agent [p.34]

2.3.2.7.3 Explanation

The provider entity is the person or organization [p.55] that is offering a Web service. The provider agent acts on behalf of the provider entity that owns it.

2.3.2.8 Requester Agent

2.3.2.8.1 Definition

A requester agent is a software agent [p.31] that wishes to interact with a provider agent [p.34] in order to request that a task be performed on behalf of its owner — the requester entity [p.36] .

2.3.2.8.2 Relationships to other elements

a requester agent is [p.57]

> an agent [p.31]

a requester agent uses

> a service [p.37]

a requester agent acts on behalf of

> a requester entity [p.36]

2.3.2.8.3 Explanation

The requester agent is the software agent that requires a certain function to be performed on behalf of its owner — the requester entity. From an architectural perspective, this is the agent [p.31] that is looking for and invoking or initiating an interaction with a provider agent.

2.3.2.9 Requester Entity

2.3.2.9.1 Definition

The requester entity is the person or organization that wishes to use a provider entity [p.35] 's Web service.

2.3.2.9.2 Relationships to other elements

a requester entity

> is a [p.57] person or organization [p.55]

a requester entity owns [p.59]

> a requester agent [p.36]

2.3.2.9.3 Explanation

The requester entity is the person or organization [p.55] that wishes to make use of a Web service. The requester entity is the counterpart to the provider entity [p.35] .

2.3.2.10 Service

2.3.2.10.1 Definition

A service is an abstract resource that represents a capability of performing tasks that represents a coherent functionality from the point of view of provider entities [p.35] and requester entities [p.36] . To be used, a service must be realized by a concrete provider agent [p.34] .

2.3.2.10.2 Relationships to other elements

a service is a [p.57]

resource [p.48]

a service performs

one or more tasks [p.43]

a service has [p.59]

a service description [p.39]

a service has a [p.59]

service interface [p.40]

a service has [p.59]

service semantics [p.42]

a service has [p.59]

an identifier [p.47]

a service has [p.59]

a service semantics [p.42]

a service has [p.59]

one or more service roles [p.41] in relation to the service's owner

a service may have [p.59]

one or more policies [p.55] applied to it.

a service is owned by [p.59]

a person or organization [p.55] .

a service is provided by

a person or organization [p.55] .

a service is realized by [p.60]

a provider agent [p.34] .

a service is used by

a requester agent [p.36] .

2.3.2.10.3 Explanation

A service is an abstract resource [p.48] that represents a person or organization [p.55] in some collection of related tasks [p.43] as having specific service roles [p.41] . The service may be realized by one or more provider agents [p.34] that act on behalf of the person or organization — the provider entity.

The critical distinction of a Web service, compared to other Web resources, is that Web services do not necessarily have a representation [p.48] ; however, they *are* associated with actions [p.30] .

Issue (ws_get):

What should be the representation returned by an HTTP "GET" on a Web service URI?

What should be the representation of a Web service? Should a service description be available at the service URI?

Resolution:

None recorded.

For a Web service to be compliant with this architecture there must be sufficient service descriptions [p.39] associated with the service to enable its use by other parties. Ideally, a service description will give sufficient information so that an automatic agent may not only use the service but also decide if the service is appropriate; that in turn implies a description of the semantics of the service.

We distinguish a number of things in their relation to a service: a service has an owner; a service must be realized by a (software) provider agent; a requester agent may interact with a provider agent; and a provider agent has an owner (the provider entity [p.35]). Web services are inherently *about* computer-to-computer interactions between requester and provider agents; yet they are also ultimately deployed in human service because the requester and provider agents act on behalf of their owners.

Web services are focused on actions [p.30] . It is convenient, for the purposes of characterizing their semantics, to capture this in terms of tasks [p.43] . The semantics of any computational system is bound with the behavior of the system: and the intended semantics is bound with some desired behavior. Tasks combine the concept of action with intention: i.e., Web services are conventionally invoked with a given purpose in mind. The purpose can be expressed as an intended goal state: such as a book being delivered or a temperature reading being taken.

There is *no* requirement for there to be a one-to-one correspondence between messages [p.19] and services. A given message may be processed by more than one service, especially in the situation where there are service intermediaries, and a given service may, of course, process more than one kind of message. We formalize this by asserting that a service adopts one or more service roles [p.41] . The service role identifies the intended role as determined by the owner [p.55] of the service. A given role is characterized by the aspects of messages it is concerned with.

2.3.2.11 Service Description

2.3.2.11.1 Definition

A service description is a set of documents that describe the interface to and semantics of a service [p.37] .

2.3.2.11.2 Relationships to other elements

a service description is [p.57]

> a machine-processable description of a service [p.37]

a service description is [p.57]

> a machine-processable description of the service's interface [p.40]

a service description contains

> a machine-processable description of the messages [p.19] that are exchanged by the service [p.37]

a service description may include

> a description of the service's semantics [p.42]

a service description is expressed in

> a service description language

2.3.2.11.3 Explanation

A service description contains the details of the interface and, potentially, the expected behavior of the service. This includes its data types, operations, transport protocol information, and address [p.18] . It could also include categorization and other metadata to facilitate discovery and utilization. The complete description may be realized as a set of XML description documents.

There are many potential uses of service descriptions: they may be used to facilitate the construction and deployment of services, they may be used by people to locate appropriate services, and they may be used by requester agents to automatically discover appropriate provider agents in those case where requester agents are able to make suitable choices.

2.3.2.12 Service Interface

2.3.2.12.1 Definition

A service interface is the abstract boundary that a service exposes. It defines the types of messages and the message exchange patterns that are involved in interacting with the service, together with any conditions implied by those messages.

2.3.2.12.2 Relationships to other elements

a service interface defines

> the messages [p.19] relevant to the service

2.3.2.12.3 Explanation

A service interface defines the different types of messages that a service sends and receives, along with the message exchange patterns that may be used.

2.3.2.13 Service Intermediary

2.3.2.13.1 Definition

A service intermediary is a Web service whose main role is to transform messages in a value-added way. (From a messaging point of view, an intermediary processes messages en route from one agent to another.) Specifically, we say that a service intermediary is a service whose outgoing messages are equivalent to its incoming messages in some application-defined sense.

2.3.2.13.2 Relationships to other elements

A service intermediary is [p.57]

> a service [p.37] .

A service intermediary adopts

> a specific service role [p.41] .

A service intermediary preserves

> the semantics of messages it receives and sends.

2.3.2.13.3 Explanation

A service intermediary is a specific kind of service which typically acts as a kind of filter on messages it handles. Normally, intermediaries do not consume messages but rather forward them to other services. Of course, intermediaries do often *modify* messages but, it is of the essence that *from some application specific perspective* they do not modify the meaning of the message.

Of course, if a message is altered in any way, then from some perspectives it is no longer the same message. However, just as a paper document is altered whenever anyone writes a comment on the document, and yet it is still the same document, so an intermediary modifies the messages that it receives, forwarding the same message with some changes.

Coupled with the concept of service intermediary is the service role [p.41] is adopts. Typically, this involves one or more of the messages' headers rather than the bodies of messages. The specification of the header is coupled with the permissable semantics of the intermediary should make it clear to what extent the messages forwarded by an itnermediary are the same message and what modifications are permitted.

There are a number of situations where additional processing of messages is required. For example, messages that are exchanged between agents within an enterprise may not need encryption; however, if a message has to leave the enterprise then good security may suggest that it be encrypted. Rather than burden every software agent with the means of encrypting and decrypting messages, this functionality can be realized by means of an intermediary. The main responsiblity of the software agents then becomes ensuring that the messages are routed appropriately and have the right headers targetted at the appropriate intermediaries.

2.3.2.14 Service Role

2.3.2.14.1 Definition

A service role is an abstract set of tasks which is identified to be relevant by a person or organization [p.55] offering a service. Service roles are also associated with particular aspects of messages exchanged with a service [p.37] .

2.3.2.14.2 Relationships to other elements

a service role is [p.57]

> a set of service tasks [p.43]

a service role may be defined

> in terms of particular properties of messages [p.19] .

a service role may be established by

> a service owner [p.55] .

2.3.2.14.3 Explanation

A service role is an intermediate abstraction between service [p.37] and task [p.43] . A given message that is received by a service may involve processing associated with several service roles. Similarly, messages emitted may also involve more than one service role.

We can formalize the distinction by noting that a service role is typically associated with a particular property of messages. For *ultimate* processing, the service role may be to determine some final disposition of messages received. However, other service roles may be associated with more generic properties of messages — such as their encryption, or whether they reference a customer or inventory item.

Service roles identify the points of interest that a service owner has in the processing of messages. As such, they are established by the party that offers [p.55] in the service.

2.3.2.15 Service Semantics

2.3.2.15.1 Definition

The semantics of a service is the behavior expected when interacting with the service. The semantics expresses a contract (not necessarily a legal contract) between the provider entity [p.35] and the requester entity [p.36] . It expresses the intended real-world effect of invoking the service. A service semantics may be formally described in a machine readable form, identified but not formally defined, or informally defined via an "out of band" agreement between the provider entity and the requester entity.

2.3.2.15.2 Relationships to other elements

a service semantics is [p.57]

> the contract between the provider entity [p.35] and the requester entity [p.36] concerning the effects and requirements pertaining to the use of a service [p.37]

a service semantics describes [p.58]

> the intended effects of using a service [p.37]

a service semantics is about [p.57]

> the service tasks [p.43] that constitute the service.

a service semantics should be identified

> in a service description [p.39]

a service semantics may be described

> in a formal, machine-processable language

2.3.2.15.3 Explanation

Knowing the type of a data structure is not enough to understand the intent and meaning behind its use. For example, methods to deposit and withdraw from an account typically have the same type signature, but with a different effect. The effects of the operations are the semantics of the operation. It is good practice to be *explicit* about the intended effects of using a Web service; perhaps even to the point of constructing a machine readable description of the semantics of a service.

Machine processable semantic descriptions provide the potential for sophisticated usage of Web services. For example, by accessing such descriptions, a requester agent may autonomously choose which provider agent to use.

Apart from the expected behavior of a service, other semantic aspects of a service include any policy restrictions on the service, the relationship between the provider entity and the requester entity, and what manageability features are associated with the service.

2.3.2.16 Service Task

2.3.2.16.1 Definition

A service task is an action or combination of actions that is associated with a desired goal state. Performing the task involves executing the actions, and is intended to achieve a particular goal state.

2.3.2.16.2 Relationships to other elements

a service task is [p.57]

> an action [p.30] or combination of actions.

a service task is associated with [p.59]

> one or more intended goal states [p.34] .

a service task is performed [p.30] by

> executing the actions [p.30] associated with the task.

a service task has a [p.59]

> service interface [p.40]

2.3.2.16.3 Explanation

A service task is an abstraction that encapsulates some intended effect of invoking a service.

Tasks are associated with goal states — characterized by predicates that are satisfied on successful completion.

The performance of a task is made observable by the exchange of messages between the requester agent [p.36] and the provider agent [p.34] . The specific pattern of messages is what defines the choreography associated with the task.

In addition to exchanged messages, there may be other private actions associated with a task. For example, in a database update task, the task may be signaled by an initiating message and a completion message, which are public, and the actual database update, which is typically private.

In the case of a service oriented architecture [p.60] only the public aspects of a task are important, and these are expressed entirely in terms of the messages exchanged.

Tasks represent a useful unit in modeling the semantics of a service [p.37] and indeed of a service role [p.41] — a given service may consist of a number of tasks.

2.3.3 The Resource Oriented Model

The Resource Oriented Model focuses on those aspects of the architecture that relate to resources [p.48] . Resources are a fundamental concept that underpins much of the Web and much of Web services; for example, a Web service is a particular kind of resource that is important to this architecture.

The ROM focuses on the key features of resources that are relevant to the concept of resource, independent of the role the resource has in the context of Web services. Thus we focus on issues such as the ownership of resources, policies associated with resources and so on. Then, by virtue of the fact that Web services are resources, these properties are inherited by Web services.

We illustrate the basic concepts and relationships in the ROM in Figure 2-9 [p.44] :

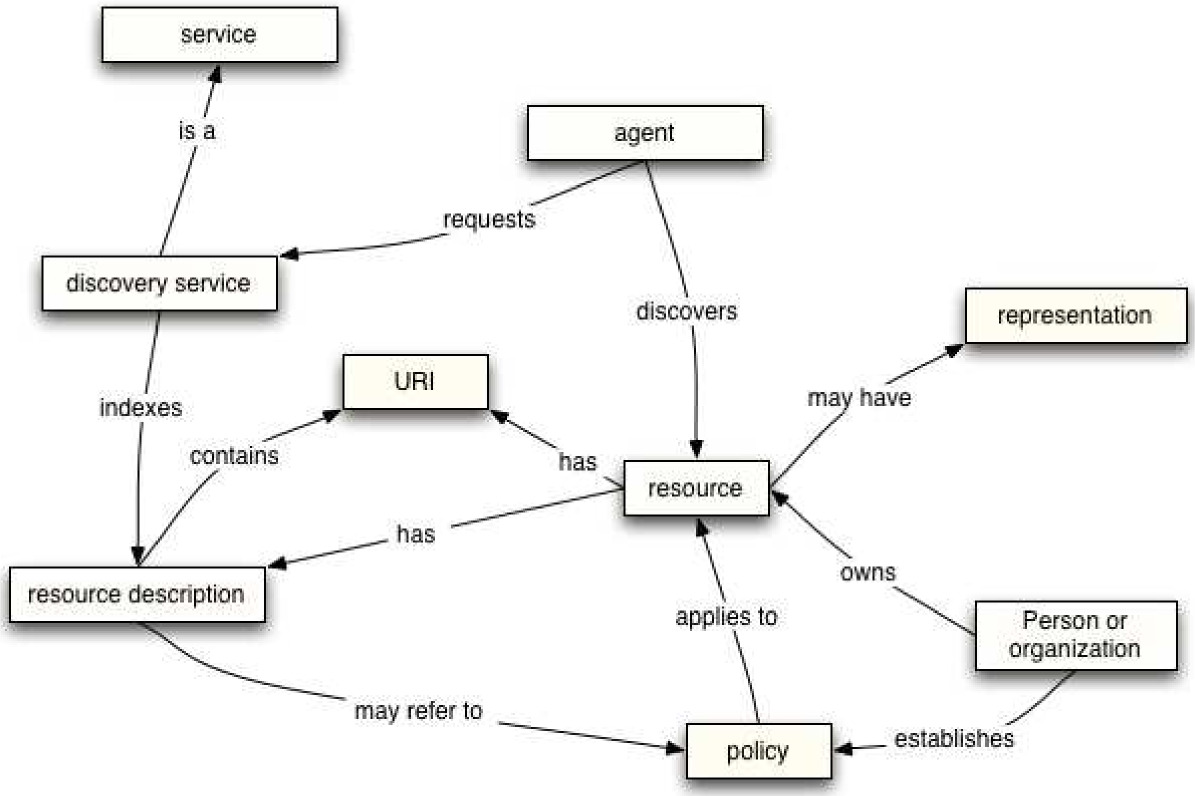

Figure 2-9. Resource Oriented Model

2.3.3.1 Discovery

2.3.3.1.1 Definition

Discovery is the act of locating a machine-processable description of a Web service-related resource that may have been previously unknown and that meets certain functional criteria. It involves matching a set of functional and other criteria with a set of resource descriptions. The goal is to find an appropriate Web service-related resource.[WS Glossary] [p.96]

2.3.3.1.2 Relationships to other elements

Discovery is

> the act of locating a resource description [p.49]

Discovery involves

> matching a set of functional and other criteria with a set of resource descriptions.

Discovery may be performed [p.30]

by an agent [p.31] , or by an end-user

Discovery may be realized [p.60]

using a discovery service [p.46]

2.3.3.1.3 Explanation

In the context of Web services, the resources being discovered are usually service descriptions. If a requester entity does not already know what service it wishes to engage, the requester entity must discover one. There are various means by which discovery can be performed. Various things — human end users or agents — may initiate discovery. Requester entities may find service descriptions during development for static binding, or during execution for dynamic binding. For statically bound requester agents, using discovery is optional, as the service description might be obtained in other ways, such as being sent directly from the provider entity to the requester entity, developed collaboratively, or provided by a third party, such as a standards body.

2.3.3.2 Discovery Service

2.3.3.2.1 Definition

A discovery service is a service that enables agents to retrieve Web service-related resource descriptions.

2.3.3.2.2 Relationships to other elements

A discovery service is [p.57]

a service [p.37]

A discovery service is used to

publish descriptions [p.49]

A discovery service is used to

search for resource descriptions [p.49]

A discovery service may be used

by an agent [p.31]

2.3.3.2.3 Explanation

A discovery service is used to publish and search for descriptions meeting certain functional or semantic criteria. It is primarily intended for use by requester entities, to facilitate the process of finding an appropiate provider agent for a particular task. However, depending on the implementation and policy of the discovery service (**3.4.2 Discovery: Registry, Index or Peer-to-Peer?** [p.71]), it may also be used by provider entities to actively publish their service descriptions.

Although the resource model is general purpose, the most important resource for our purposes is the Web service. Furthermore, the primary role of a discovery service is to facilitate the discovery of Web services.

For dynamic discovery, the requester agent may interact directly with the discovery service to find an appropriate provider agent to engage. For static discovery, a human may interact with the discovery service through an appropriate software agent, such as a browser.

The use of an automated discovery service is optional, since other means can be used to enable a requester entity and provider entity to agree [p.67] on the service description that will govern the interaction. For example, the requester entity might obtain the service description directly from the provider entity, the two parties might develop the service description collaboratively, or, in some circumstances, the service description may be created by the *requester* entity and dictated to the provider entity. (For example, a large company may require its suppliers to provide Web services that conform to a particular service description.) Likewise, a requester entity can obtain a service description from other sources besides a discovery service, such as a local file system, FTP site, URL, or WSIL document.

2.3.3.3 Identifier

2.3.3.3.1 Definition

An identifier is an unambiguous name for a resource.

2.3.3.3.2 Relationships to other elements

an identifier should be realized [p.60]

    a URI

an identifier identifies

    a resource that is relevant to the architecture

2.3.3.3.3 Explanation

Identifiers are used to identify resources. In the architecture we use Uniform Resource Identifiers [RFC 2396] [p.96] to identify resources.

Issue (urivsqname):

Should URIs be used to identify Web services components, rather than QNames?

Some specifications use QNames to identify things. However, QNames may be ambiguous, because the same QName may be used to identify things of different types. (In effect, specifications having this practice have different symbol spaces to distinguish the different uses of a QName.) Should URIs be preferred instead of QNames for Web services? A significant majority of this Working Group believes the answer is yes.

Resolution:

None recorded.

2.3.3.4 Representation

2.3.3.4.1 Definition

A representation is a piece of data that describes a resource state.

2.3.3.4.2 Relationships to other elements

a resource may have a [p.59]

representation

2.3.3.4.3 Explanation

Representations are data objects that reflect the state of a resource. A resource has a unique identifier (a URI). Note that a representation of a resource need not be the same as the resource itself; for example the resource asociated with the booking state of a restaurant will have different representations depending on when the representation is retrieved. A representation is usually retrieved by performing an HTTP "GET" on a URI.

2.3.3.5 Resource

2.3.3.5.1 Definition

A resource is defined by [RFC 2396] [p.96] to be anything that can have an identifier [p.47] . Although resources in general can be anything, this architecture is only concerned with those resources that are relevant to Web services and therefore have some additional characteristics. In particular, they incorporate the concepts of ownership and control: a resource that appears in this architecture is a *thing* that has a name, may have reasonable representations and which can be said to be owned. The ownership of a resource is critically connected with the right to set policy on the resource.

2.3.3.5.2 Relationships to other elements

a resource has [p.59]

an identifier [p.47]

a resource may have [p.59]

zero or more representations

a resource may have [p.59]

zero or more resource descriptions [p.49] a resource is owned by [p.59]

a person or organization [p.55]

a resource may be governed by

zero or more policies [p.55]

2.3.3.5.3 Explanation

Resources form the heart of the Web architecture itself. The Web is a universe of resources that have URIs as identifiers, as defined in [RFC 2396] [p.96] .

From a real-world perspective, a most interesting aspect of a resource is its ownership: a resource is something that can be owned, and therefore have policies applied to it. Policies applying to resources are relevant to the management of Web services, security of access to Web services and many other aspects of the role that a resource has in the world.

2.3.3.6 Resource description

2.3.3.6.1 Definition

A resource description is any machine readable data that may permit resources to be discovered. Resource descriptions may be of many different forms, tailored for specific purposes, but all resource descriptions must contain the resource's identifier.

2.3.3.6.2 Relationships to other elements

A resource description contains [p.59]

the resource [p.48] 's identifier [p.47]

A resource description may reference

the policies [p.55] applicable to the resource

A resource description may reference

the semantics [p.42] applicable to the resource

2.3.3.6.3 Explanation

A resource description is a machine-processable description of a resource. Resource descriptions are used by and within discovery services [p.46] to permit agents to discover the resource.

The precise contents of a resource description will vary, depending on the resource, on the purpose of the description and on the accessibility of the resource. However, for our purposes it is important to note that the description must contain the resource's identifier. I.e., a description of the form: "the new resource that is owned by XYZ co." is not regarded as a valid resource description because it does not mention the resource's identifier.

A primary purpose of resource descriptions is to facilitate the discovery of the resource. To aid that purpose, the description is likely to contain information about the location of the resource, how to access it and potentially any policies that govern the policy. Where the resource is a Web service, the description may also contain machine-processable information about how to invoke the Web service and the expected effect of using the Web service.

Note that a resource description is fundamentally distinct from the resource representation [p.48] . The latter is a snapshot reflecting the state of resource, the description is meta-level information about the resource.

2.3.4 The Policy Model

The Policy Model focuses on those aspects of the architecture that relate to policies [p.55] and, by extension, security and quality of service.

Security is fundamentally about constraints; about constraints on the behavior on action and on accessing resources. Similarly, quality of service is also about constraints on service. In the PM, these constraints are modeled around the core concept of policy [p.55] ; and the relationships with other elements of the architecture. Thus the PM is a framework in which security can be realized.

However, there are many other kinds of constraints, and policies that are relevant to Web services, including various application-level constraints.

The concepts and relationships in the PM are illustrated in Figure 2-10 [p.50] :

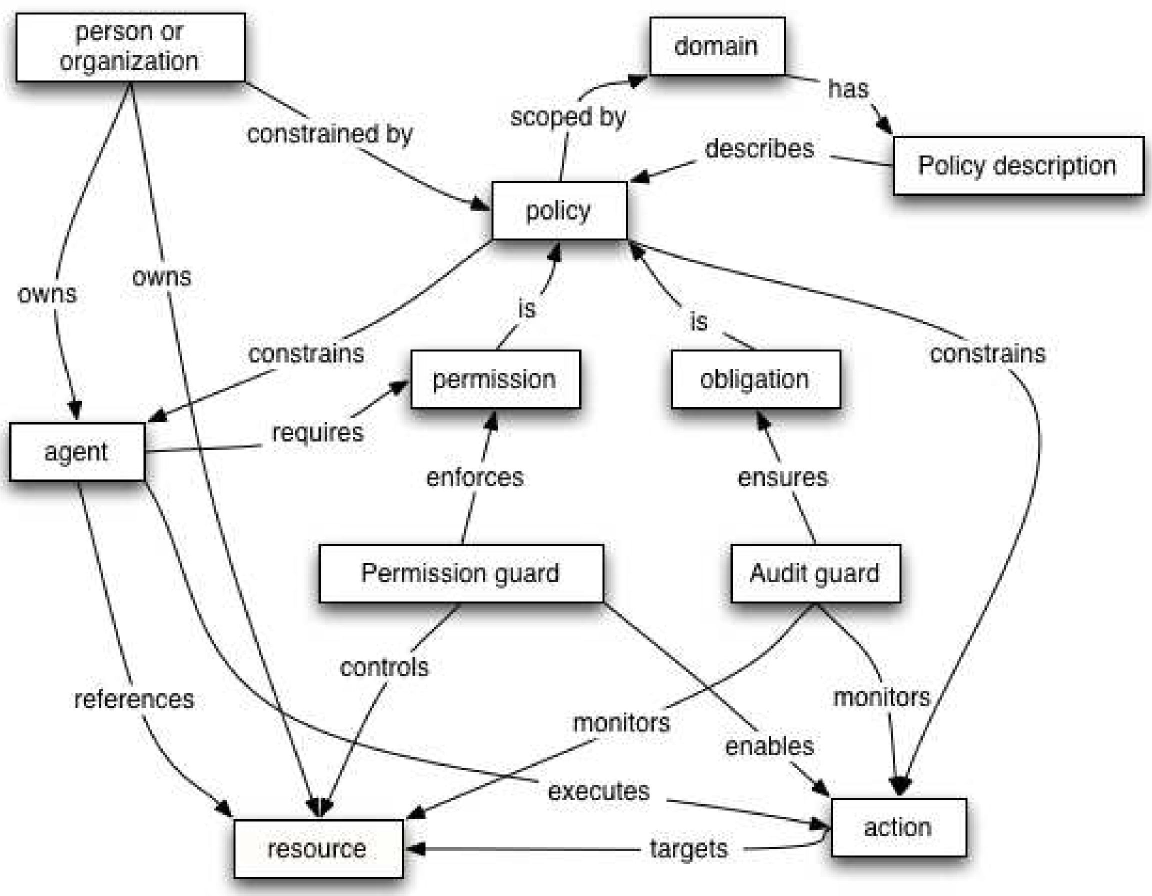

Figure 2-10. Policy Model

2.3.4.1 Audit Guard

2.3.4.1.1 Definition

An audit guard is a mechanism used on behalf of an owner that monitors actions and agents to verify the satisfaction of obligations.

2.3.4.1.2 Relationships to other elements

a audit guard is a [p.57]

a policy guard [p.57]

an audit guard may monitor

one or more resources. [p.48]

an audit guard may monitor

actions [p.30] relative to one or more services [p.37] .

an audit guard may determine

if an agent [p.31] 's obligations have been discharged.

2.3.4.1.3 Explanation

An audit guard is an enforcement mechanism. It is used to monitor the discharge of obligations. The role of the audit guard is to monitor that agents, resources and services are consistent with any associated obligations established by the service's owner or manager.

Typically, an audit guard monitors the state of a resource or a service, ensuring that the obligation is satisfied. It determines whether the associated obligations are satisfied.

By their nature, it is not possible to proactively enforce obligations; hence, an obligation violation may result in some kind of retribution after the fact of the violation.

2.3.4.2 Domain

2.3.4.2.1 Definition

A domain is an identified set of agents and/or resources that is subject to the constraints of one of more policies. [p.55]

2.3.4.2.2 Relationships to other elements

A domain is [p.57]

a collection of agents [p.31] and/or resources. [p.48]

A domain defines

the scope of application of one or more policies [p.55]

2.3.4.2.3 Explanation

A domain defines the scope of applicability of policies [p.55] . A domain may be defined explicitly or implicitly. Members of an explicitly defined domain are enumerated by a central authority; members of an implicitly defined domain are not. For example, membership in an implicitly defined domain may depend on the state of the agent or something it possesses, and thus may be dynamic.

2.3.4.3 Obligation

2.3.4.3.1 Definition

An obligation is a kind of policy that prescribes actions and/or states of an agent and/or resource.

2.3.4.3.2 Relationships to other elements

an obligation is a [p.57]

  kind of policy [p.55]

an obligation may require

  an agent [p.31] to perform one or more actions [p.30]

an obligation may require

  an agent or service to be in one or more allowable states

an obligation may be discharged

  by the performance of an action [p.30] or other event.

2.3.4.3.3 Explanation

An obligation is one of two fundamental types of policies [p.55] . When an agent has an obligation to perform some action, then it is required to do so. When the action is performed, then the agent can be said to have satisfied its obligations.

Not all obligations relate to actions. For example, an agent providing a service may have an obligation to maintain a certain state of readiness. (Quality of service policies are often expressed in terms of obligations.) Such an obligation is typically not discharged by any of the obligee's actions; although an event (such as a certain time period expiring) may discharge the obligation.

Obligations, by their nature, cannot be proactively enforced. However, obligations are associated with enforcement mechanisms: audit guards [p.51] . These monitor controlled resources and agents and may result in some kind of *retribution*; retributions are not modeled by this architecture.

An obligation may continue to exist after its requirements have been met (for example, an obligation to maintain a particular credit card balance), or it may be discharged by some action or event.

2.3.4.4 Permission

2.3.4.4.1 Definition

A permission is a kind of policy that prescribes the allowed actions and states of an agent and/or resource.

2.3.4.4.2 Relationships to other elements

a permission is a [p.57]

type of policy [p.55]

a permission may enable

one or more actions [p.30]

a permission may enable

one or more allowable states

2.3.4.4.3 Explanation

A permission is one of two fundamental types of policies [p.55] . When an agent has permission to perform some action, to access some resource, or to achieve a certain state, then it is expected that any attempt to perform the action etc., will be successful. Conversely, if an agent [p.31] does *not* have the required permission, then the action should fail even if it would otherwise have succeeded.

Permissions are enforced by guards, in particular permission guards [p.54] , whose function is to ensure that permission policies are honored.

2.3.4.5 Permission Guard

2.3.4.5.1 Definition

A permission guard is a mechanism deployed on behalf of an owner to enforce permission policies.

2.3.4.5.2 Relationships to other elements

a permission guard is a [p.57]

a policy guard [p.57]

a permission guard is a [p.57]

a mechanism that enforces permission policies [p.53]

a permission guard may control

one or more resources. [p.48]

a permission guard enables

actions [p.30] relative to one or more services. [p.37]

2.3.4.5.3 Explanation

A permission guard is an enforcement mechanism that is used to enforce permission policies [p.55] . The role of the permission guard is to ensure that any uses of a service or resource are consistent with the policies established by the service's owner or manager.

Typically, a permission guard sits between a resource or service and the requester of that resource or service. In many situations, it is not necessary for a service to be aware of the permission guard. For example, one possible role of a message intermediary [p.40] is to act as a permission guard for the final intended recipient of messages.

A permission guard acts by either enabling a requested action or access, or by denying it. Thus, it is normally possible for permission [p.53] policies to be proactively enforced.

2.3.4.6 Person or Organization

2.3.4.6.1 Definition

A person or organization may be the owner of agents that provide or request Web services.

2.3.4.6.2 Relationships to other elements

a person or organization may own [p.59]

an agent [p.31]

a person or organization may belong to

a domain [p.52]

a person or organization may establish

policies [p.55] governing resources that they own

2.3.4.6.3 Explanation

The WSA concept of person or organization [p.55] is intended to refer to the real-world people that are represented by agents that perform actions on their behalf. All actions considered in this architecture are ultimately rooted in the actions of humans.

2.3.4.7 Policy

2.3.4.7.1 Definition

A policy is a constraint on the behavior of agents or people or organizations.

2.3.4.7.2 Relationships to other elements

a policy is a [p.57]

constraint on the allowable actions or states of an agent [p.31] or person or organization [p.55]

a policy may have [p.59]

an identifier [p.47]

a policy may be described [p.58]

in a policy description [p.56]

a policy may define

a capability [p.33]

2.3.4.7.3 Explanation

A policy is a constraint on the behavior of agents as they perform actions or access resources.

There are many kinds of policies, some relate to accessing resources in particular ways, others relate more generally to the allowable actions an agent may perform: both as provider agents and as requester agents.

Logically, we identify two types of policy: permissions [p.53] and obligations [p.52] .

Although most policies relate to actions of various kinds, it is not exclusively so. For example, there may be a policy that an agent must be in a certain state (or conversely may not be in a particular state) in relation to the services it is requesting or providing.

Closely associated with policies are the mechanisms for establishing policies and for enforcing them. This architecture does not model the former.

Policies have applications for defining security properties, quality of service properties, management properties and even application properties.

2.3.4.8 Policy Description

2.3.4.8.1 Definition

A policy description is a machine-processable description of a policy or set of policies.

2.3.4.8.2 Relationships to other elements

a policy description describes [p.58]

a policy [p.55]

2.3.4.8.3 Explanation

A policy description is a machine processable description of some constraint on the behavior of agents as they perform actions, access resources.

The policy description itself is not the policy, but it may define the policy and it may be used to determine if the policy *applies* in a given situation.

Policy descriptions may include specific conditions, such as "agents of XXX Co. may access files in directory FFF". They may also include more general rules, such as "if an entity has the right to access files in the directory FFF, it also has the obligation to close them after 20 seconds.".

2.3.4.9 Policy Guard

2.3.4.9.1 Definition

A policy guard is a mechanism that enforces one or more policies. It is deployed on behalf of an owner.

2.3.4.9.2 Relationships to other elements

a policy guard has [p.59]

> an owner responsible for establishing the guard

2.3.4.9.3 Explanation

A policy guard is an abstraction that denotes a mechanism that is used by owners of resources to enforce policies.

The architecture identifies two kinds of policy guards: audit guards [p.51] and permission guards [p.54] . These relate to the core kinds of policies (obligation and permission policies respectively).

2.4 Relationships

This section defines terms that appear in our architectural models but are not specific to Web services or Web services architecture. However, they are defined here to help clarify our use of these terms in this document.

2.4.1 The *is a* relationship

2.4.1.1 Definition

The *X* is a *Y* relationship denotes the relationship between concepts *X* and *Y*, such that every *X* is also a *Y*.

2.4.1.2 Relationships to other elements

Assuming that *X* is a *Y*, then:

true of

> if *P* is true of *Y* then *P* is true of *X*

contains

> if *Y* has a [p.59] *P* then *X* has a [p.59] *Q* such that *Q* is a [p.57] *P*.

transitive

> if *P* is true of *Y* then *P* is true of *X*

2.4.1.3 Explanation

Essentially, when we say that concept *X* is a *Y* we mean that every feature of *Y* is also a feature of *X*. Note, however, that since *X* is presumably a more specific concept than *Y*, the features of *X* may also be more specific variants of the features of *Y*.

For example, in the service [p.37] concept, we state that every service has an identifier. In the more specific Web service [p.37] concept, we note that a Web service has an identifier in the form of a URI identifier.

2.4.2 The *describes* relationship

2.4.2.1 Definition

The concept *Y* describes *X* if and only if *Y* is an expression of some language *L* and that the values of *Y* are instances of *X*.

2.4.2.2 Relationships to other elements

Assuming that *Y* describes *X*, then: if *Y* is a valid expression of *L*, then the values of *Y* are instances of concept *X*

2.4.2.3 Explanation

Essentially, when we say that *Y* describes concept *X* we are saying that the expression *Y* denotes instances of *X*.

For example, in the service description [p.39] concept, we state that service descriptions are expressed in a service description language. That means that we can expect legal expressions of the service description language to be instances of the service description concept.

2.4.3 The *has a* relationship

2.4.3.1 Definition

Saying that "the concept *X* has a *Y* relationship" denotes that every instance of *X* is associated with an instance of *Y*.

2.4.3.2 Relationships to other elements

Assuming that *X* has a *Y*, then: if *E* is an instance of *X* then *Y* is valid for *E*.

2.4.3.3 Explanation

When we say that "concept *X* has a *Y*" we mean that whenever we see an *X* we should also see a *Y*

For example, in the Web service [p.37] concept, we state that Web services have URI identifiers. So, whenever the Web service concept is found, we can assume that the Web service referenced has an identifier. This, in turn, allows implementations to use identifiers to reliably refer to Web services. If a given Web service does not have an identifier associated with it, then the architecture has been violated.

2.4.4 The *owns* relationship

2.4.4.1 Definition

The relationship "*X* owns *Y*" denotes the relationship between concepts *X* and *Y*, such that every *X* has the right and authority to control, utilize and dispose of *Y*.

2.4.4.2 Relationships to other elements

Assuming that *X* owns *Y*, then:

policy

> *X* has the right to establish policies that constrain agents [p.31] and other entities in their use of *Y*

disposal

> *X* has the right to transfer some or all of his rights with respect to *Y* to another entity.

transitive

> if *P* is true of *Y* then *P* is true of *X*

2.4.4.3 Explanation

Essentially, when we say that *X* owns *Y* we mean that *X* has a significant set of rights with respect to *Y*, and that those rights are transferrable.

In general, ownership is partial, and there may be many entities that have rights with respect to some service or resource.

2.4.5 The *realized* relationship

2.4.5.1 Definition

The statement "concept *X* is realized as *Y*" denotes that the concept *X* is an abstraction of the concept *Y*. An equivalent view is that the concept *X* is implemented using *Y*.

2.4.5.2 Relationships to other elements

Assuming that *X* is realized as *Y*, then:

implemented

> if *Y* is present, or true of a system, then the concept *X* applies to the system

reified

> *Y* is a reification of the concept *X*.

2.4.5.3 Explanation

When we say that the concept or feature *X* is realized as *Y*, we mean that *Y* is an implementation or reification of the concept *X*. I.e., if *Y* is a valid concept of a system then we have also ensured that the concept *X* is valid of the same system.

For example, in the correlation [p.21] feature, we state that message correlation requires that we associate identifiers with messages. This can be realized in a number of ways — including the identifier in the message header, message body, in a service binding and so on. The message identifier is a key to the realization of message correlation.

3 Stakeholder’s Perspectives

This section examines the architecture from various perspectives, each perspective representing one coherent view of the architecture. For example, security represents one major stakeholder’s perspective of the architecture itself.

3.1 Service Oriented Architecture

3.1.1 Distributed Systems

A *distributed system* consists of diverse, discrete software agents that must work together to perform some tasks. Furthermore, the agents in a distributed system do not operate in the same processing environment, so they must communicate by hardware/software protocol stacks over a network. This means that communications with a distributed system are intrinsically less fast and reliable than those using direct code invocation and shared memory. This has important architectural implications because distributed systems require that developers (of infrastructure and applications) consider the unpredictable latency of remote access, and take into account issues of concurrency and the possibility of partial failure [Dist Comp] [p.96] .

Distributed *object* systems are distributed systems in which the semantics of object initialization and method invocation are exposed to remote systems by means of a proprietary or standardized mechanism to broker requests across system boundaries, marshall and unmarshall method argument data, etc. Distributed objects systems typically (albeit not necessarily) are characterized by objects maintaining a fairly complex internal state required to support their methods, a fine grained or "chatty" interaction between an object and a program using it, and a focus on a shared implementation type system and interface hierarchy between the object and the program that uses it.

A Service Oriented Architecture (SOA) is a form of distributed systems architecture that is typically characterized by the following properties:

- Logical view: The service is an abstracted, *logical* view of actual programs, databases, business processes, etc., defined in terms of what it *does*, typically carrying out a business-level operation.
- Message orientation: The service is formally defined in terms of the messages exchanged between provider agents and requester agents, and not the properties of the agents themselves. The internal structure of an agent, including features such as its implementation language, process structure and even database structure, are deliberately abstracted away in the SOA: using the SOA discipline one does not and should not need to know how an agent implementing a service is constructed. A key benefit of this concerns so-called legacy systems. By avoiding any knowledge of the internal structure of an agent, one can incorporate any software component or application that can be "wrapped" in message handling code that allows it to adhere to the formal service definition.
- Description orientation: A service is described by machine-processable meta data. The description supports the public nature of the SOA: only those details that are exposed to the public and important for the use of the service should be included in the description. The semantics of a service should be documented, either directly or indirectly, by its description.
- Granularity: Services tend to use a small number of operations with relatively large and complex messages.
- Network orientation: Services tend to be oriented toward use over a network, though this is not an absolute requirement.
- Platform neutral: Messages are sent in a platform-neutral, standardized format delivered through the interfaces. XML is the most obvious format that meets this constraint.

3.1.2 Web Services and Architectural Styles

Distributed object systems have a number of architectural challenges. [Dist Comp] [p.96] and others point out:

- Problems introduced by latency and unreliability of the underlying transport.
- The lack of shared memory between the caller and object.
- The numerous problems introduced by partial failure scenarios.
- The challenges of concurrent access to remote resources.
- The fragility of distributed systems if incompatible updates are introduced to any participant.

These challenges apply irrespective of whether the distributed object system is implemented using COM/CORBA or Web services technologies. Web services are no less appropriate than the alternatives if the fundamental criteria for successful distributed object architectures are met. If these criteria are met, Web services technologies may be appropriate if the benefits they offer in terms of platform/vendor neutrality offset the performance and implementation immaturity issues they may introduce.

Conversely, using Web services technologies to implement a distributed system doesn't magically turn a distributed object architecture into an SOA. Nor are Web services technologies *necessarily* the best choice for implementing SOAs -- if the necessary infrastructure and expertise are in place to use COM or CORBA as the implementation technology and there is no requirement for platform neutrality, using SOAP/WSDL may not add enough benefits to justify their costs in performance, etc.

In general SOA and Web services are most appropriate for applications:

- That must operate over the Internet where reliability and speed cannot be guaranteed;
- Where there is no ability to manage deployment so that all requesters and providers are upgraded at once;
- Where components of the distributed system run on different platforms and vendor products;
- Where an existing application needs to be exposed for use over a network, and can be wrapped as a Web service.

3.1.3 Relationship to the World Wide Web and REST Architectures

The World Wide Web operates as a networked information system that imposes several constraints: Agents identify objects in the system, called *resources*, with Uniform Resource Identifiers (URIs). Agents represent, describe, and communicate resource state via *representations* of the resource in a variety of widely-understood data formats (e.g. XML, HTML, CSS, JPEG, PNG). Agents exchange representations via protocols that use URIs to identify and directly or indirectly address the agents and resources. [Web Arch] [p.96]

An even more constrained architectural style for reliable Web applications known as *Representation State Transfer* (REST) has been proposed by Roy Fielding and has inspired both the W3C Technical Architecture Group's architecture document [Web Arch] [p.96] and many who see it as a model for how to build Web services [Fielding] [p.96] . The REST Web is the subset of the WWW (based on HTTP) in which agents provide *uniform interface semantics* -- essentially create, retrieve, update and delete -- rather than arbitrary or application-specific interfaces, and manipulate resources only by the exchange of *representations*. Furthermore, the REST interactions are "stateless" in the sense that the meaning of a message does not depend on the state of the conversation.

We can identify two major classes of Web services:

- *REST-compliant Web services,* in which the primary purpose of the service is to manipulate XML representations of Web resources using a uniform set of "stateless" operations; and
- *arbitrary Web services,* in which the service may expose an arbitrary set of operations.

Both classes of Web services use URIs to identify resources and use Web protocols (such as HTTP and SOAP 1.2) and XML data formats for messaging. (It should be noted that SOAP 1.2 *can* be used in a manner consistent with REST. However, SOAP 1.2 can also be used in a manner that is *not* consistent with REST.)

3.2 Web Services Technologies

Web service architecture involves many layered and interrelated technologies. There are many ways to visualize these technologies, just as there are many ways to build and use Web services. Figure 3-1 [p.63] below provides one illustration of some of these technology families.

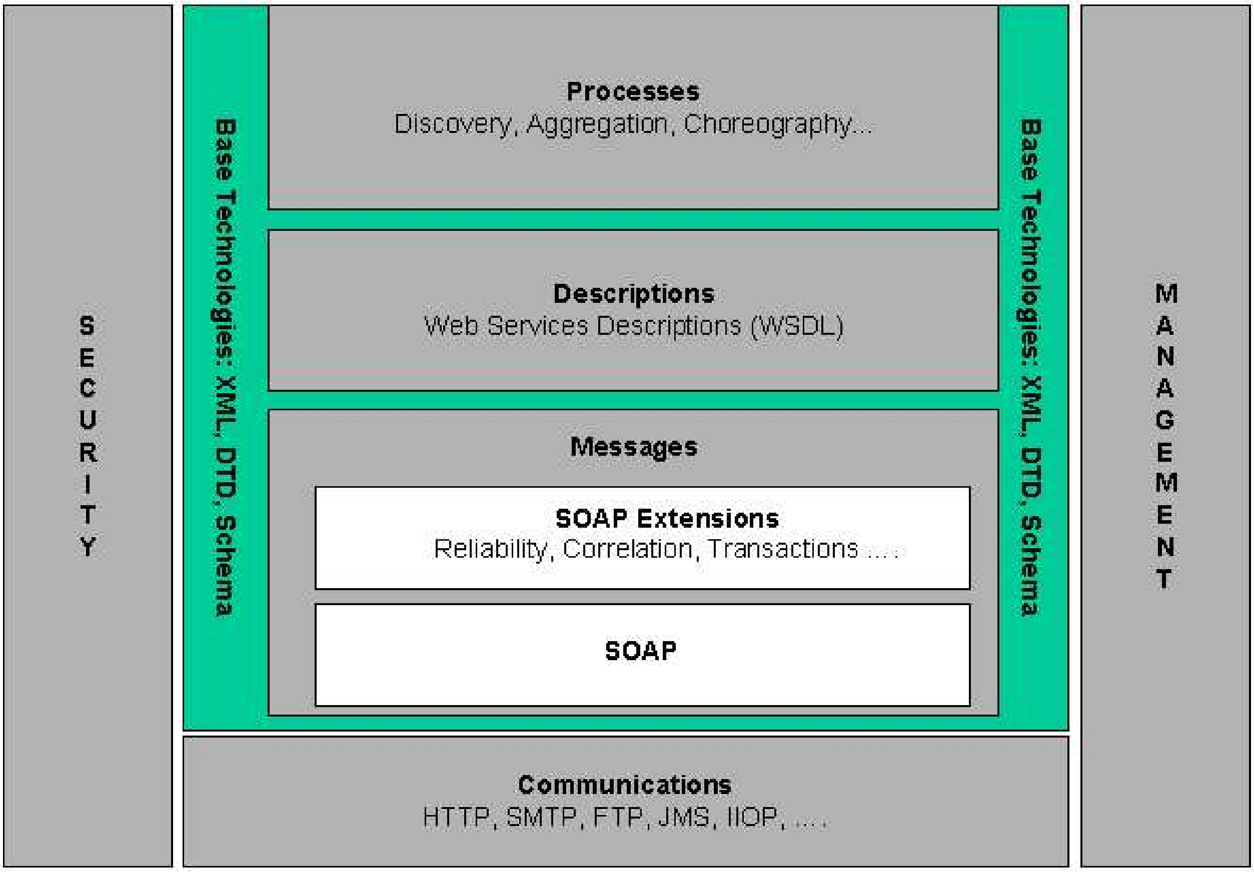

Figure 3-1. Web Services Architecture Stack

In this section we describe some of those technologies that seem critical and the role they fill in relation to this architecture. This is a necessarily bottom-up perspective, since, in this section, we are looking at Web services from the perspective of tools which can be used to design, build and deploy Web serivces.

The technologies that we consider here, in relation to the Architecture, are XML, SOAP, WSDL. However, there are many other technologies that may be useful. (For example, see the list of Web services specifications compiled by Roger Cutler and Paul Denning.) See also **B An Overview of Web Services Security Technologies** [p.93]

3.2.1 XML

XML solves a key technology requirement that appears in many places. By offering a standard, flexible and inherently extensible data format, XML significantly reduces the burden of deploying the many technologies needed to ensure the success of Web services.

The important aspects of XML, for the purposes of this Architecture, are the core syntax itself, the concepts of the XML Infoset [XML Infoset] [p.97] , XML Schema and XML Namespaces.

XML Infoset is not a data format per se, but a formal set of information items and their associated properties that comprise an abstract description of an XML document [XML 1.0] [p.97] . The XML Infoset specification provides for a consistent and rigorous set of definitions for use in other specifications that need to refer to the information in a well-formed XML document.

Serialization of the XML Infoset definitions of information may be expressed using XML 1.0 [XML 1.0] [p.97] . However, this is not an inherent requirement of the architecture. The flexibility in choice of serialization format(s) allows for broader interoperability between agents in the system. In the future, a binary encoding of the XML infoset may be a suitable replacement for the textual serialization. Such a binary encoding may be more efficient and more suitable for machine-to-machine interactions.

3.2.2 SOAP

SOAP 1.2 provides a standard, extensible, composable framework for packaging and exchanging XML messages. In the context of this architecture, SOAP 1.2 also provides a convenient mechanism for referencing capabilities [p.33] (typically by use of headers).

[SOAP 1.2 Part 1] [p.96] defines an XML-based messaging framework: a processing model and an exensibility model. SOAP messages can be carried by a variety of network protocols; such as HTTP, SMTP, FTP, RMI/IIOP, or a proprietary messaging protocol.

[SOAP 1.2 Part 2] [p.96] defines three optional components: a set of encoding rules for expressing instances of application-defined data types, a convention for representing remote procedure calls (RPC) and responses, and a set of rules for using SOAP with HTTP/1.1.

While SOAP Version 1.2 [SOAP 1.2 Part 1] [p.96] doesn't define "SOAP" as an acronym anymore, there are two expansions of the term that reflect these different ways in which the technology can be interpreted:

1. Service Oriented Architecture Protocol: In the general case, a SOAP message represents the information needed to invoke a service or reflect the results of a service invocation, and contains the information specified in the service interface definition.

2. Simple Object Access Protocol: When using the optional SOAP RPC Representation, a SOAP message represents a method invocation on a remote object, and the serialization of in the argument list of that method that must be moved from the local environment to the remote environment.

3.2.3 WSDL

WSDL 2.0[WSDL 2.0 Part 1] [p.97] is a language for describing Web services [p.39] .

WSDL describes Web services starting with the messages that are exchanged between the requester and provider agents. The messages themselves are described abstractly and then bound to a concrete network protocol and message format.

Web service definitions can be mapped to any implementation language, platform, object model, or messaging system. Simple extensions to existing Internet infrastructure can implement Web services for interaction via browsers or directly within an application. The application could be implemented using COM, JMS, CORBA, COBOL, or any number of proprietary integration solutions. As long as both the sender and receiver agree [p.67] on the service description, (e.g. WSDL file), the implementations behind the Web services can be anything.

3.3 Using Web Services

The introduction outlined and illustrated (in Figure 1-1 [p.9]) the four broad steps involved in the process of engaging a Web service (see **1.4.5 Overview of Engaging a Web Service** [p.8]). This section expands on these steps. Although these steps are *necessary*, they may not be *sufficient*: many scenarios will require additional steps, or significant refinements of these fundamental steps. Furthermore, the order in which the steps are performed may vary from situation to situation.

1. The requester and provider entities "become known to each other", in the sense that whichever party initiates the interaction must become aware of the other party. There are two cases.

    a) In a typical case, the *requester* agent will be the initiator. In this case, we would say that the requester entity must become aware of the provider entity, i.e., the requester agent must somehow obtain the address of the provider agent. There are two ways this may typically occur: (1) the requester entity may obtain the provider agent’s address directly from the provider entity; or (2) the requester entity may use a discovery service to locate a suitable service description (which contains the provider agent’s invocation address) via an associated functional description, either through manual discovery or autonomous selection. These cases are described more fully in **3.4 Web Service Discovery** [p.68] .

    b) In other cases, the *provider* agent may initiate the exchange of messages between the requester and provider agents. In this case, saying that the requester and provider entities become known to each other actually means that the *provider* entity becomes aware of the *requester* entity, i.e., the provider agent somehow obtains the address of the requester agent. How this occurs is application dependent and irrelevant to this architecture. Although this case is expected to be less common than when the requester agent is the initiator, it is important in some "push" or subscription scenarios.

2. The requester entity and provider entity agree [p.67] on the service description (a WSDL document) and semantics that will govern the interaction between the requester agent and the provider agent. (See the note below on "Agreeing on the Same Semantics and Service Description [p.67] for further explanation of what is meant here by "agree".)

    This does not necessarily mean that the requester and provider entities must communicate or negotiate with each other. It simply means that both parties must have the same (or compatible) understandings of the service description and semantics, and intend to uphold them. There are many ways this can be achieved, such as:

    - The requester and provider entities may communicate directly with each other, to explicitly agree on the service description and semantics.

    - The provider entity may publish and offer both the service description and semantics as take-it-or-leave-it "contracts" that the requester entity must accept unmodified as conditions of use.

- The service description and semantics (excepting the network address of the particular service) may be defined as a standard by an industry organization, and used by many requester and provider entities. In this case, the act of the requester and provider entities reaching agreement is accomplished by both parties independently conforming to the same standard.
- The service description and semantics (perhaps excepting the network address of the service) may be defined and published by the *requester* entity (even if they are written from provider entity's perspective), and offered to provider entities on a take-it-or-leave-it basis. This may occur, for example, if a large company requires its suppliers to provide Web services that conform to a particular service description and semantics. In this case, agreement is achieved by the provider entity adopting the service description and semantics that the requester entity has published.

Depending on the scenario, Step 2 (or portions of Step 2) may be performed prior to Step 1.

3. The service description and semantics are input to, or embodied in, both the requester agent and the provider agent as appropriate. In other words, the information in them must either be input to, or implemented in, the requester and provider agents. There are many ways this can be achieved, and this architecture does not specify or care what means are used. For example:

   - An agent could be hard coded to implement a particular, fixed service description and semantics.
   - An agent could be coded in a more general way, and the desired service description and/or semantics could be input dynamically.
   - An agent could be created first, and the service description and/or semantics could be generated or deduced from the agent code. For example, a tool could examine a set of existing class files to generate a service description.

   Regardless of the approach used, from an information perspective both the semantics and the service description must somehow be input to, or implemented in, both the requester agent and the provider agent before the two agents can interact. (This is a slight simplification; see the note below on "Agreeing" on the Same Semantics and Service Description [p.67] for further explanation.)

4. The requester agent and provider agent exchange SOAP messages on behalf of their owners.

Note:

"Agreeing" on the Same Semantics and Service Description. Although it is convenient to say that the requester and provider entities must "agree" on the semantics and the service description, it is a slight simplification (and perhaps slightly misleading) to say that the parties *must* agree on the *same* semantics and service description:

- The word "agree" often connotes an active communication between the parties and an explicit act (such as signing a contract) to cause the agreement to become binding on the two parties, yet neither of these is required in the case of step 2 above.

- It is a slight simplification to say that the requester and provider agents must implement the *same* semantics and WSD, for two reasons: (1) the requester agent implements them from the perspective of the requester entity, while the provider agent implements them from the perspective of the provider entity (for example, one party's input is the other party's output); and (2) the requester and provider agents only need to implement those aspects of the service description and semantics that are *relevant* to their respective roles.

In summary, it is convenient (and evocative) to say that the requester and provider entities must *agree* on the semantics and the service description that will govern the interaction between the requester and provider agents, but it would be more accurate to say that they simply need to have a *congruent* or *non-conflicting view* of the semantics and service description of the interaction.

3.4 Web Service Discovery

If the requester entity wishes to initiate an interaction with a provider entity and it does not already know what provider agent it wishes to engage, then the requester entity may need to "discover" a suitable candidate. Discovery is "the act of locating a machine-processable description of a Web service that may have been previously unknown and that meets certain functional criteria. " [WS Glossary] [p.96] The goal is to find an appropriate Web service.

A discovery service [p.46] is a service that facilitates the process of performing discovery. It is a logical role, and could be performed by either the requester agent, the provider agent or some other agent.

Figure 3-2 [p.68] ("Discovery Process") expands on Figure 1-1 [p.9] to describe the process of engaging a Web service when a discovery service is used.

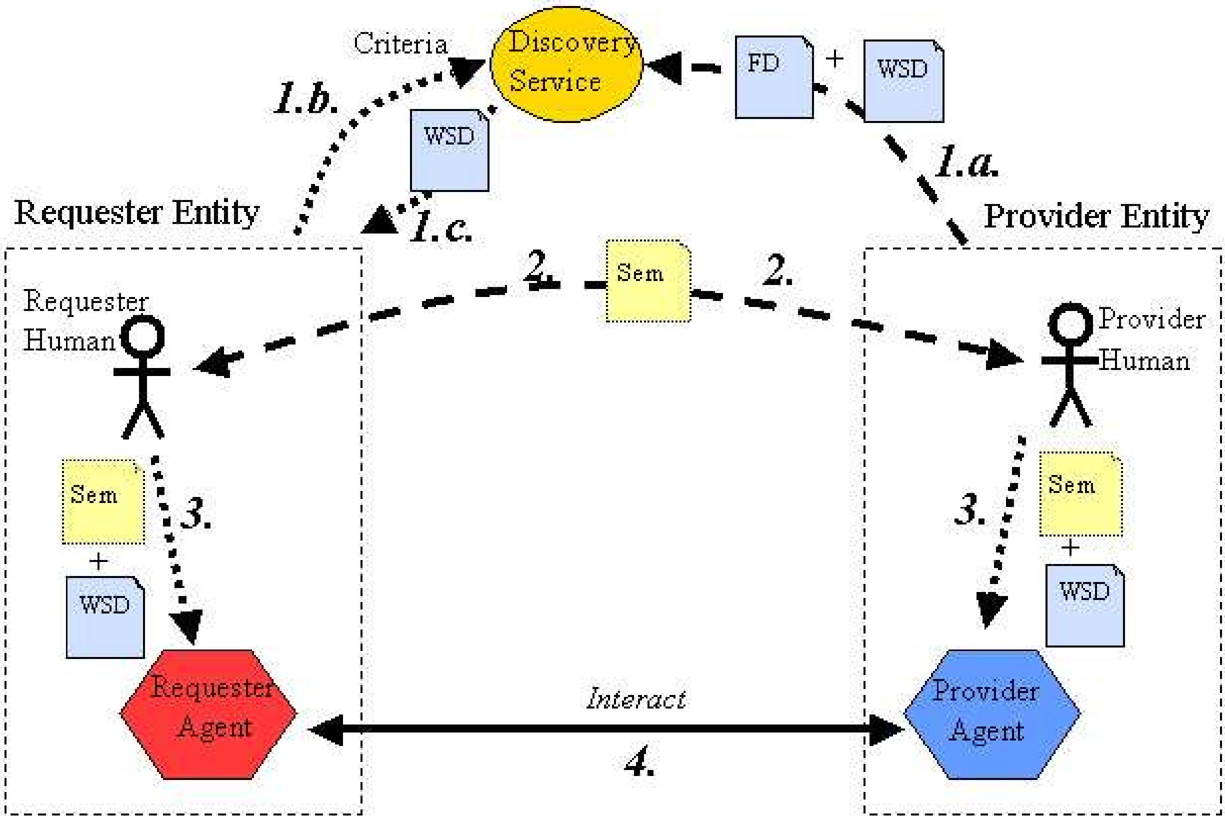

Figure 3-2. Discovery Process

Service engagement using a discovery service proceeds in roughly the following steps.

1. The requester and provider entities "become known to each other":

   a) The discovery service somehow obtains both the Web service description ("WSD" in Figure 3-2 [p.68]) and an associated functional description ("FD") of the service.

   The functional description ("FD" in Figure 3-2 [p.68]) is a machine-processable description of the functionality (or partial semantics) of the service that the provider entity is offering. It could be as simple as a few words of meta data or a URI, or it may be more complex, such as a TModel (in UDDI) or a collection of RDF, DAML-S or OWL-S statements.

   This architecture does not specify or care how the discovery service obtains the service description or functional description. For example, if the discovery service is implemented as a search engine, then it might crawl the Web, collecting service descriptions wherever it finds them, with the provider entity having no knowledge of it. Or, if the discovery service is implemented as a registry (such as UDDI), then the provider entity may need to actively publish the service description and functional description directly to the discovery service.

b) The requester entity supplies criteria to the discovery service to select a Web service description based on its associated functional description, capabilities [p.33] and potentially other characteristics. One might locate a service having certain desired functionality or semantics; however, it may be possible to specify "non-functional" criteria related to the provider agent, such as the name of the provider entity, performance or reliability criteria, or criteria related to the provider entity, such as the provider entity's vendor rating.

c) The discovery service returns one or more Web service descriptions (or references to them) that meet the specified criteria. If multiple service descriptions are returned, the requester entity selects one, perhaps using additional criteria.

2. The requester and provider entities agree [p.67] on the semantics ("Sem" in Figure 3-2 [p.68]) of the desired interaction. Although this may commonly be achieved by the provider entity defining the semantics and offering them on a take-it-or-leave-it basis to the requester entity, it could be achieved in other ways. For example, both parties may adopt certain standard service semantics that are defined by some industry standards body. Or in some circumstances the requester could define the semantics. The important point is that the parties must *agree* (in the sense described in **3.3 Using Web Services** [p.66]) on the semantics, regardless of how that is achieved.

   Step 2 also requires that the parties agree [p.67] (in the sense described in **3.3 Using Web Services** [p.66]) on the service description that is to be used. However, since the requester entity obtained the Web service description in Step 1.c, in effect the requester and provider entities have already done so.

3. The service description and semantics are input to, or embodied in, both the requester agent and the provider agent, as appropriate.

4. The requester agent and provider agent exchange SOAP messages on behalf of their owners.

3.4.1 Manual Versus Autonomous Discovery

The discovery process described above is not specific about who or what within the requester entity actually performs the discovery. Under *manual discovery*, a requester *human* uses a discovery service (typically at design time) to locate and select a service description that meets the desired functional and other criteria. Under *autonomous discovery*, the requester *agent* performs this task, either at design time or run time. Although the steps are similar in either case, the constraints and needs are significantly different, such as:

- *Interface requirements.* The requirements for something that is intended for human interaction are very different from the requirements for something that is intended for machine interaction.

- *Need for standardization.* There is far less need to standardize an interface or protocol that humans use than those that machines are intended to use.

- *Trust.* People do not necessarily trust machines to make decisions that may have significant consequences. This is explained more fully in **3.6.4.5 Trust and Discovery** [p.82] .

In the case of autonomous discovery, the need for machine-processable semantics is greatly increased.

One situation in which autonomous discovery is often needed is when the requester agent has been interacting with a particular provider agent, but for some reason needs to refresh its choice of provider agent, either because the previous provider agent is no longer available, or other reasons.

3.4.2 Discovery: Registry, Index or Peer-to-Peer?

At present, there are three leading viewpoints on how a discovery service should be conceived: as a *registry*, as an *index*, or as a *peer-to-peer* system. What are the differences? For what purpose is one better than the other?

3.4.2.1 The Registry Approach

A *registry* is an authoritative, centrally controlled store of information.

- Publishing a service description requires an active step by the provider entity: it must explicitly place the information into the registry before that information is available to others. In the case of a registry:

- The registry owner decides *who* has authority to place information into, or update, the registry. Although the owner of registry R may delegate permission to approved provider entities that wish to publish their own service descriptions, an arbitrary third party could not publish a description of someone else's service in registry R. This means, for example, that company X would not be able to register a functional description of company Y's service, even if that description would be valuable to others and may be superior in some ways to Y's own description.

- The registry owner decides *what* information is placed in the registry. Others cannot independently augment that information.

UDDI is often seen as an example of the registry approach, but it can also be used as an index.

3.4.2.2 The Index Approach

In contrast with a registry, an *index* is a compilation or guide to information that exists elsewhere. It is not authorative and does not centrally control the information that it references. In the case of an index:

- Publishing is passive: the provider entity exposes the service and functional descriptions on the Web, and those who are interested (the index owners) collect them without the provider entity's specific knowledge.

- Anyone can create their own index. When descriptions are exposed, they can be harvested using spiders and arranged into an index. Multiple organizations may have such indexes.

- The information contained in an index could be out of date. However, since the index contains pointers to the authoritative information, the information can be verified before use.

- An index could include third-party information.
- Different indexes could provide different kinds of information — some richer, some sparser.
- Free-market forces determine which index people will use to discover the information that they seek.

Google is often cited as an example of the index approach.

It is important to note that the key difference between the registry approach and the index approach is not merely the difference between a registry itself and an index *in isolation*. Indeed, UDDI could be used as a means to implement an individual index: just spider the Web, and put the results into a UDDI registry. Rather, the key difference is one of *control*: Who controls *what* and *how* service descriptions get discovered? In the registry model, it is the owner of the registry who controls this. In the index model, since anyone can create an index, market forces determine which indexes become popular. Hence, it is effectively the market that controls what and how service descriptions get discovered.

3.4.2.3 Peer-to-Peer (P2P) Discovery

Peer-to-Peer (P2P) computing provides an alternative that does not rely on centralized registries; rather it allows Web services to discover each other dynamically. Under this view, a Web service is a node in a network of peers, which may or may not be Web services. At discovery time, a requester agent queries its neighbors in search of a suitable Web service. If any one of them matches the request, then it replies. Otherwise each queries its own neighboring peers and the query propagates through the network until a particular hop count or other termination criterion is reached.

Peer-to-peer architectures do not need a centralized registry, since any node will respond to the queries it receives. P2P architectures do not have a single point of failure, such as a centralized registry. Furthermore, each node may contain its own indexing of the existing Web services. Finally, nodes contact each other directly, so the information they they receive is known to be current. (In contrast, in the registry or index approach there may be significant latency between the time a Web service is updated and the updated description is reflected in the registry or index.)

The reliability provided by the high connectivity of P2P systems comes with performance costs and lack of guarantees of predicting the path of propagation. Any node in the P2P network has to provide the resources needed to guarantee query propagations and response routing, which in turn means that most of the time the node acts as a relayer of information that may be of no interest to the node itself. This results in inefficiencies and large overhead especially as the nodes become more numerous and connectivity increases. Furthermore, there may be no guarantee that a request will spread across the entire network, therefore there is no guarantee to find the providers of a service.

3.4.2.4 Discovery Service Trade-Offs

Because of their respective advantages and disadvantages, P2P systems, indexes and centralized registries strike different trade-offs that make them appropriate in different situations. P2P systems are more appropriate in dynamic environments in which proximity naturally limits the need to propagate requests, such as ubiquitous computing. Centralized registries may be more appropriate in more static or controlled environments where information does not change frequently. Indexes may be more appropriate in situations that must scale well and accommodate competition and diversity in indexing strategies.

3.4.3 Federated Discovery Services

Although the registry viewpoint is a more centralized approach to discovery than the index approach, there will arise situations where multiple registries exist on the Web. It is expected that multiple indexes will also exist. In such an environment, web service requesters that need to use a discovery service may need to obtain information from more than one registry or index. Federation refers to the ability to consolidate the results of queries that span more than a single registry or index, and make them appear more like a single service.

A registry or index may contain information about other registries or indexes to help support federation. For example, a registry dedicated to air travel services may know about another registry dedicated to rail travel services. A third registry for general travel services may contain information about some travel services, but may look to other registries for certain categories of services. A search of the general travel registry may return a referral to the requester pointing them to the rail travel registry. Federation of results in this scenario, as contrasted to the referral, would require the general travel registry to submit a query to the rail travel registry on behalf of the requester. The general travel registry would then merge the results of the query to the rail travel registry with the results of a query to its own registry. The general, rail, and air travel registries may need to share a common taxonomy or ontology to avoid forwarding inappropriate queries to other registries. In this scenario, we assume the general travel registry examined the query from the requester and therefore did not forward the query to the air travel registry.

The general travel registry could have discovered the rail travel registry using a spider or index approach. An indexing engine could have come across a registry, and based on the information it harvested from the registry classified it as a rail travel registry. An alternative approach would be for the rail travel registry to publish information to the general travel registry and using the shared taxonomy could classify itself as a registry for rail travel services.

Note that each registry or index may provide a web service for discovery, so it may be appropriate to use a choreography or orchestration description language to describe the exchanges among these services needed for federation.

3.4.4 Functional Descriptions and Discovery

As mentioned at the beginning of **2.3.3.1 Discovery** [p.45] , Web services discovery requires the ability to search for appropriate Web services based on functional descriptions ("FD" in Figure 3-2 [p.68]) or other criteria. Because these functional descriptions need to be machine processable, written by many provider entities and read by many requester entities, an appropriate language for representing functional descriptions should at least be:

- Web friendly (based on URIs and globally scalable)
- Unambiguous
- Capable of expressing any existing or future functionality
- Capable of expressing existing and new vocabularies and relationships between functionalities

This is an area that needs further standardization work. One such effort is OWL-S.

3.5 Web Service Semantics

For computer programs to successfully interact with each other a number of conditions must be established:

1. There must be a physical connection between them, such that data from one process may reach another

2. There must be agreement [p.67] (in the sense discussed in **3.3 Using Web Services** [p.66]) on the *form* of the data such as whether the data is lines of text, XML structures, etc.

3. The two (or more) programs must share agreement [p.67] (in the sense discussed in **3.3 Using Web Services** [p.66]) as to the intended meaning of the data. For example, whether the data is intended to represent an HTML page to be rendered, or whether the data represents the current status of a bank account; the expectations and the processing involved in processing the data is different — even if the form of the data is identical.

4. There must be agreement [p.67] (in the sense discussed in **3.3 Using Web Services** [p.66]) as to the implied processing of messages exchanged between the programs. For example, purchase ordering Web service is expected — by the agent that places the order — to process the document containing the purchase order *as a purchase order*, as opposed to simply recording it for auditing purposes.

As we shall see below, more may be required, but for now this list is sufficient.

3.5.1 Message semantics and visibility

The extent to which the shared agreement about the form, structure and meaning of a message is shared *beyond* just the agents involved with the message governs the overall *visibility* of the message semantics. The emphasis on messages, rather than on the actions that are caused by messages, means that SOAs have good visibility: third parties may inspect the flow of messages and have a some assurance as to the services being invoked and the roles of the various parties. This, in turn, means that intermediaries, such as firewalls, are in a better situation for performing their functions. A firewall can look at the message traffic, and at the structure of the message, and make predictable and reasonable decisions about security.

In REST-compliant SOAs, additional visibility comes from the uniform interface semantics, essentially those of the HTTP protocol: an intermediary can inspect the URI of the resource being manipulated, the TCP/IP address of the requester, and the interface operation requested (e.g. GET, PUT, DELETE) and determine whether the requested operation should be performed. The TCP/IP and HTTP protocols have a widely supported set of conventions (e.g. known ports) to support intermediaries, and firewalls, proxies, caches, etc. are almost universal today.

Visibility, however, goes beyond firewalls. In this architecture, instead of emphasising a REST-style uniform interface, we emphasize messages' structure in terms of envelopes, headers and bodies. We enhance visibility architecturally by fostering agreements on particular forms of headers. For example, by having well-known standards that describe the form and interpretation of authentication tokens in headers, we can simultaneously reduce the cost of performing authentication and increase the overall visiblity of the message's semantics: if the authentication aspect of a message can be specified in a standard way then it is easier for a larger number of interested parties to process the message. Furthermore, increased visibility can reduce the cost of entry into a marketplace.

Other potential examples of standardized headers include support for message reliability, support for message correlation, support for process flow and service composition and support for choreography.

This argument can be extended from obvious infrastructure-related processing of messages to more application-related processing of the message. For example, by capturing customer identification in a well-understood header, then all applications capable of processing that header will be able to extract the customer information of a message *independently* of the intended final disposition of the message.

This, in turn, suggests an extremely powerful architectural approach to message processing: different stakeholders in an organization, represented by different applications processing different aspects of messages, can collaborate with a minimal pre-ordained design.

3.5.2 Semantics of the Architectural Models

The different models in the architecture focus on different aspects of the interoperability issues between Web service agents. The Message Oriented Model [p.17] focuses on how Web service agents (requester and provider agents) may interact with each other using a message oriented communication model. The format of messages as XML infosets and the structuring of messages in terms of envelopes, headers and bodies, as described in that model, acts to lay a foundation for the standard comprehension of messages exchanged between Web service agents.

The Service Oriented Model [p.29] builds on the basics of message communication by adding the concept of action [p.30] and service [p.37] . Essentially, the service model allows us to interpret messages as requests for actions and as responses to those requests. Furthermore, it allows an interpretation of the different aspects of messages to be expressed in terms of different expectations, in well understood ways, of the different parts of the message: in effect, an incremental and layered approach to service is possible using well understood headers.

The Resource Oriented Model [p.44] extends this further by adding the concept of resource [p.48] . Resources are important internally to the architecture (a Web service is best understood as a resource in the context of Web service management and in terms of policy management) and externally: resources are an important metaphor for interpreting the interaction between a requester entity [p.36] and a provider entity [p.35] .

3.5.3 The Role of Metadata

An important part of the Service Oriented Architecture approach is the extensive use of metadata. This is important for several reasons: it fosters interoperability by requiring increased precision in the documentation of Web services and it permits tools to give a higher degree of automation to the development of Web services (and hence lowers the cost of deploying same).

The metadata associated with a Web service can be regarded as a partial machine-readable description of the semantics of the Web service. In particular using technologies such as WSDL, a Web service can be described in a machine readable document as to the forms of expected messages, the datatypes of elements of messages and using a choreography [p.32] description language the expected flows of messages between Web service agents.

However, current technologies used for describing Web services are probably not yet sufficient to meet interoperability requirements on a global scale. We see the following areas where increased and richer meta-data would further enhance interoperability:

- It should be possible to identify the real-world entities referenced by elements of messages. For example, when using a credit card to arrange for the purchase of goods or services, the element of the message that contains the credit card information is fundamentally a reference to a real-world entity: the account of the card holder.

  The appropriate technology for this is standardized ontology languages, such as OWL.

- It should be possible to identify the expected effects of any actions undertaken by Web service requester and provider agents. That this cannot be captured by datatyping can be illustrated with the example of a Web service for withdrawing money from an account as compared to depositing money (more accurately, transferring from an account to another account, or vice versa). The datatypes of messages associated with two such services may be identical, but with dramatically different effects: instead of being paid for goods and services, the risk is that one's account is drained instead.

  We expect that a richer model of services, together with technologies for identifying the effects of actions, is required. Such a model is likely to incorporate concepts such as contracts (both legally binding and technical contracts) as well as ontologies of action.

- Finally, a Web service program may "understand" what a particular message means in terms of the expected results of the message, but, unless there is also an understanding of the relationship between the requester entity [p.36] and the provider entity [p.35] , the provider agent may not be able to accurately determine whether the requested actions are *warranted*.

  For example, a provider agent may receive a request to transfer money from one account to another. The request may be valid in the sense that the datatypes of the message are correct, and that the semantic markers associated with the message lead the provider agent to correctly interpret the message as a transfer request. However, the transaction still may not be valid, or fully comprehensible, unless the provider agent can properly identfy the relationship of the requester agent's owner (i.e., the requester entity) to the requested action. Currently, such concerns are often treated simply as security considerations, which they are, in an ad hoc fashion. However, when one considers issues such as delegated authority, proxy requests, and so on, it becomes clear that a simple authentication model cannot accurately capture the requirements.

  We expect that a model that formalizes concepts such as institutions, roles (in business terms), "regulations" and regulation formation will be required. With such a model we should be able to capture not only simple notions of authority, but more subtle distinctions such as the authority to delegate an action, authority by virtue of such delegation, authority to authorize and so on.

3.6 Web Services Security

Threats to Web services involve threats to the host system, the application and the entire network infrastructure. To secure Web services, a range of XML-based security mechanisms are needed to solve problems related to authentication, role-based access control, distributed security policy enforcement, message layer security that accommodate the presence of intermediaries.

At this time, there are no broadly-adopted specifications for Web services security. As a result developers can either build up services that do not use these capabilities or can develop ad-hoc solutions that may lead to interoperability problems.

Web services implementations may require point-to-point and/or end-to-end security mechanisms, depending upon the degree of threat or risk. Traditional, connection-oriented, point-to-point security mechanisms may not meet the end-to-end security requirements of Web services. However, security is a balance of assessed risk and cost of countermeasures. Depending on implementers risk tolerance, point-to-point transport level security can provide enough security countermeasures.

3.6.1 Security policies

From the perspective of this architecture, there are three fundamental concepts related to security: the resources [p.48] that must be secured; the mechanisms by which these resources are secured (i.e., policy guards [p.57]); and policies [p.55] , which are machine-processable documents describing constraints on these resources.

Policies can be logically broken down into two main types: permission policies and obligatory policies. A permission policy concerns those actions and accesses that entities are permitted to perform and an obligation policy concerns those actions and states that entities are required to perform. These are closely related, and dependent: it is not consistent to be obliged to perform some action that one does not have permission to perform. A given policy document is likely to contain a mix of obligation and permission policy statements.

The two kinds of policies have different enforcement mechanisms: a permission guard is a mechanism that can be used to verify that a requested action or access is permitted; an audit guard can only verify after the fact that an obligation has not been met. The precise form of these guards is likely to vary, both with the resources being controlled and with the implementation technologies deployed.

The architecture is principally concerned with the existence of guards and their role in the architecture. In a well engineered system it may be possible to construct guards that are not directly visible to either the requester or provider agents. For example, the unauthorized access threat may be countered by a mechanism that validates the identity of potential agents who wish access the controlled resource. That mechanism is, in turn, controlled by the policy document which expresses what evidence must be offered by which agents before the access is permitted.

A permission guard acts as a guard enabling or disabling access to a resource or action. In the context of SOAP, for example, one important role of SOAP intermediaries is that of permission guards: the intermediary may not, in fact, forward a message if some security policy is violated.

Not all guards are active processes. For example, confidentiality of information is encouraged by encryption of messages. As noted above, it is potentially necessary to encrypt not only the content of SOAP messages but also the identities of the sender and receiver agents. The guard here is the encryption itself; although this may be further backed up by other active guards that apply policy.

3.6.2 Message Level Security Threats

Traditional network level security mechanisms such as Transport Layer Security (SSL/TLS), Virtual Private Networks (VPNs), IPSec (Internet Protocol Security) and Secure Multipurpose Internet Mail Exchange (S/MIME) are point-to-point technologies. Although traditional security technologies are used in Web services security, however, they are not sufficient for providing end-to-end security context, as Web services require more granularities. In general, Web services use a message-based approach that enables complex interactions that can include the routing of messages between and across various trust domains.

Web services face traditional security challenges. A message might travel between various intermediaries before it reaches its destination. Therefore, message-level security is important as opposed to point-to-point, transport-level, security. In Figure 3-3 [p.78] below, the requester agent is communicating with the ultimate receiver through the use of one or more intermediaries. The security context of the SOAP message is end-to-end. However, there may be a need for the intermediary to have access to some of the information in the message. This is illustrated as a security context between the intermediary and the original requester agent, and the intermediary and the ultimate receiver.

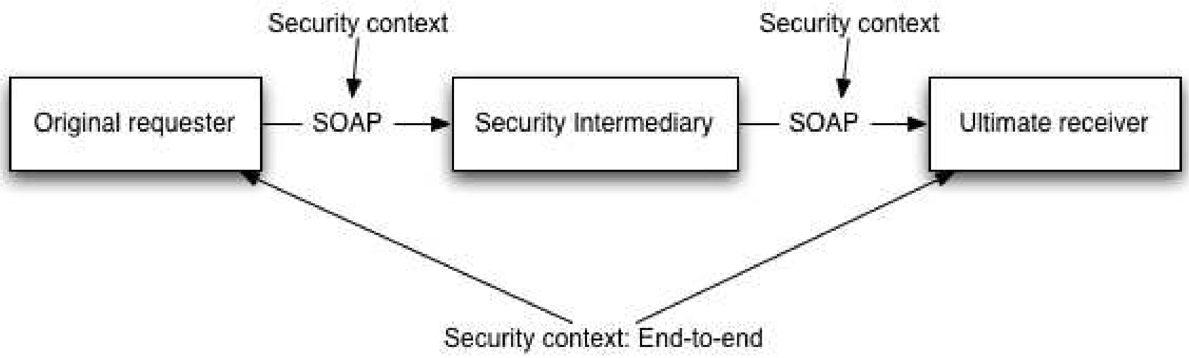

Figure 3-3. End-to-End Security

The threats listed below addresses message security.

3.6.2.1 Message Alteration

These threats affect message integrity, whereby, an attacker may modify parts (or the whole) message. For example, an attacker may delete part of a message, or modify part of a message, or insert extra information into a message. The attacks may affect message header and/or body parts.

An attacker may also affect message integrity by manipulating its attachments. For example, an attacker may delete an attachment, or modify an attachment, or insert an attachment into a message.

3.6.2.2 Confidentiality

In this threat, unauthorized entities obtain access to information with in a message or message parts. For example, an intermediary obtains access to credit card information that was intended for the ultimate recipient.

3.6.2.3 Man-in-the-middle

Man-in-the-middle attacks are also known as bucket-brigade attacks. In this kind of assault it is possible for an attacker to compromise a SOAP intermediary and then intercepts messages between the web service requester and the ultimate receiver. The original parties will think that they are communicating with each other. The attacker may just have access to the messages or may modify them. Mutual authentication techniques can be used to alleviate the threats of this attack.

3.6.2.4 Spoofing

Spoofing is a complex attack that exploits trust relationships. The attacker assumes the identity of a trusted entity in order to sabotage the security of the target entity. As far as the target entity knows, it is carrying on a conversation with a trusted entity. Usually, spoofing is used as a technique to launch other form of attacks such as forged messages. Strong authentication techniques are needed to defend against such attacks.

3.6.2.5 Denial of Service

Denial of service (DoS) attacks focus on preventing legitimate users of a service from the ability to use the service. DoS attacks are easy to implement and can cause significant damage. DoS attacks can disrupt the operation of the agent that is under attack and effectively disconnect it from the rest of the world. DoS attacks can take various forms and target variety of services. DoS attacks exploit weaknesses in the architecture of the system that is under attack. Ironically, security mechanisms themselves add overhead that can be exploited in DoS attacks.

Distributed denial of service (DDoS) attacks uses the resources of more than one machine to launch synchronized DoS attacks on a resource.

3.6.2.6 Replay Attacks

In this attack an intruder intercepts a message and then replays it back to a targeted agent. Appropriate authentication techniques coupled with techniques such as time stamp and sequence numbering the messages can defend against replay attacks.

3.6.3 Web Services Security Requirements

There are many security challenges for adopting Web services. At the highest level, the objective is to create an environment, where message level transactions and business processes can be conducted securely in an end-to-end fashion. There is a need to ensure that messages are secured during transit, with or without the presence of intermediaries. There may also be a need to ensure the security of the data in storage.

The requirements for providing end-to-end security for Web services are summarized in the next sub-sections.

3.6.3.1 Authentication Mechanisms

Authentication is needed in order to verify the identities of the requester and provider agents. In some cases, the use of mutual authentication may be needed since the participants may not necessarily be directly connected by a single hop. For example the participants might be the initial requester and an intermediary. Depending on the security policy it may be possible to authenticate the requester, the receiver or to mandate the use of mutual authentication.

Several methods can be used to authenticate services. Techniques include: passwords, one time pass and certificates. Password-based authentication must use strong passwords. Password authentication alone may be insufficient. Based on vulnerability assessment it may be necessary to combine password authentication with other authentication and authorization process such as certificates, Lightweight Directory Access Protocol (LDAP), Remote Authentication Dial-in User Service (RADIUS), Kerberos, and Public Key Infrastructure (PKI).

3.6.3.2 Authorization

Authorization is needed in order to control access to resources. Once authenticated, authorization mechanisms control the requester access to appropriate system resources. There should be controlled access to systems and their components. Policy determines the access rights of a requester. The principle of least privilege access should be used when access rights are given to a requester.

3.6.3.3 Data Integrity and Data Confidentiality

Data integrity techniques ensure that information has not been altered, or modified during transmission without detection. Data confidentiality ensures that the data is only accessible by the intended parties. Data encryption and digital signature techniques can be used for this purpose.

3.6.3.4 Integrity of Transactions and Communications

This is needed to ensure that the business process was done properly and the flow of operations was executed in a correct manner.

3.6.3.5 Non-Repudiation

Non-repudiation is a security service that protects a party to a transaction against false denial of the occurrence of that transaction by another party. Non-repudiation technologies provide evidence about the occurrence of transactions that that may be used by a third party to resolve disagreement.

3.6.3.6 End-to-End Integrity and Confidentiality of Messages

The integrity and confidentiality of messages must be ensured even in the presence of intermediaries.

3.6.3.7 Audit Trails

Audit trails are needed in order to trace user access and behavior. They are also needed in order to ensure system integrity through verification. Audit trails can be performed by agents. Such agents can play the role of an audit guard that can monitor; watch resources and other agents, validating those obligations that have been established are respected and/or discharged. It is often not possible to prevent the violation of obligations. Instead, if an audit guard detects a policy violation, some form of retribution or remediation must be enacted. The precise forms of this are, of course, beyond the scope of this architecture.

3.6.3.8 Distributed Enforcement of Security Policies

Implementers must be able to define a security policy and enforce it across various platforms with varying privileges.

3.6.4 Security Consideration of This Architecture

Organizations that implement Web services must be able to conduct business in a secure fashion. This implies that all aspects of Web services including routing, management, publication, and discovery should be performed in a secure manner. Web services implementers must be able to utilize security services such as authentication, authorization, encryption and auditing.

Web services messages can flow through firewalls, and can be tunneled through existing ports and protocols. Web services security requires the use of appropriate corporate wide policies that may need to be integrated with external cross-enterprise policy and trust resolution. Organizations may need to implement the capabilities that are listed next.

3.6.4.1 Cross-Domain Identities

Requester and provider agents may communicate with each other using various identity verification schemes from different security domains. Many systems define role based access privileges based on identity. It is important for Web services to be able to support the mapping of identities across multiple domains and even within a single domain.

A provider entity and a requester entity may use their identities to encrypt and sign messages that they exchange. They may exchange identity credentials within a context of initial messages (handshake). That allows further trusted interactions. Service’s identity is optional, and it is perfectly possible to implement a business service without an identity if it always acts on behalf of a requester entity (that is, impersonating the requester entity). Not having a requester entity’s identity translates into anonymous access, which is rarely allowed for business services.

3.6.4.2 Distributed Policies

Security Policies that are associated with requester entity, service and discovery mechanism can be used to define the access privileges of request and responses between parties. These polices can be validated at run time in the context of interaction. Each party in an interaction validates its own policies.

3.6.4.3 Trust Policies

Trust Policies are distributed policies that apply to the environment of the other side's party in an interaction. A requester entity needs to *trust* the environment of a service and the provider entity needs to trust the environment of the requester entity. Trust policies may be recursive — they may be defined against trust policies of involved parties and even whole domains. An example of this is: "I will trust you if you trust my friend and my friend trusts you."

Distributed Identities, Policies and Trust can be described and processed by a machine. For example, an X.509 certificate can be embedded in an message, thus asserting the sender's Identity. A Policy can be described in XML and attached to the service contract. Machines could process, resolve and adjust security based on the given descriptions.

Trust mechanisms can be used to form Delegation and Federation relationships. These mechanisms can be used to facilitate secure interactions between web services across trust boundaries in a distributed fashion.

3.6.4.4 Secure Discovery Mechanism

Secure Discovery Mechanism enforces policies that govern publication and discovery of a service. For example, developers of SOA applications for the procurement department may not be allowed to discover services available in the human resources department, if those developers are not entitled to use human resources services. When publishing a service, an identity is usually necessary to assert service publication policies, except for some cases of peer-to-peer discovery. When a requester entity discovers a service, it may or may not provide an Identity; discovery may well be anonymous.

3.6.4.5 Trust and Discovery

Suppose a requester entity discovers a Web service being offered by a provider entity that was previously unknown to that requester entity. Should the requester entity *trust* that service? If the use of that service requires the requester to divulge sensitive information (such as credit card numbers) to the service then there may be significant risk involved.

This decision — whether or not to trust a particular service — inherently arises when a requester entity chooses a Web service from a previously unknown provider entity. This has ramification in the discovery process, and leads to an important difference between manual discovery and autonomous discovery.

When manual discovery is used, a human makes the judgement (perhaps using other, independently obtained information) of whether to trust and engage a previously unknown service that is discovered. Whereas with autonomous discovery, a machine makes this decision. Since people may not trust a machine to make significant judgement decisions that could put themselves or their organizations at risk, agents performing autonomous discovery are often limited to using private discovery services that list only those services that have been pre-screened and deemed trustworthy by the requester entity. This limited form of autonomous discovery would be more precisely called autonomous *selection*, since the available candidates are already known in advance. Two other ways to mitigate the trust issue in automated discovery include: (1) a agent could autonomously discover candidate Web services and then show them to the human user to choose; or (2) an agent could autonomously discover candidate services and then check a trusted registery for independent information about them, such as a Dunn and Bradstreet quality rating.

3.6.4.6 Secure Messaging

Secure Messaging ensures privacy, confidentiality and integrity of interactions. Digital signatures techniques can be used to help ensure non-repudiation.

Techniques that ensure channel security can be used for securing messages. However, such techniques are applicable in a few limited cases. Examples include a static direct connection between a requester agent and a provider agent. For some applications, such mechanisms can be appropriate. However, in the general case, message security techniques such as encryption and signing of the message payload can be used in routing and reliable messaging.

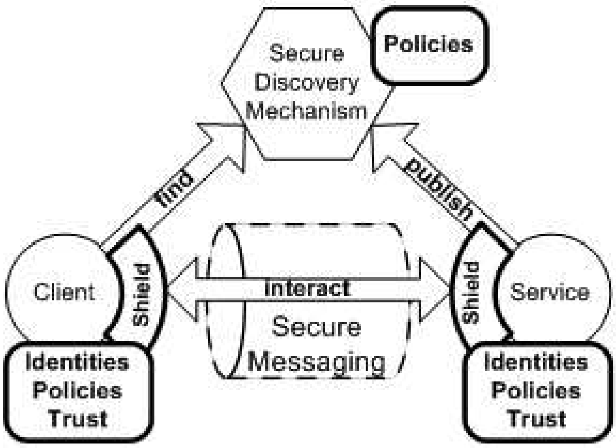

Figure 3-4. Secure Discovery

3.6.5 Privacy Considerations

Issue (privacy_needs_more_work):

The relationship between privacy and Web services technology needs clarification.

There is considerably more complexity to privacy than treated in this section.

Resolution:

None recorded.

Privacy as related to behavior, habits and actions are expressed in terms of policies that the owners of data — typically the users of Web services — have, together with mechanisms necessary to ensure that the owners' rights are respected.

Privacy policies are typically much more of the obligatory form than access control policies. A policy that requires a provider agent to properly propagate P3P tags, for example, represents an obligation on the provider entity. However, it is not possible to prevent a rogue provider agent from leaking private information. Thus, it should be possible to monitor the public actions of the Web service to verify that the P3P tags are propagated appropriately.

Many privacy-related constraints are concerned with maintaining certain kinds of state. For example, a provider entity may have a constraint that any P3P tags associated with a use of one of its Web services are appropriately propagated to third parties. Such a constraint cannot easily expressed in terms of the allowed actions that the provider agent may perform. It is an obligation to ensure that the publicly observable condition (the proper use of P3P tags) is always maintained (presumably maintained in private also). Similarly, a provider agent may link the possible actions that a requester agent may perform to the requester agent maintaining a particular level of secure access (e.g., administrative tasks may only be performed if the request is using secure communications).

3.7 Peer-to-Peer Interaction

To support Web services interacting in a peer to peer style, the architecture must support peer to peer message exchange patterns, must permit Web services to have persistent identity, must permit descriptions of the capabilities of peers and must support flexibility in the discovery of peers by each other.

In the message exchange pattern [p.23] concept we allow for Web services to communicate with each other using a very general concept of message exchange. Furthermore, we allow for the fact that a message exchange pattern can itself be identified — this permits interacting Web service agents to explicitly reference a particular message pattern in their interactions.

A Web service wishing to use a peer-to-peer style interaction may use, for example, a publish-subscribe form of message exchange pattern. This kind of message exchange is just one of the possible message exchange patterns possible when the pattern is explicitly identifiable.

In the agent [p.31] concept we note that agents have identifiers [p.47] . The primary role of an agent identifier is to permit long running interactions spanning multiple messages. Much like correlation, an agent's identifier can be used to link messages together. For example, in a publish and subscribe scenario, a publishing Web service may include references to the Web service that requested the subscription, separately from and additionaly to, the actual recipient of the service.

The agent [p.31] concept also clarifies that a given agent may adopt the role of a provider agent [p.34] and/or a requester agent [p.36] . I.e., these are roles of an agent, not necessarily intrinsic to the agent itself. Such flexibility is a key part of the peer to peer mode of interaction between Web services.

In the service [p.37] concept we state that services have [p.59] a semantics [p.42] that may be identified in aservice description [p.39] and that may be expressed in a service description language [p.39] . This identification of the semantics of a service, and for more advanced agents the description of the service contract itself, permits agents implementing Web services to determine the capabilities of other peer agents. This is turn, is a critical success factor in the architecture supporting peer-to-peer interaction of Web services.

Finally, the fact that services [p.37] have descriptions [p.39] means that these descriptions may be published in discovery agencies [p.46] and also retrieved from such agencies. In effect, the availability of explicit descriptions enables Web services and agents to discover each other automatically as well as having these hard-coded.

3.8 Web Services Reliability

Dealing with errors and glitches is an inescapable fact of life, especially in the context of a global network linking services belonging to many different people. While we cannot eliminate errors and glitches, our goal is to both reduce the the error frequency for interactions and, where errors occur, to provide a greater amount of information about either successful or unsuccessful attempts at service.

Note that our focus on reliability is not really on issues such as syntax errors, or even badly written applications. There is sufficient scope for things to go wrong at the level of network connections being broken, servers being switched off and on in the middle of transactions and even people entering incorrect information in some description file.

In the context of Web services, we can address the issues of reliability at several distinct levels: the reliable and predictable delivery of infrastructure services, such as message transport and service discovery, of reliable and predictable interactions between services, and of the reliable and predictable behavior of individual requester and provider agents. This analysis is generally separate from concerns of fault tolerance, availability or security, but there may of course be overlapping issues.

In the context of security, deliberate acts can cause things to go wrong -- for example, denial of service attacks. This is a sufficiently important case that we deal with it in a separate section [p.77] .

3.8.1 Message reliability

Reliability at the level of messages is often referred to as reliable messaging. In any distributed system there are fundamental limits to the reliability of communication between agents on a public network. However, in practice there are techniques that we can use to greatly increase the reliability of messages, and in those cases where communication fails then we can gain some feedback as to what went wrong.

In more detail, we identify two properties of message sending that are important: the sender of the message would like to be able to determine whether a given message has been received by its intended receiver and that the message has been received exactly once.

Knowing if a message has been received correctly allows the sender to take compensating action in the event the message has not been received. At the very least, the sender may attempt to resend a message that has not been received.

The general goal of reliable messaging is to define mechanisms that make it possible to achieve these objectives with a high probability of success in the face of inevitable but unpredictable network, system and software failures.

This goal may also be examined with respect to whether one wishes to confirm only the receipt of a message, or perhaps also to confirm the validity of that message. Three questions may be asked about message validity:

1. Was the message received the same as the one sent? This may be determined by such techniques as byte counts, check sums, digital signatures.

2. Does the message conform to the formats specified by the agreed upon protocol for the message? Typically determined by automatic systems using syntax constraints (e.g. XML well formed) and structural constraints (validate against one or more XML schemas or WSDL message definitions).

3. Does the message conform to the business rules expected by the receiver? For this purpose additional constraints and validity checks related to the business process are typically checked by application logic and/or human process managers.

Of these, the first is considered to be part of reliable messaging, the last is partly addressed by Web service choreography, but is more closely related to the business expectations of the parties.

The Web services architecture does not itself give specific support for reliable messaging, or for reporting in the event of failure. However, it does give guidance as to how this may be accomplished. The headers and body structure of messages can be utilized: by providing standardized headers to support message auditing then message reliability infrastructures can be deployed in ways that do not need to impact applications and services.

In effect, we can augment message traffic as necessary with specific headers and intermediaries that implement specific semantics for message reliability and reporting in the case that message communication fails. Recall that the architecture does not itself mandate a specific means of message delivery. In fact, we envisage many potential modes of communication, including HTTP, SMTP, JMS based message transports. A given message may even involve multiple kinds of message transport. However, since all messages are structured according to SOAP, we can incorporate overall message reliability within the SOAP message structure.

Message reliability is most often achieved via an acknoweldgement infrastructure, which is a set of rules defining how the parties to a message should communicate with each other concerning the receipt of that message and its validity. WS-Reliability and WS-ReliableMessaging are examples of specifications for an acknowledgement infrastructure that leverage the SOAP Extensibility Model. In cases where the underlying transport layer already provides reliable messaging support (e.g. a queue-based infrastructure), the same level of reliability can be achieved in SOAP by defining a binding that relies on the underlying properties of the transport.

3.8.2 Service reliability

As with message reliability, we are not in a position to be able to offer guarantees that service provider agents and/service requester agents will always perform flawlessly; again, especially in the context of a distributed system over a public network where the different agents may be owned by different people and subject to different policies and management it is not possible to engineer complete service reliability. However, as with message communication we can deploy techniques that greatly enhance reliability and reduce the cost of failure. The principal technique here is one of transactional context management.

Transaction management allows conversations between agents to be managed so that all the parties involved have a greater degree of confidence that the transactions between them progress satisfactorily, and in the event of failure the failure may be identified and transactions either cancelled, rolled back or compensated for.

The architecture does not give specific advice on how to implement transactional reliability. However, again as with message reliability, the combination of the flexible and extensible message structures and the concept of multiple processing of messages (via intermediaries implementing service roles [p.41]) gives us guidance.

One way to incorporate transactional support would be to use standardized headers containing information such as transactional bracket markers and context information that are added to messages exchanged between service requester agents and service provider agents in such a way that *intermediaries* can process messages and monitor transactions in a way that only minimally impacts existing applications. Specialized transactional intermediaries could process messages' transaction-specific headers (such as beginning of transaction, commitment, roll-back and so on) and mark messages that they process with the results; so that applications can respond appropriately.

Related to transactional monitoring is the monitoring of service choreographies. A significant aspect of the specification of the interface of a service is the pattern of message traffic that one might see. For simple cases, this pattern is often very straightforward; however, for most realistic cases, the choreography of services can be very complex. Monitoring that messages are arriving in the order expected is potentially a significant tool in the deployment of reliable services.

Again, as with transactional monitoring, one approach would be to deploy specialized intermediary processes whose specific function is to ensure that the choreographic as well as the static (i.e., message structure) requirements of service usage are being met. This is especially important when the provider agent of a service is not in the same ownership domain as the requester agent.

The key architectural property being used here is the potential deployment of third party services that monitor and process messages in specific role-oriented ways that neither the requesters of services nor the providers of serives needs to be unduly concerned with. This is possible because the architecture does not require messages to be consumed by single agents — nor conversely to be produced by single agents — but allows multiple agents to *collaborate* in the processing of a given message. Each service role establishes a specific functionality, often encoded in specific headers of the messages.

3.8.3 Reliability and management

The reliability of the individual requester and provider agents is out of scope of this architecture as we do not comment on the realization of Web services. In some cases reliability at this level can be enhanced by provider entities adopting deployment platforms that have strong management capabilities. Note that platform manageability represents a *different* perspective than the notion of management identified in Service Management [p.88] (below), which focuses on the manageability of services from a peer or business-partner perspective.

3.9 Web Service Management

Web service management is the management of Web services through a set of management capabilities that enable monitoring, controlling, and reporting of, service qualities and service usage. Such service qualities include health qualities such as availability (presence and number of service instances) and performance (e.g. access latency and failure rates), and also accessibility (of endpoints). Facets of service usage information that may be managed include frequency, duration, scope, functional extent, and access authorization.

A Web service becomes manageable when it exposes a set of management operations that support management capabilities. These management capabilities realize their monitoring, controlling and reporting functions with the assistance of a management information model that models various types of service usage and service quality information associated with management of the Web service. Typical information types include request and response counts, begin and end timers, lifecycle states, entity identifiers (e.g. of senders, receivers, contexts, messages, etc.).

Although the provision of management capabilities enables a Web service to become manageable, the extent and degree of permissible management are defined in management policies that are associated with the Web service. Management policies therefore are used to define the obligations for, and permissions to, managing the Web service.

Just as the Web service being managed needs to have common service semantics that are understood by both the requester and provider entities, Web service management also requires common management semantics, in relation to management policies and management capabilities, to be understood by the requester and provider entities.

Figure 3-5 [p.88] illustrates how the concepts of service [p.37] , policy [p.55] and capability [p.33] defined in this architecture can be applied to management.

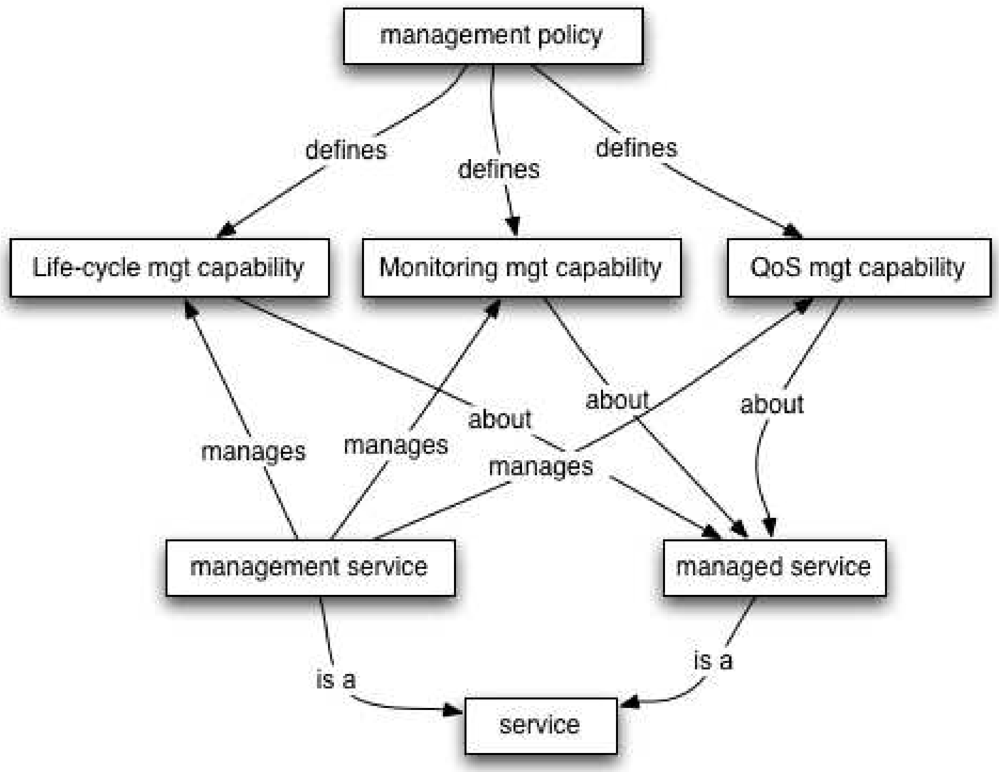

Figure 3-5. Management Concepts and Relationships

More detailed information about Web services management is available in the management documents that were produced by the Management Task Force of this Working Group.

3.10 Web Services and EDI: Transaction Tracking

One of the basic assumptions that many people make about the role of Web services is that they will be used for functions similar to those presently provided by Electronic Data Interchange (EDI). Since EDI is a well established technology, it is useful to examine the expectations that current EDI users may have for a technology that is to be used as a replacement. That is, what do they do now that they will also expect from a new technology? The most basic of these expectations concern security, message reliability and a function that we will call "tracking". Since security and message reliability are covered elsewhere in this document, this section will focus on tracking.

3.10.1 When Something Goes Wrong

What happens when a transaction goes awry for reasons other than the loss of a message or security violations? Although it is possible, and useful, to automate safeguards both at the protocol and application level, experience indicates that there is a virtually limitless variety of ways that business transactions can fail. Informal interviews with current EDI practitioners have indicated that in practice the 80-20 of what EDI people actually do is involved with the issue of finding out what has actually happened in transactions when something has gone wrong. For example, such an interaction may start with a phone call that goes something like, "Why haven’t you paid us?" and continues, "We think we have paid you". In these cases there is often a good faith desire on both sides to figure out what has happened and comply with the requirements of the transaction, but the information that people are working with may differ and coming to a common understanding can take some work.

3.10.2 The Need for Tracking

In current EDI operations, many of the questions that must be answered in these cases can be handled in an automated fashion by the vendor of the proprietary network used in the transactions. For example, if company A asks if the invoice to company B was delivered, the vendor can access its records, probably from a central repository, and respond, "The message was delivered to company B’s mailbox on Dec 24 but they have not as yet downloaded the message". Queries of this sort are relatively easy to satisfy in this environment because the vendor is in control of all aspects of the communication. In a Web services scenario, where the transactions take place in a distributed environment, with no central authority, some other means must replace the current automated queries to the EDI vendor or this important tracking capability will be lost.

One possibility would be to provide some kind of uniform tracking interface. The basic requirement here is for companies that are cooperating in a business transaction to find out at any time what is the status and history of the transaction. Significant complexity is added by the fact that multiple companies may be involved. That is, company A may initiate a transaction by sending a message to B, but the process may then involve messages between B and C. In some cases the interactions between B and C may be known to A (as opposed to being part of B’s internal process that is opaque to A). It is not immediately clear whether this should be handled by A querying both B and C, or if a responding to a query from A to B should carry with it the obligation to query C and return the results. This is presumably an issue which must be ironed out in the creation of the specification(s) for the uniform interface.

3.10.3 Examples of Tracking

As illustrations, here are some of the typical queries that A might send to B or C. Web Services Usage Scenarios [WSAUS] [p.96] contains additional examples.

1. (Query to B) Did you receive and process message M from A?
2. (Query to B) Please return copies of all messages associated with Transaction T.
3. (Query to B) Please return copies of all messages between A and B in a given time range.
4. (Query to C) Please return all messages associated with transactions involving A during a given time frame (including messages between B and C related to transactions in which A is involved).
5. (Query to B) Please return copies of messages between B and other companies involved with a transaction (or all transactions in a date range).

Of course, in all cases, the party performing the query must be authorized to receive the information.

Current EDI practices may automate some of these queries; others may involve manual processing. In general, however, there are significant cost savings to be realized by automating as much of the process as possible.

3.10.4 Requirements for Effective Tracking

In order to help automate the tracking process, there are various requirements, some of which are probably achievable using current or planned specifications and others of which may require new ones:

1. A uniform, interoperable interface for tracking queries, so that company A can send a standard query to all of its business partners. This interface should be associated with the functional Web services interfaces. For example, such an interface might be implemented as part of a management interface.

2. Standard identifiers for transactions and individual messages that are necessary to define the queries. Note that some of these queries involve identifiers of participants in a transaction other than sender and receiver of a particular message. There are clearly aspects of this requirement that are related to the choreography domain.

3. Policies controlling whether party A is authorized to make tracking queries to B. There may be several variants of such policies: e.g. a can query B about messages directly between A and B but not messages between B and C associated with transactions involving A and so on. It may be possible to establish these policies using mechanisms currently available or under development in the security or policy domain, or there may be transactional aspects to these policies that are not currently being considered.

4. A method to establish the trust relationships necessary to implement the policies in 3.

3.10.5 Tracking and URIs

One of the important connections between Web services architecture and Web architecture as a whole, is the common use of URIs. Although URIs are important to many aspects of Web services, it is particularly worth noting their potential role and benefit in indentifying and tracking transactions in Web services.

As a simple example to illustrate this benefit, suppose URIs are used as transaction identifiers. Each time a new transaction is initiated, a new URI is generated to unambiguously identify that transaction, much like a primary key in a database. However, while a database key may only be unambiguous within a particular database, a URI is *globally unambiguous*, which means that it can be conveniently transmitted to others without loss or confusion of meaning.

Furthermore, a URI may be *dereferenceable*: If the URI also represents the location of a document (or a dynamic query into a database), it could act as a convenient link for determining the status or history of that transaction, provided the user is authorized to access such information. (Security mechanisms will need to ensure that a tracking URI cannot be dereferenced without proper authority and privacy controls, but the use of URIs is largely orthogonal to this requirement.)

The potential value of this dual use of URIs — both as globally unambiguous identifiers and as universally dereferenceable links — is one of the most fundamental and important insights in the architecture of the Web. Because the Web services architecture builds on the Web architecture, Web services can leverage the benefits of clarity, simplicity, universality and convenience that this use of URI offers.

This is not to say that Web services tracking *must* be done using URIs in this way. Indeed, there are other ways tracking can be performed, and any engineering design must take many factors into consideration. Rather, the point is to illuminate the fact that, because Web services architecture is based on Web architecture, Web services have the *possibility* of taking advantage of this use of URIs.

4 Conclusions

4.1 Requirements Analysis

We believe this architecture substantially meets the requirements defined in [WSA Reqs] [p.96] , with the exception of security and privacy. Although this architecture contains substantial material that lays the foundation for addressing these, more work is needed. The Working Group wanted to do more to address these but was not able to do so with the available resources.

4.2 Value of This Work

This architecture lays the conceptual foundation for establishing interopable Web services. The architecture identifies a number of important abstractions and their interdependencies.

Contributions of this work include the following:

- Provides a coherent framework that allows specific technologies to be considered in a logical context and facilitates the work of specification writers and architects.
- Defines a consistent vocabulary, including an authoritative definition of "Web service" that has received widespread acceptance in industry [WS Glossary] [p.96] .
- Defines an OWL ontology of Web services architecture concepts [OWLO] [p.96] .
- Distinguishes SOA from distributed object architecture.
- Clarifies the architectural relationship between the Web and Web services
- Clarifies the relationship between Web services and REST.
- Identifies gaps and inconsistencies in existing Web services specifications.
- Identifies the role of semantics and the need for machine-processable semantics and ontologies in Web services

4.3 Significant Unresolved Issues

(See also the issues list previously maintained by the Working Group.)

1. What is the difference between an MEP and a Choreography? [See **2.3.1.7 Message Exchange Pattern (MEP)** [p.23]]
2. What should be the representation returned by an HTTP "GET" on a Web service URI? [See **2.3.2.10 Service** [p.37]]
3. Should URIs be used to identify Web services components, rather than QNames? [See **2.3.3.3 Identifier** [p.47]]
4. The relationship between privacy and Web services technology needs clarification. [See **3.6.5 Privacy Considerations** [p.83]]
5. SOAP 1.2 and this architecture introduce the concept of "intermediaries", but this concept is not represented in WSDL 2.0.
6. What happens if two logical WSDL documents define the same service differently? [See email thread available at http://lists.w3.org/Archives/Public/www-ws-desc/2003Dec/0045.html]
7. The relationship between conversations, correlations and transactions and choreography is unclear and needs more work.
8. There is a need for consistent tracking mechanisms in Web services. [See **3.10 Web Services and EDI: Transaction Tracking** [p.89]]

A Overview of Web Services Specifications (Non-Normative)

An annotated list of Web services specifications (available at http://lists.w3.org/Archives/Public/www-ws-arch/2004Feb/0022.html) was produced independently by two members of this Working Group, Roger Cutler and Paul Denning. *Although this Working Group feels that this is a useful list, the opinions expressed therein are the personal opinions of those authors and do not represent the consensus of the Working Group.*

B An Overview of Web Services Security Technologies (Non-Normative)

This section attempts to provide a non-exhaustive description of current available work around Web services security relevant to the requirements and solutions presented in **3.6 Web Services Security** [p.77] .

Note that although these technologies build on existing security technologies, they are relatively new and need to be fully tested in actual deployment scenarios.

B.1 XML-Signature and XML-Encryption

XML signatures are designed for use in XML transactions. It is a standard that was jointly developed by W3C and the IETF (RFC 2807, RFC 3275). The standard defines a schema for capturing the result of a digital signature operation applied to arbitrary data and its processing. XML signatures add authentication, data integrity, and support for non-repudiation to the signed data.

XML Signature has the ability to sign only specific portions of the XML tree rather than the complete document. This is important when a single XML document may need to be signed by multiple times by a single or multiple parties. This flexibility can ensure the integrity of certain portions of an XML document, while leaving open the possibility for other portions of the document to change. Signature validation mandates that the data object that was signed be accessible to the party that interested in the transaction. The XML signature will generally indicate the location of the original signed object.

XML Encryption specifies a process for encrypting data and representing the result in XML. The data may be arbitrary data (including an XML document), an XML element, or XML element content. The result of encrypting data is an XML Encryption element which contains or references the cipher data.

B.2 Web Services Security

Developed at OASIS, Web Services Security (WSS) defines a SOAP extension providing quality of protection through message integrity, message confidentiality, and message authentication. WSS mechanisms can be used to accommodate a wide variety of security models and encryption technologies.

The work provides a general mechanism for associating security tokens with messages. The specification does not require a specific type of security token. It is designed to support multiple security token formats. WSS describes how to encode binary security tokens. The specification describes how to encode X.509 certificates and Kerberos tickets. Additionally, it also describes how to include opaque encrypted keys.

The WSS specification defines an end to end security framework that provides support for intermediary security processing. Message integrity is provided by using XML Signature in conjunction with security tokens to ensure that messages are transmitted without modifications. The integrity mechanisms can support multiple signatures, possibly by multiple actors. The techniques are extensible such that they can support additional signature formats. Message confidentiality is granted by using XML Encryption in conjunction with security tokens to keep portions of SOAP messages confidential. The encryption mechanisms can support operations by multiple actors.

B.3 XML Key Management Specification (XKMS) 2.0

XKMS 2.0 is an XML-based way of managing the Public Key Infrastructure (PKI), a system that uses public-key cryptography for encrypting, signing, authorizing and verifying the authenticity of information in the Internet. It specifies protocols for distributing and registering public keys, suitable for use in conjunction with the proposed standard for XML Signature and XML Encryption.

XKMS allow implementers to outsource the task of key registration and validation to a "trust" utility. This simplify implementation since the actual work of managing public and private key pairs and other PKI details is done by third party.

An XKMS trust utility works with any PKI system, passing the information back and forth between it and the Web service. Since the trust utility does the work, the Web service itself can be kept simple. XKMS is a W3C specification.

B.4 Security Assertion Markup Language (SAML)

SAML is an Extensible Markup Language standard (XML) that supports Single Sign On. SAML allows a user to log on once to a Web site and conduct business with affiliated but separate Web sites. SAML can be used in business-to-business and business-to-consumer transactions.

There are threes basic SAML components: assertions, protocol, and binding. Assertions can be one of three types: authentication, attribute, and authorization. Authentication assertion validates the identity of the user. The attribute assertion contains specific information about the user. While, the authorization assertion identifies what the user is authorized to do.

The protocol defines how SAML request and receives assertions. There are several available binding for SAML. There are bindings that define how SAML message exchanges are mapped to SOAP, HTTP, SMTP and FTP among others. The Organization for the Advancement of Structured Information Standards (OASIS) is the body developing SAML.

B.5 XACML: Communicating Policy Information

XACML is an Extensible Markup Language standard (XML) based technology, developed by Organization for the Advancement of Structured Information Standards (OASIS) for writing access control polices for disparate devices and applications.

XACML includes an access control language and request/response language that let developers write policies that determine what users can access on a network or over the Web. XACML can be used to connect disparate access control policy engines.

B.6 Identity Federation

The Liberty Alliance is defining specifications dealing with various aspects of identity. Their phase 2 work is grouped into three categories: ID-FF, ID-WSF, and ID-SIS.

ID-FF (Identity Federation Framework) discusses how businesses or organizations can be affiliated into circles of trust and trust relationships. ID-FF includes several normative specifications, which in turn make normative references to SAML.

ID-WSF (Identity Web Services Framework) is a set of specifications for creating, discovering, using, and updating various aspects of identities through a particular type of web service known as an Identity Service. ID-WSF builds on ID-FF. A user (Principal) may register with several Identity Services. A prominent part of ID-WSF is a discovery service for locating an Identity Service for a given user (Principal). ID-SWF also defines a Data Services Template. ID-WSF has also defined a draft specification for an approach to negotiating an authentication method using SOAP messages to identify SASL mechanisms (RFC 2222).

Note that WS-Security specifically states that establishing a security context or authentication mechanisms is outside its scope. ID-WSF may fill this void. However, WS-Security also defines a Username Token Profile, which could be used as an authentication mechanism. Potentially, Liberty ID-WSF could be used to negotiate the use of WSS Username Token Profile as the authentication mechanism. Currently, WSS Username Token Profile is not registered in IANA's SASL Mechanisms collection.

ID-SIS (Identity Service Instance Specifications) defines profiles for particular types of Identity Services. These profiles conform to the ID-WSF Data Services Template. Liberty has defined two such profiles. The Employee Profile (ID-SIS-EP) defines how to query and modify information associated with a Principal in the context of their employer. The Personal Profile (ID-SIS-PP) defines how to query and modify identity information for Principals themselves.

C References (Non-Normative)

Dist Comp
: *A Note on Distributed Computing*, S. C. Kendall, J. Waldo, A. Wollrath, G. Wyant, November 1994 (See http://research.sun.com/techrep/1994/abstract-29.html.)

Fielding
: *Architectural Styles and the Design of Network-based Software Architectures*, PhD. Dissertation, R. Fielding, 2000 (See http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.)

OWLO
: *OWL Ontology of Web service architecture concepts*, M. Paolucci, N. Srinivasan, K. Sycara (See http://www.w3.org/2004/02/wsa/.)

RFC 2396
: *Uniform Resource Identifiers (URI): Generic Syntax*, IETF RFC 2396, T. Berners-Lee, R. Fielding, L. Masinter, August 1998 (See http://ietf.org/rfc/rfc2396.txt.)

SOAP 1.2 Part 1
: *SOAP Version 1.2 Part 1: Messaging Framework*, W3C Recommendation, M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, H. Nielsen, 24 June 2003 (See http://www.w3.org/TR/2003/REC-soap12-part1-20030624/.)

SOAP 1.2 Part 2
: *SOAP Version 1.2 Part 2: Adjuncts*, W3C Recommendation, M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, H. Nielsen, 24 June 2003 (See http://www.w3.org/TR/2003/REC-soap12-part2-20030624/.)

Web Arch
: *Architecture of the World Wide Web, First Edition*, W3C Working Draft, I. Jacobs, 9 December 2003 (See http://www.w3.org/TR/2003/WD-webarch-20031209/.)

WS Glossary
: *Web Services Glossary*, W3C Working Group Note, H. Haas, A.Brown, 11 Febuary 2004 (See http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/.)

WSA Reqs
: *Web Services Architecture Requirements*, W3C Working Group Note, D. Austin, A. Barbir, C. Ferris, S. Garg, 11 February 2004 (See http://www.w3.org/TR/2004/NOTE-wsa-reqs-20040211.)

WSAUS
: *Web Services Architecture Usage Scenarios*, W3C Working Group Note, H. He, H. Haas, D. Orchard, 11 February 2004 (See http://www.w3.org/TR/2004/NOTE-ws-arch-scenarios-20040211/.)

WSDL 2.0 Part 1
Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, W3C Working Draft, R. Chinnici, M. Gudgin, J-J. Moreau, J. Schlimmer, S. Weerawarana, 10 November 2003 (See http://www.w3.org/TR/2003/WD-wsdl20-20031110/.)
XML 1.0
Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler. 6 October 2000 (See http://www.w3.org/TR/2000/REC-xml-20001006.)
XML Infoset
XML Information Set, W3C Recommendation, J. Cowan, R. Tobin, 24 October 2001 (See http://www.w3.org/TR/2001/REC-xml-infoset-20011024/.)

D Acknowledgments (Non-Normative)

This document has been produced by the Web Services Architecture Working Group . The chairs of this Working Group were Chris Ferris (until July 2002), Michael Champion (starting July 2002) and Dave Hollander (starting July 2002). The chairs also wish to thank the following (listed in alphabetic order) for their substantial contributions to the final documents: Daniel Austin, Mark Baker, Abbie Barbir, David Booth, Martin Chapman, Ugo Corda, Roger Cutler, Paul Denning, Zulah Eckert, Chris Ferris, Hugo Haas, Hao He, Yin-Leng Husband, Mark Jones, Heather Kreger, Michael Mahan, Frank McCabe, Eric Newcomer, David Orchard, Katia Sycara.

Members of the Working Group are (at the time of writing, and in alphabetical order): Geoff Arnold (Sun Microsystems, Inc.), Mukund Balasubramanian (Infravio, Inc.), Mike Ballantyne (EDS), Abbie Barbir (Nortel Networks), David Booth (W3C), Mike Brumbelow (Apple), Doug Bunting (Sun Microsystems, Inc.), Greg Carpenter (Nokia), Tom Carroll (W. W. Grainger, Inc.), Alex Cheng (Ipedo), Michael Champion (Software AG), Martin Chapman (Oracle Corporation), Ugo Corda (SeeBeyond Technology Corporation), Roger Cutler (ChevronTexaco), Jonathan Dale (Fujitsu), Suresh Damodaran (Sterling Commerce(SBC)), James Davenport (MITRE Corporation), Paul Denning (MITRE Corporation), Gerald Edgar (The Boeing Company), Shishir Garg (France Telecom), Hugo Haas (W3C), Hao He (The Thomson Corporation), Dave Hollander (Contivo), Yin-Leng Husband (Hewlett-Packard Company), Mario Jeckle (DaimlerChrysler Research and Technology), Heather Kreger (IBM), Sandeep Kumar (Cisco Systems Inc), Hal Lockhart (OASIS), Michael Mahan (Nokia), Francis McCabe (Fujitsu), Michael Mealling (VeriSign, Inc.), Jeff Mischkinsky (Oracle Corporation), Eric Newcomer (IONA), Mark Nottingham (BEA Systems), David Orchard (BEA Systems), Bijan Parsia (MIND Lab), Adinarayana Sakala (IONA), Waqar Sadiq (EDS), Igor Sedukhin (Computer Associates), Hans-Peter Steiert (DaimlerChrysler Research and Technology), Katia Sycara (Carnegie Mellon University), Bryan Thompson (Hicks & Associates, Inc.), Sinisa Zimek (SAP).

Previous members of the Working Group were: Assaf Arkin (Intalio, Inc.), Daniel Austin (W. W. Grainger, Inc.), Mark Baker (Idokorro Mobile, Inc. / Planetfred, Inc.), Tom Bradford (XQRL, Inc.), Allen Brown (Microsoft Corporation), Dipto Chakravarty (Artesia Technologies), Jun Chen (MartSoft Corp.), Alan Davies (SeeBeyond Technology Corporation), Glen Daniels (Macromedia), Ayse Dilber (AT&T), Zulah Eckert (Hewlett-Packard Company), Colleen Evans (Sonic Software), Chris Ferris (IBM), Daniela Florescu (XQRL Inc.), Sharad Garg (Intel), Mark Hapner (Sun Microsystems, Inc.), Joseph Hui (Exodus/Digital Island), Michael Hui (Computer Associates), Nigel Hutchison (Software AG), Marcel

Jemio (DISA), Mark Jones (AT&T), Timothy Jones (CrossWeave, Inc.), Tom Jordahl (Macromedia), Jim Knutson (IBM), Steve Lind (AT&T), Mark Little (Arjuna), Bob Lojek (Intalio, Inc.), Anne Thomas Manes (Systinet), Jens Meinkoehn (T-Nova Deutsche Telekom Innovationsgesellschaft), Nilo Mitra (Ericsson), Don Mullen (TIBCO Software, Inc.), Himagiri Mukkamala (Sybase, Inc.), Joel Munter (Intel), Henrik Frystyk Nielsen (Microsoft Corporation), Duane Nickull (XML Global Technologies), David Noor (Rogue Wave Software), Srinivas Pandrangi (Ipedo), Kevin Perkins (Compaq), Mark Potts (Talking Blocks, Inc.), Fabio Riccardi (XQRL, Inc.), Don Robertson (Documentum), Darran Rolls (Waveset Technologies, Inc.), Krishna Sankar (Cisco Systems Inc), Jim Shur (Rogue Wave Software), Patrick Thompson (Rogue Wave Software), Steve Vinoski (IONA), Scott Vorthmann (TIBCO Software, Inc.), Jim Webber (Arjuna), Prasad Yendluri (webMethods, Inc.), Jin Yu (MartSoft Corp.) .

The people who have contributed to discussions on the www-ws-arch public mailing list are also gratefully acknowledged.