Service Framework Specification, Part 1
Version 2.0

Arindam Banerji, Claudio Bartolini, Dorothea Beringer,
Abdel Jabbar Boulmakoul, Svend Frolund, Kannan Govindarajan (Ed.),
Alan Karp, Michal Morciniec, Gregory Pogossiants, Chris Preist,
Shamik Sharma, David Stephenson, Scott Williams
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-138
June 7th , 2001*

The service framework specification is a layered specification that enables interoperability through the use of XML and formal definitions of interactions amongst services. It also provides mechanisms for dynamically discovering services as well as interacting with them through the exchange of XML documents. The specification has two parts: the first part specifies more horizontal infrastructural services such as messaging, service definition, transactions, management, vocabularies, and discovery of services. The second part focuses on higher level business interactions such as negotiation and contract formation.

* Internal Accession Date Only Approved for External Publication

Abstract

The web is evolving from a platform for serving web pages to a platform that hosts applications that interact using standard protocols. In addition, XML has emerged as a way to enable loose coupling amongst these applications or services. These technologies essentially enable a world of dynamic web services or e-services that are created and used in a dynamic fashion. Web services architectures provide a framework for creating, and deploying loosely coupled applications. One of the consequences of the loose coupling is that any entity that a web service may interact with may not exist at the point of time the web service is developed. New web services may be created dynamically just as new web pages are added to the web and web services should be able to discover and invoke such services without recompiling or changing any line of code. A fundamentally different component model is required for modeling web services. This is because the assumptions that are made by traditional distributed component models are violated by web services. In addition, we believe that a comprehensive web services platform has at least three related technologies:

1. The technologies that define the hosting platform that hosts services. Service providers typically will host their services on this hosting platform.
2. The technologies that define the hub that allows services to dynamically discover other services and establish trust in the context of the community. This hub technology potentially is compatible with other hub-like efforts such as UDDI (www.uddi.org).
3. The technologies that define the standard conventions that ensure that services can inter-operate with each other irrespective of their implementations.

The hosting platform provides, among other things, technologies that are required to model existing business asset/process as a web-service, allows clients of web services to invoke services, etc. The hub provides technologies for web services to be described, discovered, etc., and the standard conventions specify the things that have to be standardized so that web services hosted on various web service platforms inter-operate.

The service framework specification is a layered specification that enables interoperability through the use of XML and formal definitions of interactions amongst services. It also provides mechanisms for dynamically discovering services as well as interacting with them through the exchange of XML documents. The specification has two parts: the first part specifies more horizontal infrastructural services such a messaging, service definition, transactions, management, vocabularies, and discovery of services. The second part focuses on higher level business interactions such as negotiation and contract formation.

Table of Content

1 Introduction

1.1 About this document

This document contains the specification of SFS version 2.0. It is meant for architects and software engineers who are interested in the technical details of the SFS. Though the main concepts are the same in SFS 1.1 and SFS 2.0, there have been various significant changes. Some of these are:

1. The message format has been defined to work with SOAP 1.1
2. The conversation definition language has changed from XMI based finite-state machine description to a new language that is used to specify interactions and conversations.
3. The notion of vocabularies as a mechanism for defining meta-data of e-services has been introduced and the matchmaking section has been updated accordingly.
4. The atomocity of interactions and conversations is addressed with the transaction specification.

In this introductory chapter, we motivate the need for a service framework specification by introducing the notion of e-services. We briefly discuss how e-services cannot be modeled through traditional distributed component architectures. We then outline HPs service framework specification and briefly discuss how it addresses the issues that are required for enabling the world of e-services.

1.2 Purpose of the Service Framework Specification

The E-services Vision

The internet is evolving beyond being an infrastructure for putting up electronic store fronts. It is beginning to provide a channel for companies to automate their business processes, and their relationships with their suppliers and customers. This new infrastructure does not require a lot of investment in value added networks as was the case for EDI based business automation solutions. However, most infrastructures that exist today are proprietary, and though there are a few standardization efforts for specific business processes as witnessed by RosettaNet (www.rosettanet.org), most implementations do not inter-operate. For instance, in order to inter-operate with a partner who has software provided by a specific company, one at least needs some software from the same company in order to interoperate. Furthermore, the recent explosion of marketplaces, each with its own XML schemas, has led to fragmentation. This fragmentation will eventually lead to companies having to invest in different technology stacks for participating in various marketplaces. For example, if a company buys steel from one market place and plastics from another marketplace, it may potentially need to invest in two completely different technology stacks for participating in the two marketplaces if the technologies powering the marketplaces are different.

HP's e-service vision addresses this problem by allowing any existing business asset/process to be modeled as an e-service. In addition, it also specifies other conventions that enable interoperability amongst services. Essentially, there are three pieces of technology that need to be specified for providing the infrastructure for HPs e-services vision. These are:

1. The specification that provides the technical conventions supported by e-services.
2. The hosting platform that hosts these e-services and provides the communication between these e-services potentially across enterprises.
3. The hub that serves as a aggregation point for the e-services and also helps in establishing a web of trust for the e-services interacting at the hub.

Exposing business assets as services and standardizing interactions amongst services has the added advantage that any enterprise can out-source parts of its operation that it does not have expertise in. In addition, since the vision of e-services enables e-services to dynamically find new e-services that it can interact with, enterprises can find new providers for the service relatively quickly. A specific application of this dynamism is in the e-procurement arena. For example, the average sourcing/procurement cycle in enterprises is of the order of 3-4 months (cite appropriate study). Of this time, about 50% of the time is spent in identifying the appropriate suppliers, about 20% of the time in handling the RFQ (request for quotes) process, and an additional 10% of the time is spent in negotiating the appropriate deal. The ability to dynamically find suppliers can translate to significant time savings, and therefore to lowering of costs. Essentially, the procurement and fulfillment business process are modeled as services, and a hub is the aggregation point for the services. In such an architecture, finding a new supplier is the same as finding the fulfillment service of the supplier at the hub. HPs e-services vision enables such a dynamic world by allowing business processes to be modeled as e-services, by providing a platform for hosting such e-services, by defining the technical conventions that enable the interoperability between e-services, and by defining a hub, or aggregation mechanism for eco-systems of e-services to be built.

The complete e-service interaction cycle has the following phases:

1. Identify potential e-services to interact with
2. Negotiate terms and conditions of e-service interaction
3. Formalize a contract based on the negotiations
4. Execute the interaction
5. Monitor the execution of the interaction in order to determine compliance with contract.

The Service Framework specification specifies conventions that can be used in each phase of the interaction outlined above. In addition, it also specifies other conventions that are required for these services to communicate with each other.

Traditional component models: Why they dont suffice.

E-services cannot be modeled using traditional software component models that provide an infrastructure for enterprise-level distributed application development. Such traditional frameworks are not suited to deploying services across organizational boundaries due to the following characteristics of the inter-enterprise e-services:

- **Loose Coupling among E-services:** Changes to the e-service should not require re-installation of software components by the users of the e-service.

- **Dynamic binding:** Typically, application designers bind software components to one another at development time. The notion of services implemented and provided by different service providers is not accounted for. Yet we must enable easy changes to the services we are using, easy discovery of new services, of new capabilities of existing services, and of new binding or location information of services.
- **Document Exchange Model:** Traditional component frameworks support a network-object model of interaction in which objects of strictly defined types are transferred between components using a request-response interaction pattern. Cross-organizational business interactions do not fit this framework well for two reasons. The interfaces of services may need to be changed in ways that cannot be captured by simple extensions. This precludes the use of object inheritance to support the inter-operability in presence of change. Secondly, interactions can be long lived. Therefore, asynchronous exchange of XML documents is better suited for cross-organizational business transactions.
- **Differing semantics**: The interpretation of the data communicated among enterprises is different for each enterprise. For instance, the address field of a purchase order may have different significance for the parties. If a uniform object model is used, the semantics of data often tends to be similar or homogeneous contributing to tighter coupling.
- **Distributed securit**y: Security responsibilities are split amongst the enterprises. Each enterprise manages its end of the security infrastructure independently.
- **Disparate technology stacks** are used. Each enterprise decides on the computing infrastructure independently taking into consideration many factors.
- **Firewalls**: Interactions need to traverse corporate firewalls. Traditional distributed systems are tuned for applications that are deployed within the enterprise.

Communication requires standards

In order for e-services to communicate with each other they have to agree on the technology, standards and protocols for communication. They also need to agree on the syntax and semantics of data they are going to exchange. However, the data that they exchange can be classified into infrastructure parts that the infrastructure uses in order to route the message and application specific parts that the infrastructure does not inspect.

There are various reasons for a standard service framework specification:

- Electronic market places and service portals need to be built fast based using off-the shelf components, and the various market places and portals should be able to interact seamlessly.
- E-services need to be developed and deployed fast and with ease.
- Standard bodies are trying to standardize business processes in order to enable companies to interact in B2B environments.

In order to achieve these goals, it is necessary specify standards that define the following:

1. The basic concepts and mechanisms of an system of e-services
2. The interaction protocols used to communicate between e-services, and
3. The language used to define the public interface of e-services.

The Service Framework Specification from HP

The E-Speak Service Framework Specification introduces a uniform service model in order to address some of the interoperability issues concerned with application integration across enterprises. This uniform service model makes possible dynamic interactions between e-services. The SFS defines the technical conventions necessary for the creation, deployment, and interaction of e-services. It also enables the formation of dynamic electronic business relationships through negotiation, contract formation and business process integration. This common business and technology level interaction framework builds upon several emerging industry standards such as XML, MIME, SOAP, and UDDI. By defining a uniform, non-proprietary, and extensible means for interaction between Internet services, business collaboration in the new economy through direct interaction, auction sites, exchanges or aggregators can be unified.

1.3 Overview of SFS 2.0

When architecting e-services systems we have to clearly distinguish between the external visible behavior of an e-service, i.e., its external interaction protocol, and the implementation of the e-service, i.e., its actual internal implementation. This concept is often referred to by other terms such as the interface of an e-service and its realization, or the public process of the e-service and its private process. The external interaction protocol and the internal implementation of an e-service are closely linked as the internal implementation should match the specified external behavior. SFS provides standards for defining the external interaction protocol of e-services. The mapping between the external interaction protocol and the internal implementation is not specified in the Service Framework Specification.

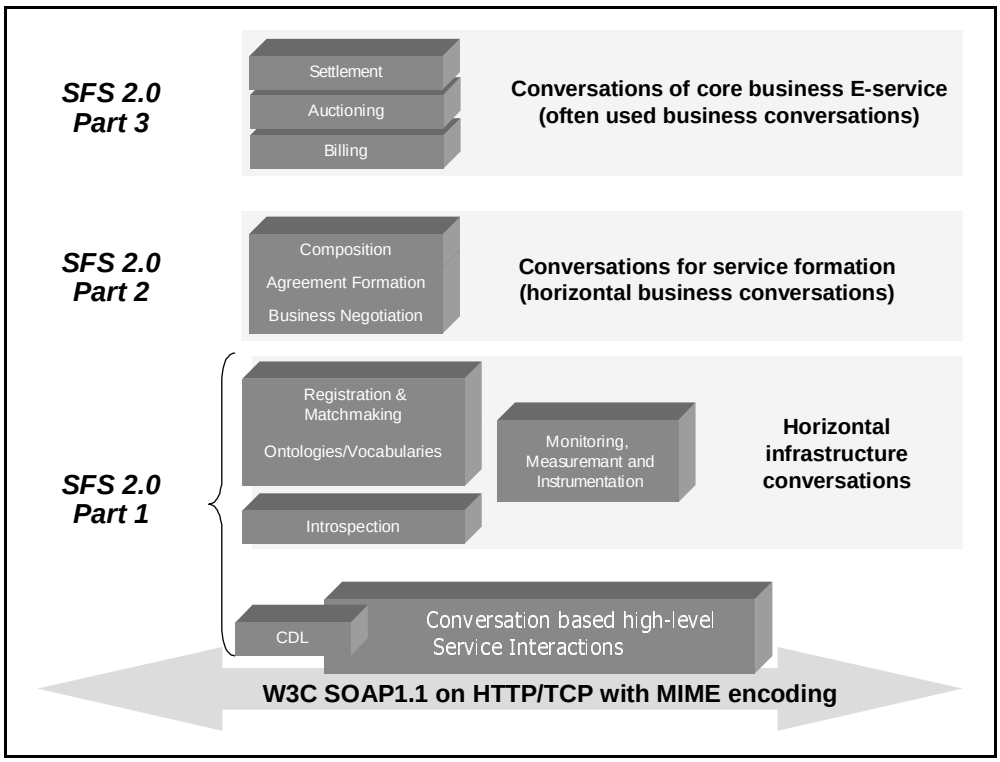


Figure 1: Specifications for service interaction

The various specifications for service interactions can be thought of as a stack, where each layer builds on the previous one as shown in figure 1. The lowest layer defines the standard transport and messaging protocols used, and the language for defining the payloads of messages and their orders. The next layer, the horizontal infrastructure conversations, defines the conventions that allow a basic system of e-services to function. This involves the functionality required for offer registration, match-making, discovery, transactions, monitoring and management. The higher layers contain horizontal business conversations. The conversations of service formation layer is needed where services need to negotiate contracts with each other. The top layer contains core e-services like billing, auctioning and settlement.

SFS 2.0. Part I contains the following parts:

SFS concepts: The e-services vision envisages that the e-services are aggregated into eco-systems where the relationships between the e-services are formed dynamically. In such eco-systems, e-services communicate with each by asynchronous messages containing XML documents. The sequence of messages exchanged in order to perform one specific task forms a conversation. Services are characterized by two things: their description that is registered at a matchmaker, and the set of conversations they support that represents the set of interfaces they implement. The services are described using the notion of vocabularies that provide an extensible mechanism for defining the meta-data of e-services.

SFS messaging: SFS enables the interaction of services developed, hosted and provided by different service providers. SFS messaging defines the format and default protocol mapping of the messages exchanged by the services. It defines the standard structure of the messages and is based on XML/SOAP/MIME. The SFS tags that are common to all SFS messages, as well as the mapping to an example transport protocol, HTTP, is also specified.

CDL: The conversation description language CDL allows standard bodies and service providers to define the expected external behavior of services in a formal way. CDL is an XML based language that defines documents exchanged between services, and the requested order of these documents. It is the language used to define the business payload in the SFS messages and makes it possible for different service providers and implementers to provide compatible services. It also allows the creation of conversation libraries by vertical standard bodies that can be used by the service providers in that domain.

Transactions: In traditional distributed systems, the notion of transactions typically involves four properties: Atomicity, Consistency, Idempotence, and Durability (ACID properties). The tighter coupling between the various parts of the distributed system makes the problem of transactions solvable to some extent. In the world of e-services, however, guaranteeing all the four properties can be a challenge. In SFS, we first consider the problem of atomicity and propose two alternative approaches to solve the problem. These two approaches are based on compensation and two-phase commit protocols.

Service registration and discovery mechanisms: Service descriptions specify the various properties of a service offering. These properties contain e.g. the interfaces supported and the provider

of the service, but also many domain specific characteristics. The necessary terms for the domain specific part of the description are defined in vocabularies. Vocabularies and service descriptions play an important role in the conversations used for registering and finding services. In SFS 2.0 these are the following conversations:

- *Offer registration conversation*: this conversation is used by service providers to register their services or their need for services with a registry service.
- *Introspection conversations*: the two introspection conversations allow a potential service consumer to inquire a service for its capabilities.
- *Match-making conversation*: service consumers can ask registry services about potential services using the match-making conversation. This conversation matches the criteria given by service consumers with the service descriptions of available service offers.

Service monitoring:
In order for an eco-system to function, comprehensive management support is critical. This management functionality not only provides support for low-level operations such as starting, stopping, and monitoring the state of e-services, but also provides support for high-level operations such as generating audit trails, contract compliance, etc.

SFS 2.0 part II containts the following parts:

Negotiation Framework
The negotiation framework provides building blocks for expressing various negotiation protocols that occur in practice. This framework captures not only 2 party negotiations, but also multi party negotiations such as auctions, exchanges, etc. Note that the negotiation framework provides support for various negotiation protocols, but not the negotiation strategy used in the negotiations.

Contract Framework
The contract framework establishes the conventions necessary for forming electronic contracts amongst e-services. It builds on the matckmaking and negotiation frameworks for this purpose.

SFS 2.0 part III is not available yet.

1.4 Related work

We now briefly outline other efforts in the industry that have a similar scope.

UDDI:

UDDI specifies common API's for service registries, and specifies how UDDI registries have to be operated. These registries contain information about the service providers and the services they provide. In addition, these registries also provide models for enterprises and specifies how the UDDI registries synchronize the data they contain. There is significant overlap between the functionality in UDDI and SFS as far as the notion of registering services, searching for services, etc.

Eventually, since parts of SFS are being submitted to UDDI, and support for UDDI APIs will be added to SFS, SFS will be fully compatible with UDDI.

RosettaNet:

RosettaNet is a consortium that defines standards for e-business. The RosettaNet PIPs defines the business content of the messages exchanged between companies, i.e. the content of the documents and the order of the document. RosettaNet refers to this also as the public process. Currently most PIPs concern procurement. While the guidelines that specify how to describe PIPs can be compared to the Conversation Description Language CDL in SFS, the guidelines are not formal and are only used to specify those public processes standardized by RosettaNet. CDL can be used for specifying standardized conversations as well as any conversations some partners agree upon.

The RosettaNet Implementation Framework (RNIF) defines the format of the message exchanged between businesses, i.e. all the header tags needed in additional to the business payload, the content of acknowledge messages, the handling of security, and how the messages map to HTTP(S). This part of RosettaNet can be compared to SFS messaging.

RosettaNet addresses a more static e-business environment than SFS in that prior to having electronic exchange companies set up a business relationship and trading partner agreements. Therefore RosettaNet does not address dynamic elements like service registrations and service discovery.

ebXML:

ebXML is an OASIS/UN initiative to define all the layers in the web services architecture stack. These include registries, business process modeling, service descriptions, and transporting/packaging/messaging. The ebXML architecture is based on the Open-edi reference model, in that it supports a business operational view (BOV) to describe the relevant aspects of business transactions and a functional service view (FSV) to implement the business operational view using standard technologies such as XML, Java etc. More specifically, the BOV deals with operational conventions, agreements and mutual obligations of a trading partner, while the FSV addresses the functional capabilities, service interfaces, protocols, data transfer infrastructure, inter-operability among XML vocabularies of different businesses, discovery, deployment and runtime scenarios.

ebXML architecture specifies suitable registry service interfaces/wrappers to connect to UDDI. This establishes inter-operability among UDDI and ebXML service registries. ebXML's messaging service extends support for various transport protocols such as HTTP, SMTP, FTP etc, but it is not SOAP compatible. Hence, ebXML messages need to be explicitly cast/converted into SOAP, before sending it to a UDDI service. ebXML registries are distributed much like UDDI registries. UDDI does not mandate conformance requirements as to what constitutes a UDDI service, while ebXML specifies conformance to the complete ebXML technical specification as a pre-requisite for qualifying as an ebXML service (OASIS specified test suites are recommended by ebXML for conformance testing). ebXML architecture refers tpaML (trading partner agreement, invented by IBM) to enforce multi-party business process integration, while UDDI is not related to tpaML at this point.

BizTalk:

Biztalk is an industry initiative started by Microsoft with the goal of driving the rapid, consistent adoption of XML to enable e-commerce and application integration. The Biztalk platform comprises Biztalk.org (a library of XML schemas for various vertical industry business processes), the Biztalk Framework 2.0, a specification based on industry standards for data exchange in a reliable and secure manner over the Internet, and the Biztalk Server 2000, the engine that unites EAI, B2B and the Biztalk orchestration technology to allow companies to build dynamic business processes that span applications and organizational boundaries. From the web services stack architecture perspective, Biztalk specifies the technical conventions such as reliable messaging and routing (using SOAP/XML/S-MIME like technologies), and business process modeling/process orchestration using X-LANG, a workflow modeling language (designed by Microsoft).

2 SFS Concepts

The following diagram provides an overview of the main concepts of SFS and how they relate to each other. The diagram uses UML class diagram notation.

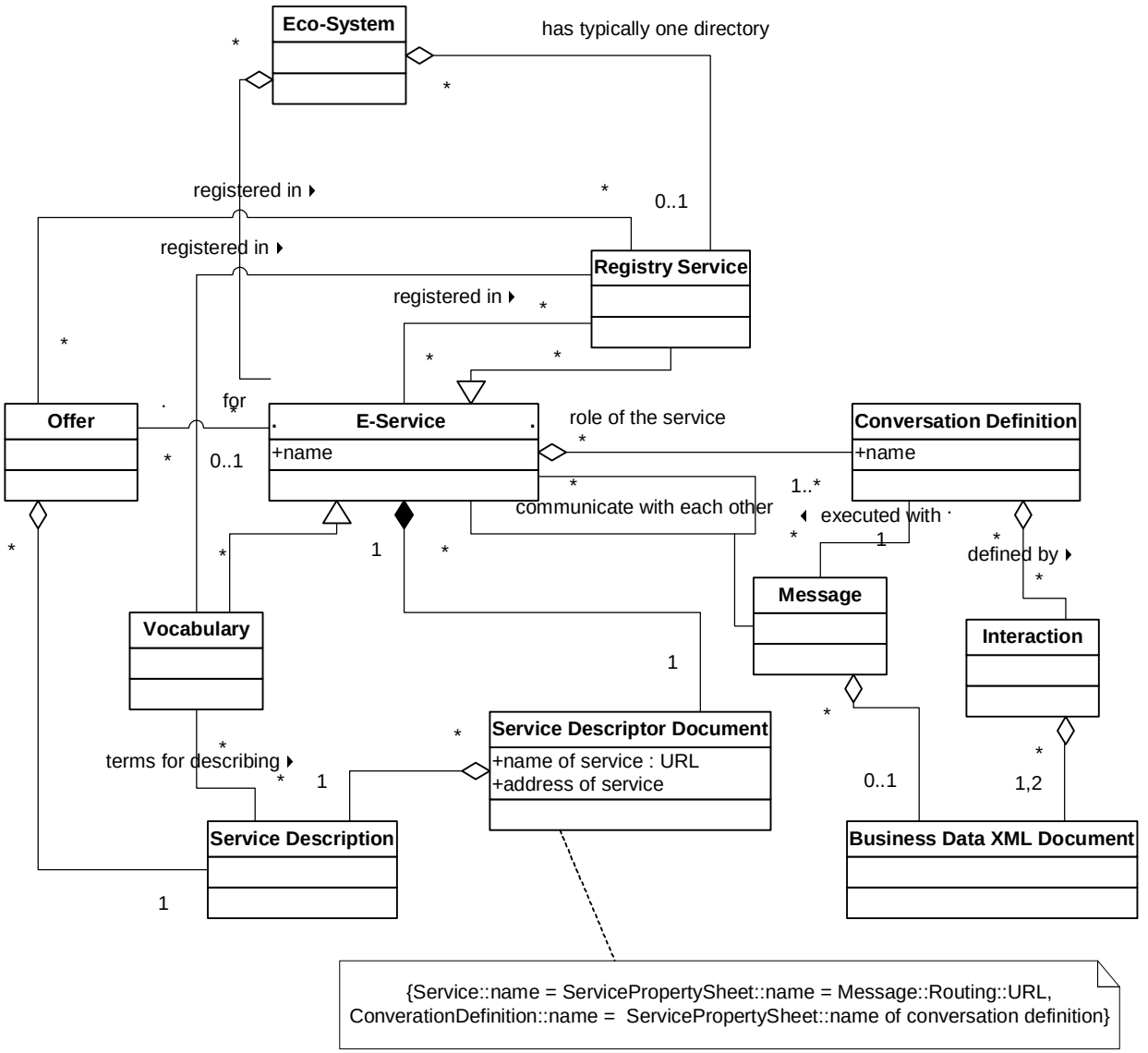


Figure 1: UML model of main SFS concepts

As shown in above figure, an eco-system consists of various e-services interacting with each other across organizational boundaries. Offers for using or providing these e-services get registered in a registry service, also often called a match-maker or an e-services village. These offers contain a description of all the relevant characteristics of the needed or provided service. The service registry matches queries by service users and providers with the registered offers (match making conversations). The registry service itself is a speciliced e-service. Each e-service also knows about its capabilities and interfaces and can be introspected by using the introspection conversations.

This conversation returns a service descriptor that contains various predefined fields about the address, provider and conversations of the e-service, plus a service description done in any vocabulary chosen by the service provider and registered with a service registry. The e-services communicate with each by asynchronous messages containing XML documents. The content of the documents and the order their exchange is defined by interactions and their possible orders specified in conversation definitions.

2.1 E-services and Eco-systems

2.1.1 E-services

Definition

In general, **e-services**[1] are applications or components that can be accessed by other applications or components over the internet and across organizational boundaries. In the context of SFS, **e-services** have to fulfill additional conditions. These conditions are the following:

- E-services are addressable units of software (e.g. by URIs), and other services can send them messages over the internet using the address.
- E-services are optionally registered with a service registry (a special e-service). This allows e-services to dynamically find new e-services or changes to existing e-services, as defined by SFS.
- E-services support introspection, i.e. they must be able to provide information about themselves as defined by SFS.
- E-services communicate using the document exchange model and SFS messaging.

The only exceptions are applications that only use other e-services, but are never used by other e-services. Such applications are called end-user applications and are also considered to be e-services though they normally are not registered in a registry and cannot be introspected.

In SFS all applications that fulfill the above definition are called e-services, independent of whether they provide a service or use a service. The e-services can take on different roles in their communication. Depending on the kind of the communication, different terms for these roles are used. Examples are: initiator and listener, client and service or server, user and service, consumer and service, consumer and service provider. In all these cases, the clients, consumers, and initiators are e-services. The actual role taken on by e-services depends on the functionality of the e-services.

Interfaces and implementations

One can clearly distinguish between the interface and implementation of e-services. The interface (also called public process) defines the functionality visible to the external user and how this functionality is accessed. The implementation (also called private process) realizes this interface, and the implementation details are hidden from the users of the e-service. Polluting the conversa-

1. Often also called web-services.

tional interface with details of the implementation limits the reusability of the conversation definition. When properly seperated, the same interface can be implemented by different service providers using any programming language of their choice. One service implementation might provide all the functionality itself, whereas another service implementation might use other e-services to provide the same functionality.

The term e-service refers to both, the e-service interface (the functionality provided by the e-service) and the e-service implementation (the software component providing the functionality).

E-services and web-pages

E-services are different from simple web-pages. Web-pages may also offer access to applications across the internet and across organizational boundaries. However, web-pages are targeted at human users, whereas e-services are accessed by other applications. E-services can use HTTP as a transport protocol, but they can also use other transport protocols, and their address may or may not be an URI. E-services are about machine to machine communication, web-pages about human to machine conversations. There is an overlap where web-pages targeted at humans get also accessed by applications. Often these applications act as proxies and are special parsers able to find the relevant information in the pages and to present them to other applications. In fact, these proxy applications then are e-services with the business logic provided by web-pages.

Customers may access web-pages that in turn access other applications from other companies over the internet. Whereas SFS does not apply to the communication between the customer and the web-page, it applies to the communication between the web-page applications and the other applications. A typical scenario for this are service provided to mobile phone customers.

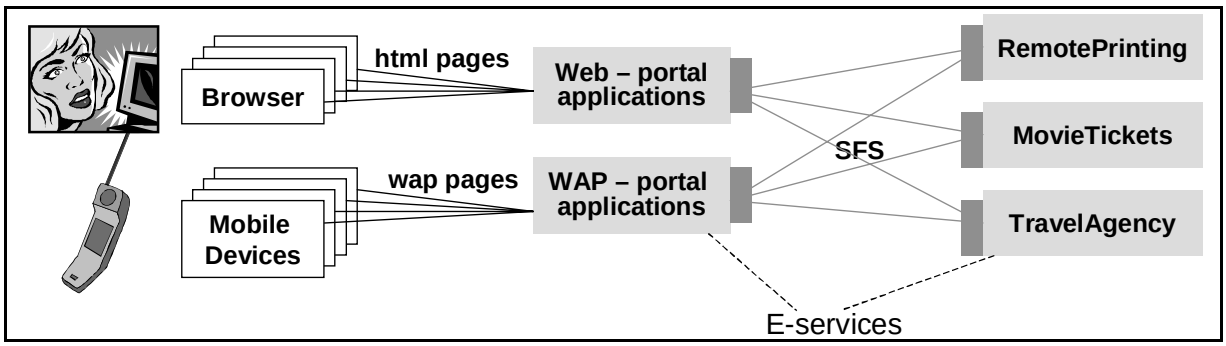


Figure 2: Portal services acting as e-services and providing user interface

Business logic in e-services

In most cases, e-services delegate the actual business logic to other applications. E-service might use other e-services hosted by other service providers in order to fulfill the required tasks or provide the requested information. Or they might use any other in-house applications or back-end systems in order to provide their tasks. Such applications might be applications implemented as EJBs on application servers, remote objects accessible over CORBA, or ERP systems like SAP.

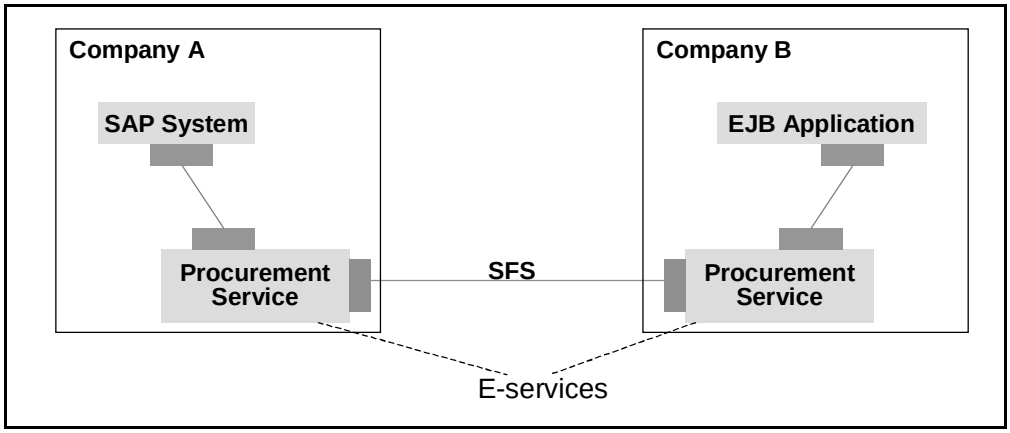


Figure 3: Existing business logic applications used by E-services

E-services may provide some standalone functionality, e.g. mathematical calculations or providing weather forecasts. Especially in B2B contexts, e-services are often used to enable business processes or workflows from two companies to interact. The applications controlling the overall business processes, e.g. workflows hosted by workflow engines or ERP systems, provide interfaces for company external access. These interfaces can be implemented as e-services.

2.1.2 Eco-systems

Definition

An *eco-system* is a collection of e-services that are inter-related because of the functionality they provide each other. In addition, the eco-system provides mechanisms for establishing the web of trust between the various e-services participating in the eco-system.

For example, board designers need information about various hardware components like IC's, microprocessors, resistances, they are putting together. After circuits have been designed, they need to be checked, and the components need to be ordered. An eco-system contains the applications used for designing boards, services that provide information about the characteristics of the components, services that know the availability of the components, services for ordering components, services for verifying designs, and finally services that route designs to prototype manufactureres. This eco-system specifies the interfaces of all involved e-services, and makes sure all the necessary support e-services like service registries and billing are in place. By allowing all these e-services to work together across organizational boundaries, the eco-system provides important functionalities to its various end-users like designers using the e-service enabled design applications or the seller of components.

Service providers and consumers

Service providers are entities that host e-services. For example, a bank may host an e-service that provides financial services and thus is the service provider of these e-services. The term *service consumer* if often used both for an organization using services of other organizations, and for the end-user applications that directly or indirectly use other e-services in the eco-system. Often, an

organization providing services also uses services, thus acting as both, a provider and a consumer of services.

Roles in an eco-system

E-services in an eco-system can often be classified along the role that they enact in the eco-system. These roles can either refer to the role played by an e-service in a conversation, or a higher-level role in an eco-system. Examples of roles in an eco-system are buyer, seller, match-maker, negotiator. An e-service can play several roles concurrently or subsequently. During a specific conversation all e-services play either one of the following roles during a specific conversation:

- listener (also referred to as server, and service)
- initiator (also referred to as client, and consumer)

Service registries

A *service registry* is a directory for e-services. It allows service providers to register services, and service consumers to find services. Service registries act as match-makers by introducing consumers searching for a service to providers capable of delivering the service. SFS compatible service registries support the registration and match-making services defined by SFS.

2.1.3 Market places

Eco-systems that provide B2B e-commerce functionality are often also called *market places*. Some e-service in a market place take on very specific roles. Many of the high-level conventions of SFS directly apply to this kind of eco-system.

A *market-maker* is an e-service that simplifies, or in some instances establishes, the web of trust among participants in an eco-system. It maintains contractual relationships with a large number of e-services and other market-makers. Its role is to establish the trustworthiness of a service, vouch for it in the marketplace, and take punitive actions if the service should violate its trust. The market-maker essentially acts as a broker between parties involved in economic transactions. It may assist negotiating parties to establish contracts with one another, and may actively monitor their exchanges to verify that the terms of agreement are being met. In times of dispute, the market-maker may assume the role of market-mediator, as it determines fault between disputing parties.

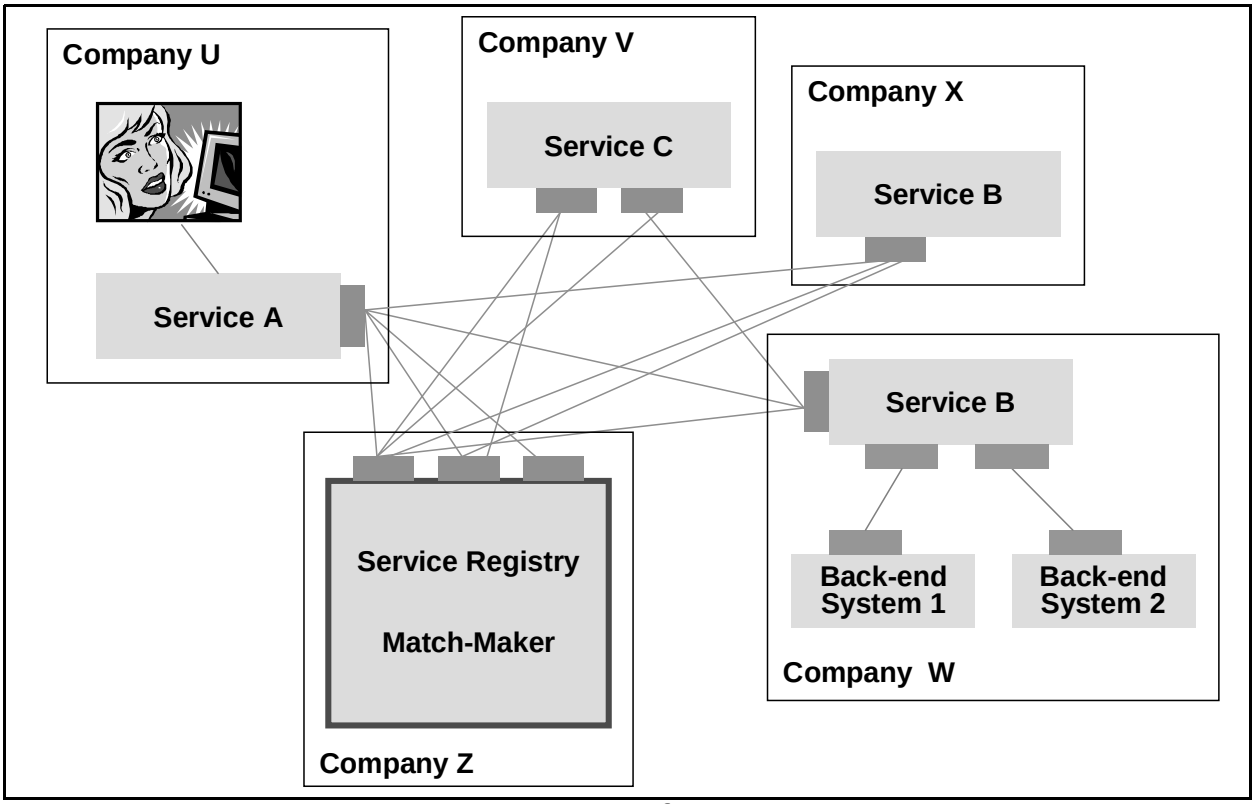


Figure 4: Example of a market place

The figure below shows an eco-system with a service registry that acts also as match maker. The various providers register offers for their services with the match-maker, and consumer either query these offers or place offers for consuming services. Latter offers can be queried by service providers. After a service has discovered another servive with which it wants to communicate, it interacts with that service directly.

In most market places there is one service taking on the role of a market-maker as well as of a service registry. The market-maker may even aggregate a set of services if one single service cannot deliver the desired features. A market-maker may also communicate with other market-makers to help create the best possible match for a service consumer.

2.2 Communication and service description concepts

2.2.1 Document exchange model

The e-services in an eco-system that adhere to the SFS standards use the document exchange model for communication. The business data to be exchanged is expressed in XML. The only exceptions are standard document files to be exchanged, e.g. MSWord or Excel files, which may be exchanged as binary attachments to XML messages.

The document exchange model has the following characteristics:

- **Clear separation between implementation and realization:** a client does not need to have any knowledge about the implementation. It does not create any remote object, and does not handle any system specific references of remote objects. The client simply sends the document to the address of the service, and it is up to the receiving software to find the right process, and if necessary, to start applications and create objects.
- **Extendable and flexible**: If a sender puts additional tags in document that the receiver does not know, or if the sender does not put in all tags that are required by the receiver, the receiver still can read the XML document, act on it, and if there is enough information, return the desired information. This allows upgrading of services without forcing all consumers to upgrade as well. It also allows clients to participate in the eco-system that for some reason cannot provide all the information required in the document schema.
- **Human readable**: If everything fails, XML documents can be read by humans and communication problems can thus be resolved.

In addition, a document exchange model based interaction operates on a larger level of granularity than an object model. Several small method calls of an object system are aggregated into one document exchange in the document exchange model. This granularity is more appropriate for the loosely coupled cross-organization communication. On the other hand, the tight coupling offered by traditional object models is more appropriate for distributed systems within an enterprise.

Documents are the basic unit of data exchanged in SFS. With documents we always refer to business documents, the data exchanged between the application logic. Which documents are exchanged between two services is defined in their conversations definitions using the conversation definition language CDL.

In SFS the content of the documents to be exchanged is defined by XML Schemas. Preferably these Schemas are defined for by standard bodies for specific problem domains, together with the semantics of each data element.

2.2.2 Messages, Conversations, Interactions

Messages

Messages in SFS are asynchronous one-way messages. Each message carries a payload containing an XML document, optional attachments, and some additional meta information about the message and the conversation also in XML. SFS messages can be exchanged over various different transport protocols, e.g. HTTP, ftp, email, ESIP and other open and proprietary messaging protocols that are capable of carrying XML documents. At this point in time, the messaging specification does not contain any mechanism for bundling multiple XML messages between two entities. It is assumed that the bundling is achieved by appending multiple XML documents within a single document in the payload.

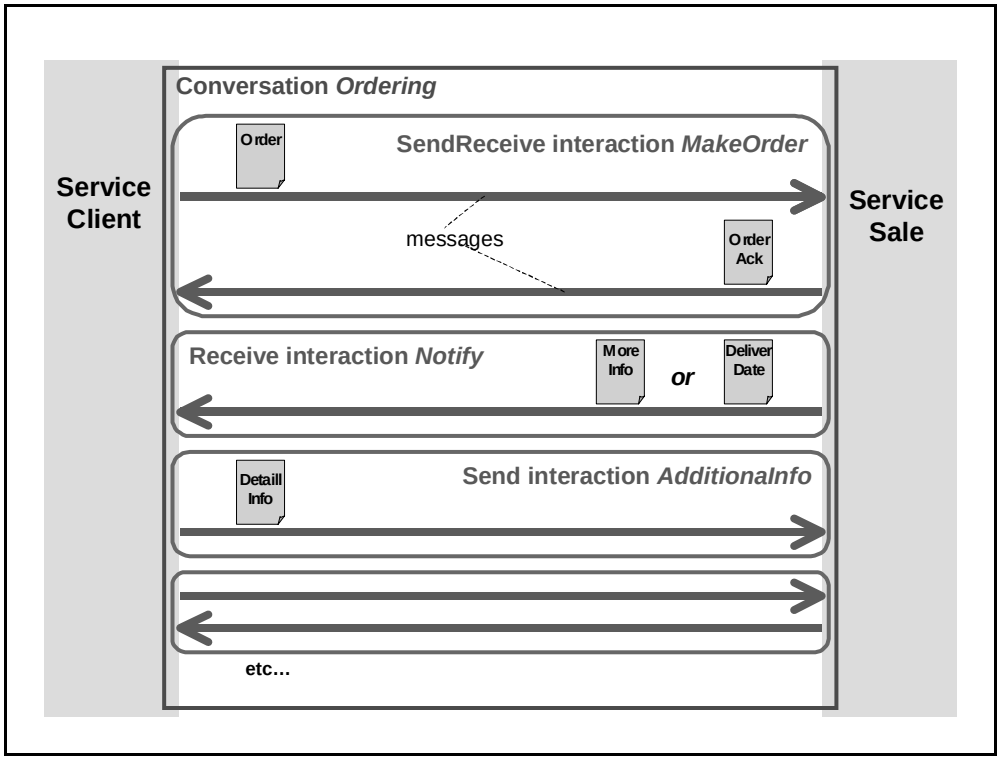


Figure 5: Conversations, Interactions, Messages, Documents

Above figure shows two services, which communicate with each other using the conversation *Ordering*. This conversation contains various interactions, the ones shown are the *MakeOrder* interaction that exchanges the documents *Order* and *OrderAck*, the *Notify* interaction that exchanges either the document *MoreInfo* or the document *DeliverDate*, and the *AdditionalInfo* interaction with the *DetailInfo* document. Additional messages without business payload are not shown.

Conversations

Conversations define not only the document definitions, but also the sequence in which they are exchanged in order to carry out specific business tasks. Though in the simplest case a conversation can consist of just one or two document exchanges, in general, they can be quite lengthy and contain an arbitrary number of document exchanges. For example, bodies such as RosettaNet define the conversation definitions for specific functions in the supply-chain industry. If for a specific problem domain no predefined conversations exist, they need to be defined by the participants of the eco-system.

Conversations are specified using the conversation definition language CDL. Each conversation defines a set of interactions between the participants needed to fulfil a specific purpose or task. It defines the documents exchanged in these interactions, and it defines the possible sequences of interactions. Conversations only define an interface of the service, not its implementation. In other words, they specify the public process, and not the private process.

Each service can support one or more conversations, taking on a specific role in each conversation (e.g. the role of the buyer, of the match-maker, or the client trying to find a service,...).

Interactions

Interactions are units of information exchange between participants in an eco-system, they are the basic building blocks of conversations in the conversation definitions using CDL. An interaction contains one or two documents exchanges, i.e. one or two messages with business payload.

In general, interactions can be classified along many dimensions:

Two-way or One-way interactions: Two-way interactions typically consist of a request and a response. The initiator of the two-way interaction views the interaction as a out-bound request message followed by the in-bound reply. The recipient of the interaction views the interaction as an inbound request followed by an outbound response. One-way interactions essentially are uni-directional interactions. They involve sending a message from the sender to the recipient. As in the case of two-way interactions the two end points of the interaction have different views on the interaction. The sender's view of the interaction is that an out-bound message is sent, whereas the receiver's view is that an in-bound message is received.

Two party or multi-party interactions: Most interactions take place between two parties. However, any interaction can occur among many parties. For instance, in order to model the participation of an e-service in an auction, one can make use of the multi-party interaction. Suppose, for instance, that the auction is an open-cry english auction, when a participating e-service places a bid, it has to reach all the participants. This broadcast of the bid by a participant to the other participants can be modeled as a single multi-party interaction[1].

Mediated Interactions: A mediated interaction is an interaction that occurs through an intermediary. The intermediation can occur at the transport level, as happens on the internet today, or at the application level. The mediated interactions in SFS are mediated at a level that is higher than the transport level mediation. For example, in a electronic hub where suppliers and consumers conduct business, mediation means that all the interactions between the suppliers and consumers occurs through the

Some of these benefits are:

- Virtualization: This allows e-services to be addressable with an extra level of indirection. This allows clients of the e-service to bind more loosely to the e-service. This allows, for instance, the mediating entity to replace the e-service with a separate e-service that has the same attributes when the original e-service goes off-line.
- Security: Mediation provides a nice mechanism to implement various security policies.
- Management: Mediation allows the interactions to be monitored and managed more easily.

1. Either each participant broadcasts to all the others, or the auction holder has to broadcast to all the aprticipants.

Some of the standard business level interactions such as negotiation, contract maintenance, etc. need the notion of mediated interactions.

Note that mediation can also occur right on the transport layer, this kind of mediation already occurs on the internet where web servers essentially mediate incoming requests to business logic. However, the mediated interactions that we discuss here are higher level mediation that the e-service can take advantage of. For instance, the mediator may expose APIs that allow any e-service to produce an audit trail of any mediated conversation that it has been a part of.

Secure interactions: The e-speak services framework enables e-services on the internet to dynamically interact with other e-services over the open internet using standard protocols. In order for this to be successful, the infrastructure has to be able to provide a comprehensive security infrastructure. Depending on the deployment, the intended use, etc., there may be different security requirements. Examples include:

- Authentication of parties involved in the interaction
- Encryption to ensure that messages carried in the interaction are not tampered with.
- Access control mechanisms that allow any party in an interaction

Each e-service should be able to define, for any interaction, what security parameters are required.

Transactional interactions: One of the most important properties that any interaction can have is a all-or-nothing semantics. This may be because a sequence of external interactions can comprise a single transaction in the implementation of an e-service. Either the interaction definition guarantees all-or nothing behavior, or it specifies compensating interactions that need to occur in order to negate the effect of the original interaction.

Disconnected interactions: The services framework provides support for disconnected interactions in order to support mobile customers. An interaction is said to be disconnected when one of the end points is not on-line when the interaction occurs. A mobile client may initiate a conversation, but may not be online when the response to its request is created. In order to support disconnected interactions, the framework needs to provide means for storing the state of the conversation and messages that are involved in a conversation. It should allow a client to resume a conversation from the state where it left off.

Though we classify these as dimensions, they may not be entirely independent of each other. For instance, if an interaction is mediated, it impacts the kinds of security that the interaction can leverage. Similarly, if the interaction is transactional, and is a two-party interaction, one of the parties can be the owner of the transaction. However, if the interaction is transactional and involves multiple parties, the protocol can be involved. In SFS 2.0, we only address one-way and two-way interactions and do not consider the other dimensions of interactions. The other issues will be included into later versions of the SFS.

2.2.3 Vocabularies and e-service descriptions

Vocabularies:

Each service in an eco-system is described by some metadata. In SFS, we introduce the notion of vocabularies for defining the metadata or terms that may be used in service descriptions and offer descriptions. In most cases, vocabularies are specified by vertical standards bodies for specific business domains, and a service provider reuses the existing vocabularies of its domain.

Service Descriptor Document:

Each SFS compliant e-service provides information about itself through the service descriptor document. It contains various information elements common to all e-services, and a description of the service in terms specific to the business domain. The common elements contain the address of the service, the list of supported conversations, required credentials, plus references to the service provider. The service description conforms to one or several selected vocabularies. In addition to appearing at matchmakers, service descriptions may be part of the service descriptor of a service.

Offers:

Before services can interact with each other they need to discover each other. SFS introduces the notion of offers that are registered with service registries. Offers are a generalization that can be specialized in many ways. Some of these are offers to sell, offers to buy, offer to provide, and offers to consume. Offers to sell are offers of service proviers for services that can be used and offers to buy are posted by potential service consumers. Both kind of offers are queried by interested services. Offers can refer to the capabilities of a service in general, e.g., describing a service that can perform specific computational tasks, or to goods traded by the services, e.g., available or needed items. Offers contain a description that uses the terms defined in a specific vocabulary to describe the service offering.

Identifying services and conversations:

The names of services and conversation definitions need to be unique within the scope of the registry in which they are registered. In addition, conversations themselves may be registered at matchmakers.

2.3 Communicating E-services: SFS Messaging Protocol Stack

The ultimate goal of communication between e-services is to exchange business payload, i.e. to exchange XML documents according to the specification of the conversation that is being carried out. In order to exchange this payload, lower level protocols are used for transport and quality of communication, and additional information has to be exchanged about the messages, services sending and receiving the messages, and the conversation being carried.

SFS clearly separates out the various layers of the protocol stack. It also makes use of existing and emerging standards for e-services communication as far as possible, adding value only where necessary to support the more advanced features provided by SFS for e-services interactions, e.g., the

concept of conversations. In the following we have a closer look at the various layers of the SFS protocol stack:

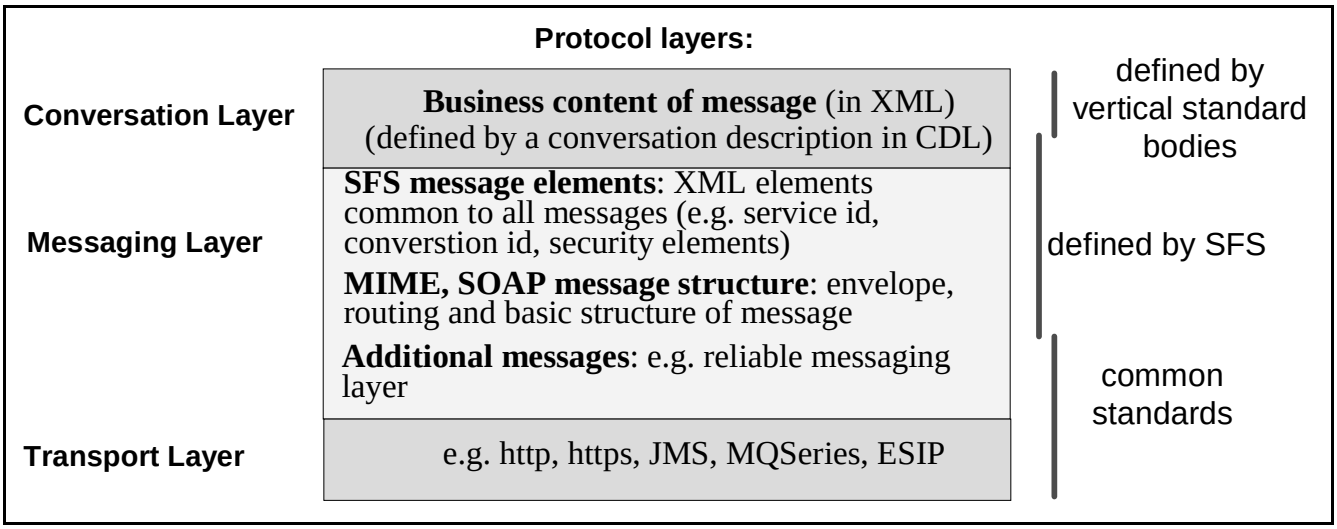


Figure 6: Figure: Protocol Stack of SFS Messaging

The **transport layer** simply provides transport of the message from the sender to the receiver, if necessary through firewalls. If transport is done over HTTP, the HTTP header and any information stored in the URI such as session ids are part of this layer. The transport layer is not part of the SFS specification, SFS messages can be exchanged over any transport layer. SFS assumes a clear decoupling between transport and messaging layer.

The **messaging layer** can be split up into three parts:

- The SFS messaging specification is concerned with providing additional information about the message that is needed by the receiver, e.g. address of the sender, intended receiver service, conversation id. These fields are defined by **SFS** and are needed by the receiver either for constructing any return messages, or for finding the right service and the right conversation instance, once a conversation has been started, this message needs to be dispatched to.

- The SFS messaging specification also defines the basic structure of the message, which is based on **SOAP** and **MIME**. Essentially, the message is a MIME multipart message, where the first part is a SOAP message with SOAP envelope, header, and body. The SFS messaging elements are placed into the SOAP header, the business content of the message is in the SOAP body.

- Depending on the transport protocol, there might be the need for an additional layer for **reliable messaging**. ESIP and MQSeries already provide reliable transport, whereas HTTP and HTTPS do not. Therefore, they require an additional layer between transport and SFS messaging to provide reliable messaging. Reliable messaging may add additional SFS messages that may either serve for acknowledgement or

repeatition. Reliable messaging is not yet defined in SFS 2.0, it is currently achieved by having a reliable transport layer. It will be addressed in future extensions of SFS.

The **conversation layer** processes the actual payload of the information exchanged between the services. This is the information in the messages that is actually handled and created by the business logic. The content of the payload is defined in the conversation definitions. The content of the payload is either defined by SFS predefined conversations such as match making, or defined by standards bodies of eco-system participants using the conversation definition language CDL.

2.4 Typical eco-system use-cases

In this sub-section, we briefly consider some of the typical operations in an eco-system. The typical interaction cycle in eco-systems consists of the following steps.

1. Identify potential e-services: This involves both registration of offers and lookup of offers at matchmakers
2. Negotiate terms and conditions of e-service conversation: This involves using the negotiation protocols to identify the terms and conditions of the conversation.
3. Formalize a contract based on the negotiations.
4. Execute the conversation.
5. Monitor the execution of the conversation in order to determine compliance with contract.

We now consider brief use cases that analyse each step in turn.

Offer registration:

Services can make themselves available to other services by registering offers with a service registry service. These offers can be offers to provide a service or offers to consume services. The registry service can be hosted e.g. by a match-maker like in the figure below (**1**). For registering offers services use the predefined registering conversations supported by any SFS compliant service registry.

Service discovery:

There are various aspects of a service that can be discovered. These include:
1. the conversations realized by a service,
2. a specific service hosted by a service provider and its various characteristics,
3. the address of the service.

These various aspects can be discovered either at development time or at run-time. The developer may just use the conversation definitions for developing its application that uses other services, or he may also cache the reference to his preferred service that implements conversations. Using the browser interface of a service registry, a developer can find suitable conversations and services (**2**). At run-time the client application can use the cached reference, or it can dynamically discover services that realize the chosen conversation using the match-making conversation with a service registry (**3**). A client application can also discover dynamically new or updated conversations by introspecting services it is either already using or that it has discovered by the match-making con-

versation. Conversation definitions and other service properties are requested from a service by using the introspection conversation (**4**).

Contract negotiation:

Once services have discovered an offer by another service, they may then negotiate with each other to form a contract. The contract is an XML document that specifies the precise rules of engagement. It is worth noting that the e-speak infrastructure itself is not responsible for generating negotiation offers and counter-offers. The infrastructure functions as a mechanism for transporting these offers, and may be responsible for maintaining the state of the transactions so that the parties negotiating may operate in a disconnected manner. Essentially, the negotiation framework in SFS provides conversation definitions that define the various negotiation protocols that can occur in typical eco-systems. These negotiation protocols encompass not only two-party negotiation protocols, but also multi-party negotiation protocols such as auctions, double auctions, etc.

Service interaction:

Services interact with each other by sending SFS messages to each other that contain the XML documents specified in the conversation definition. Normally, the communication between the services is a peer-to-peer communication that does not involve any other services (**5**). However, SFS can accomodate the notion of mediated interactions. A mediated interaction is an interaction that occurs through an intermediary. The intermediation can occur at the transport level, as happens on the internet today, or at the application level. The mediated interactions that we discuss here, are mediated at a level that is higher than the transport level mediation. For example, in a electronic hub where suppliers and consumers conduct business, mediation means that all the interactions between the suppliers and consumers occurs through the hub. This allows the hub to act as the marketmaker ensuring that the participants comply with the contracts that they have entered into.

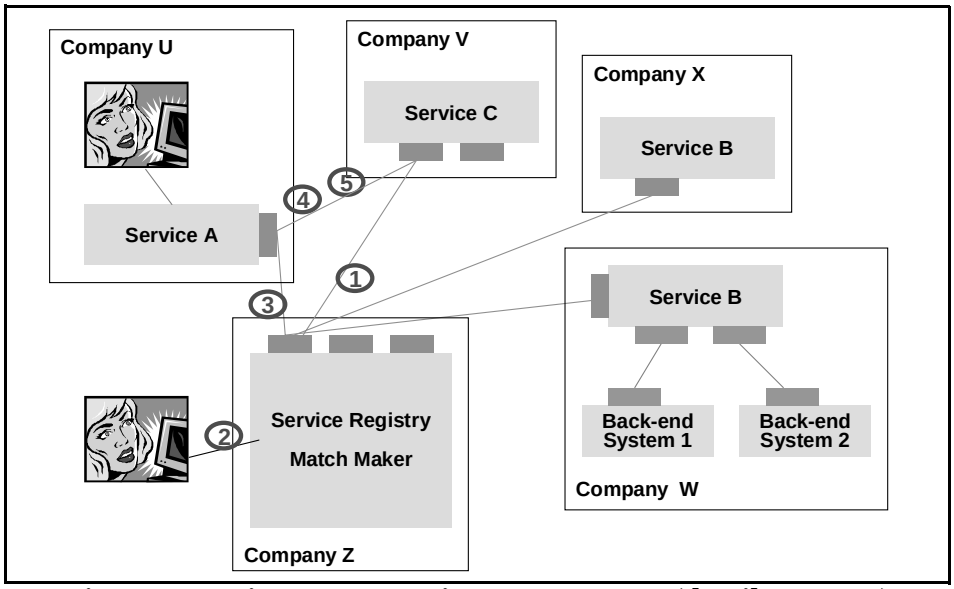


Figure 7: Various use cases in an eco-system (details see text)

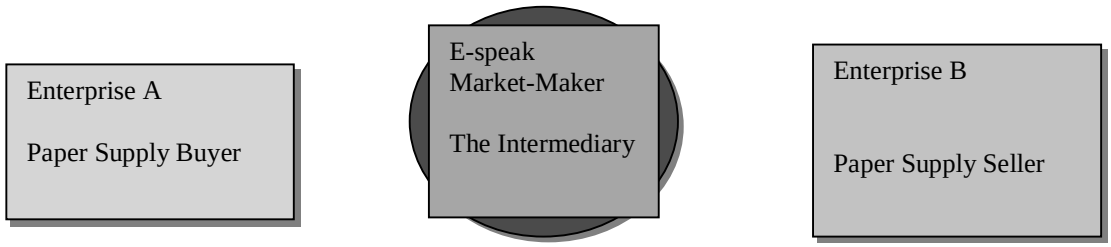
2.5 End-To-End Example: Description

To provide a concrete example of how the various components of the SFS fit together, this section introduces a complete end-to-end scenario describing the sale, dynamic discovery, and purchase of paper supplies. Note however, that this section is intended to be a high-level description only. It describes the parties involved and the roles they play, but does not provide the code necessary for each XML message. For a complete listing of XML code related to this example, please refer to the appendix of this document.

Scenario:

In this example, consider the following three parties: Enterprise A (the buyer), Enterprise B (the seller), and an e-speak Market-Maker (the intermediary). Assume that Enterprise A seeks to purchase several units of 8 1/2" x 11" acme copy plus paper. Also assume that Enterprise B sells that particular type of paper, but has no prior knowledge of Enterprise A. Finally, assume that the Market-Maker exists to serve as the central point of contact between buyers and sellers that want to be dynamically matched with one another on the Internet.

Note that it is not necessary for a market-maker to be present; any entity that provides matchmaking functionality can play this role, including the matchmaking and advertising service on the E-services Village.



(Or E-services Village)
Figure 9: Enterprise A, Enterprise B, and a Market-Maker

The following 10 steps describe a complete end-to-end scenario involving Enterprise A, Enterprise B, and a Market-Maker.

Step 1: Enterprise A Introspects the Market-Maker

Before interacting with a Market-Maker found on the Internet, an enterprise must first discover the following two pieces of information: *what* the market-maker is, and *how* it can communicate with it. The process of discovering this information is known as *service introspection*- the details of which are completely defined in the service description section of this document.

To briefly summarize how introspection relates to this end-to-end example, Enterprise A queries the market-maker by delivering introspection request messages, and the market-maker replies with introspection response messages that contain the requested information. At the end of a successful introspection, Enterprise A will understand how to communicate with the market-maker.

Alternatively, Enterprise A may talk to a well known site such as E-servicesVillage's advertising service through DSML to extract the property descriptor that would typically be returned in an Introspection Response Message.

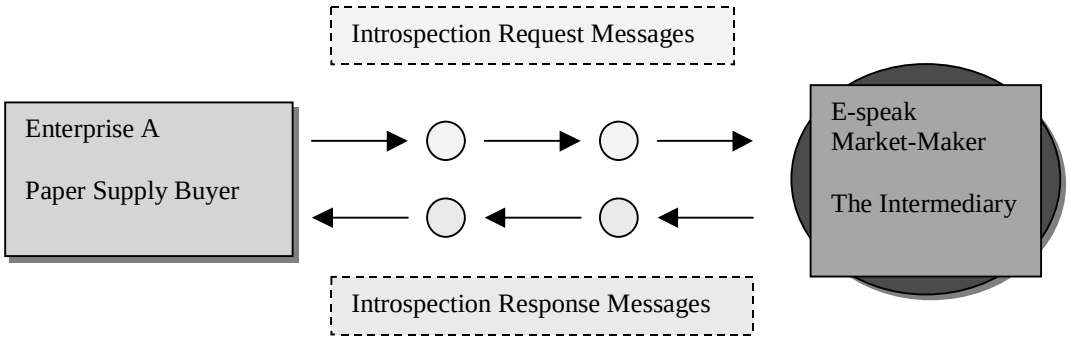


Figure 10: Enterprise A Introspecting an E-speak Market-Maker

Step 2: RFQ Generation and Registration

After successfully introspecting the market-maker, Enterprise A initiates its own internal process of creating a Request For Quotes (RFQ) for paper supplies. An RFQ is a document that describes the properties of the product that a given client seeks to purchase. In this example, Enterprise A generates an RFQ for 8 1/2" x 11" acme copy plus paper costing less than $22.95 per unit. This RFQ also contains contact information for Enterprise A, so that potential suppliers will know who to contact should they discover and respond to this RFQ in the future.

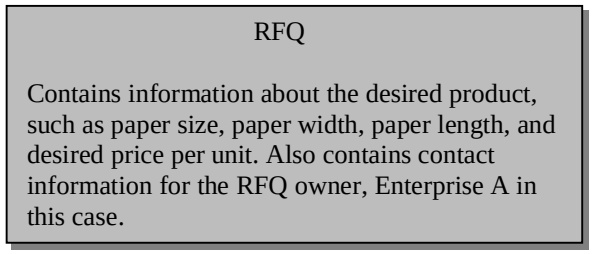


Figure 8: Simple Diagram of an RFQ for Paper Supplies

Next, Enterprise A registers this newly generated RFQ with the Market-Maker. The Market-Maker responds with an acknowledgement indicating the status of the attempted registration- "success" in this case.

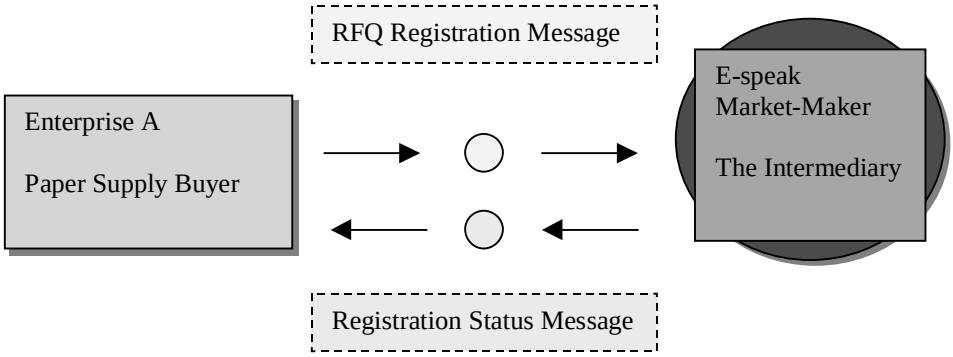


Figure 9: RFQ Registration and Status Response

Note that other clients seeking to buy paper supplies might submit similar RFQ's to the market-maker as well. A single e-speak market-maker is capable of handling requests from multiple clients.

Step 3: Enterprise B Introspects the Market-Maker

Identical to that of step 1, with the exception that Enterprise B is now introspecting the market-maker. This step requires no additional illustration.

Step 4: RFQ Lookup Request

Having successfully introspected the market-maker, Enterprise B now sends a lookup request for all paper supply RFQ's registered with the market-maker that offer to pay more than $20.00 per unit. Since Enterprise B is in the business of selling paper supplies, retrieving the Market-Maker's list of paper supply RFQ's is an effective way to obtain a new, potentially large list of clients. It is worth noting that this lookup request contains contact information for its owner, Enterprise B.

In response, the Market-Maker delivers a list of paper supply RFQ's that match the criteria of the lookup request. Since Enterprise A is willing to pay up to $22.95 per unit of paper, its RFQ is considered a match, and is included in the Market-Maker's response.

Note once again that the market-maker is capable of receiving requests from multiple clients. Therefore, it is likely that new suppliers will send similar lookup requests for paper supply RFQ's as well.

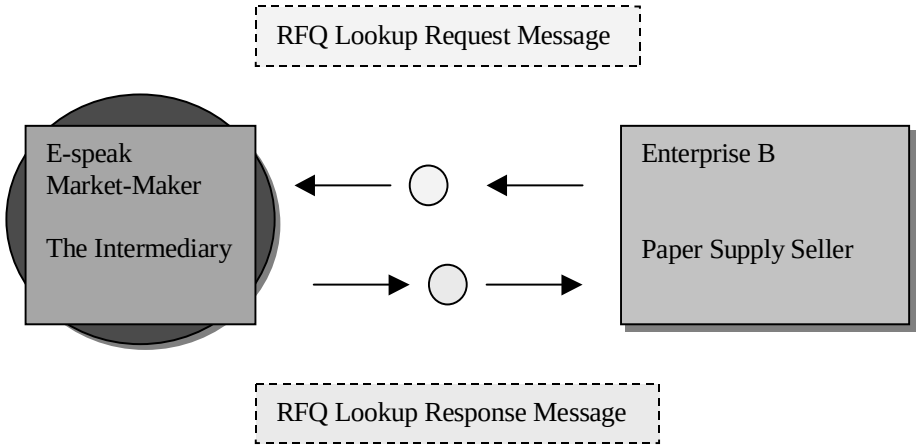


Figure 10: RFQ Lookup Request and Response Messages

After receiving the response from the Market-Maker, Enterprise B now has direct contact information for many potential buyers, including that of Enterprise A.

Step 5: Enterprise B Introspects Enterprise A

Since the Market-Maker has provided Enterprise B with a list of potential buyers, Enterprise B is now free to initiate direct communication with each buyer on the list. Because each buyer has been dynamically discovered, Enterprise B must first perform a generic introspection to determine how the buyer expects its quotes. In this scenario, Enterprise B directly introspects Enterprise A.

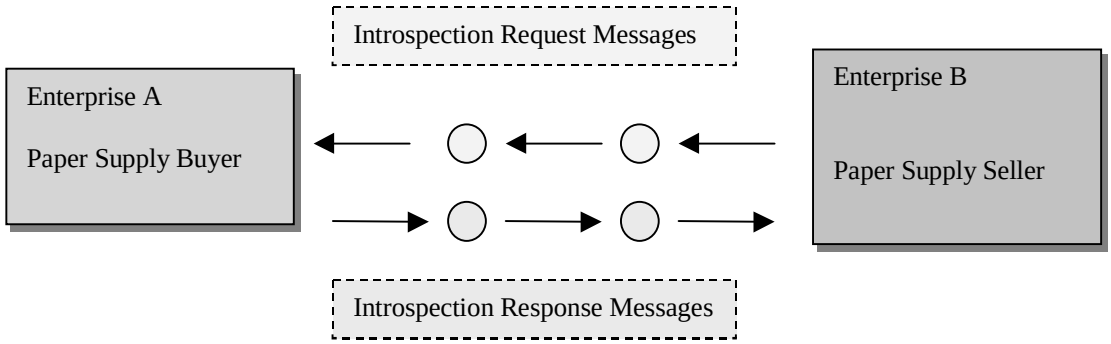


Figure 11: Enterprise B Introspecting Enterprise A

Note that this introspection is only required when the two enterprises directly interact for the first time.

Step 6: Enterprise B Sends Quote to Enterprise A

Having successfully introspected Enterprise A, Enterprise B now executes its own internal business processes for generating quotes in response to Enterprise A's RFQ. It delivers a quote to

Enterprise A. In turn, Enterprise A responds with a message acknowledging that it has received the quote.

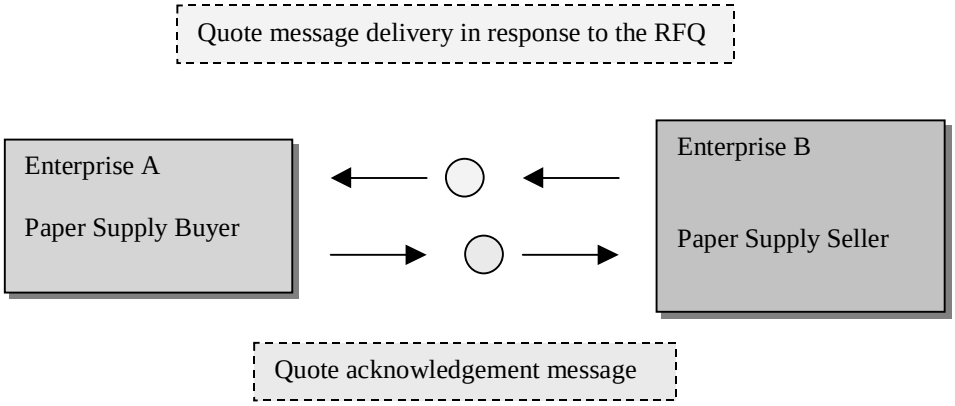


Figure 12: Enterprise B Sending a Quote in Response to Enterprise A's RFQ for Paper Supplies

Because other suppliers may have obtained Enterprise A's RFQ from the Market-Maker as well, Enterprise A could potentially receive quotes from many different sellers. In such a situation, Enterprise A would send a quote acknowledgement message to every enterprise that had issued it a quote. Additionally, if the suppliers were previously unknown, Enterprise A may choose to enlist the assistance of a third party business rating agency to determine if a particular enterprise is a reputable dealer.

Step 7: Purchase Order Negotiation

At this point, Enterprise A and Enterprise B may negotiate directly with one another until they reach agreement or disagreement on the contents of a purchase order. Negotiation itself comes in several varieties, but always consists of negotiating parties exchanging a series of offers/counter-offers until they either reach agreement or declare negotiations failed.

Each offer in the following diagram contains a partial purchase order with property values that become modified by the negotiating parties during each exchange. Enterprise A initiates the negotiation process by sending a negotiation offer, offering to negotiate the contents of a purchase order. The internal business process of enterprise A generates the purchase order template that is sent to Enterprise B.

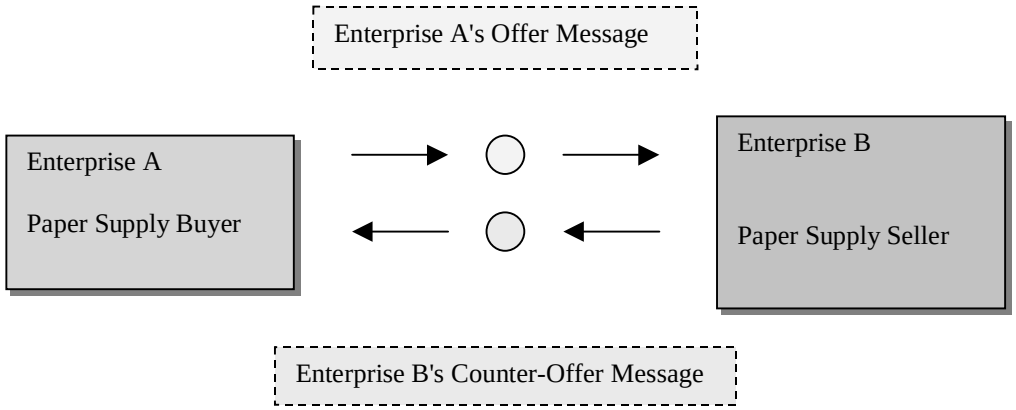


Figure 13: Negotiation Between Enterprise A and Enterprise B

Note: For a comprehensive discussion of negotiations, consult chapter 9 in part II of this document.

Step 8: Contract Formation

Assuming that Enterprise A and Enterprise B have successfully reached agreement on the contents of the purchase order in step 7 above, the two parties are now ready to form a contract. A contract is an XML document that specifies the goods and services to be exchanged, along with the rules that both parties must follow. Contracts may also codify the sequence of actions that must be performed in order to complete the transaction between the enterprises. For instance, the contract could stipulate that payment for the purchase order must be made within 30 days of receiving the invoice.

Having previously reached agreement on all negotiable attributes of the purchase order, the two enterprises now draw up a contract, which they send to each other.

Figure 14: Enterprise A and Enterprise B Reach Agreement and Exchange Contracts With Each Other

Alternatively, the Market-Maker could form and maintain the contract on behalf of the two enterprises.

Step 9: Purchase Order Generation and Shipping Documentation Delivery

Next, Enterprise A places its order for paper supplies (which conforms to the contract) with Enterprise B. Enterprise B returns a message acknowledging that it accepts the order "as is".

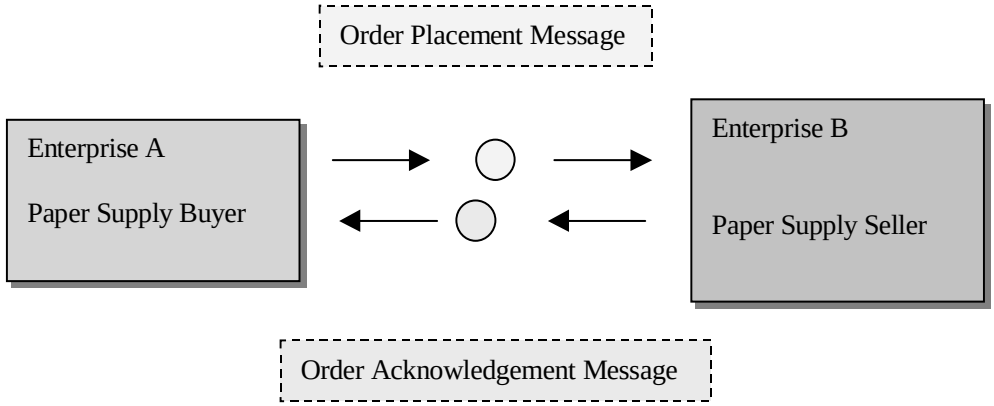


Figure 15: Purchase Order Placement and Acknowledgement

Enterprise B next creates a sales order in its enterprise system and begins the order fulfillment process. When it executes a goods issue, Enterprise B creates the necessary shipping documents and sends them to Enterprise A. In return, Enterprise A sends an acknowledgement message to Enterprise B.

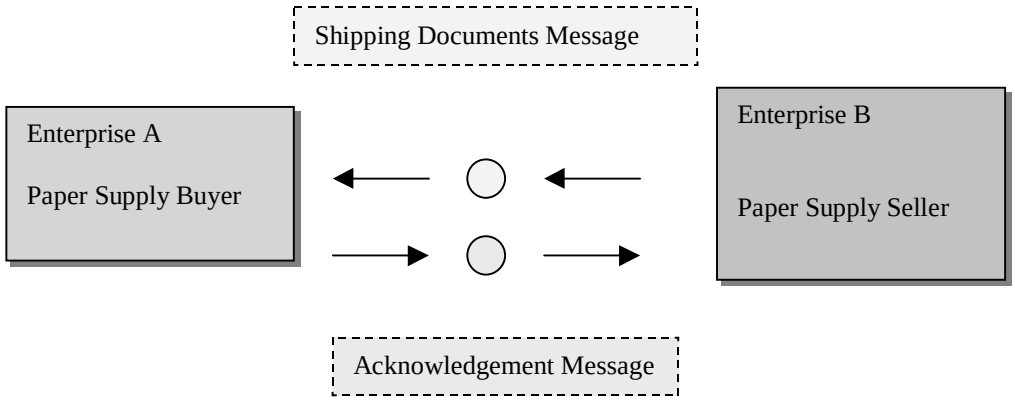


Figure 16: Shipping Document Delivery and Acknowledgement

Step 10: Invoice, Payment, Receipt, and Product Delivery

Enterprises A and B execute the remaining steps to complete the sale:

- Enterprise B sends Enterprise A an invoice.
- Enterprise A sends Enterprise B an acknowledgement response.

- Enterprise A sends Enterprise B payment for the invoice.
- Enterprise B sends Enterprise A the payment receipt.

And finally,

- Enterprise B ships the product to Enterprise A.

Alternate Scenario:

In some situations, it may be desirable for one service to be managed by another- a functionality that is made possible within E-speak through the XML Application Response Measurment (XAM) specification defined in chapter 8. The following illustration shows an alternate scenario in which Enterprise B functions as a managed service that communicates with a management service housed by Enterprise C. For a complete illustration of the types of messages passed between an application and its measurement agent, refer to section on management of e-services (chapter 8).

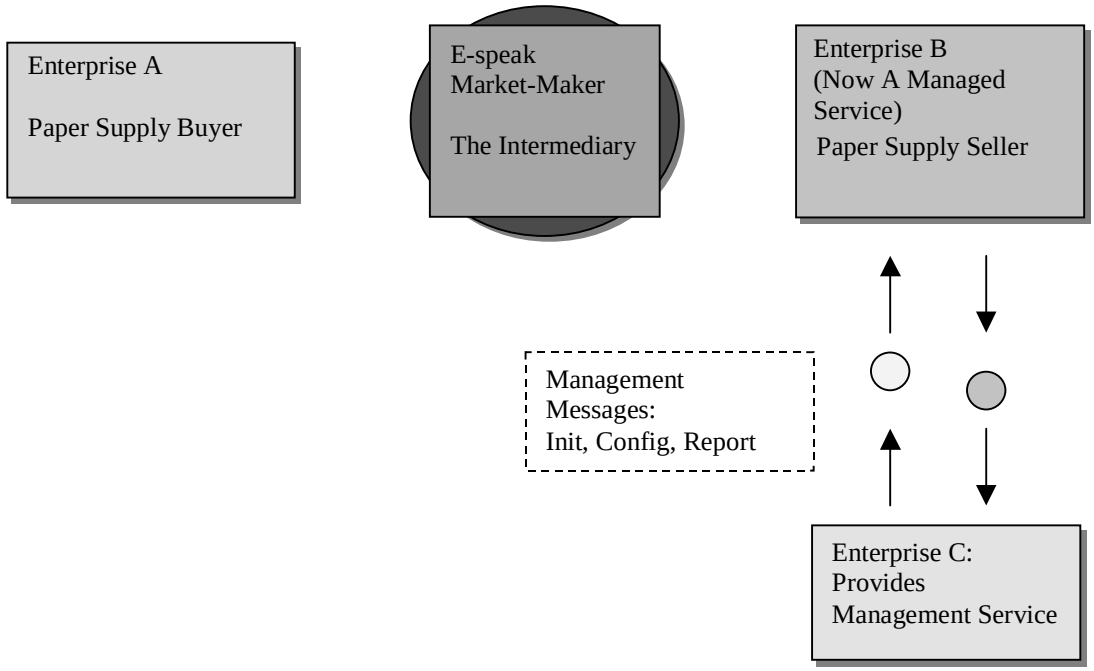


Figure 17: Alternate Scenario With Enterprise C Managing Enterprise B

3 SFS Messaging

In order for services to interact with each other, they need to define the exact protocol and message format they are using. The Service Framework Specification defines such a format, based on existing and emerging standards, and taking into account the specific requirements of loosely coupled services carrying out long-lived business conversations.

3.1 SFS Messages

The messaging layer is responsible for maintaining a communication channel between two service instances. This communication channel carries back and forth all the business logic payload belonging to one specific conversation instance. Furthermore, the messaging layer has to add all the meta information necessary for dispatching the business logic payload to the correct conversation controller and business logic instance. In SFS the message format chosen for carrying the business logic payload is based and SOAP 1.1 and MIME.

3.1.1 The SFS message structure

SFS messages have to carry business logic payload plus meta information about the message like routing, property and manifest information.

The business logic payload can be split up into two parts:

- **XML documents**: The CDL conversation definition defines when which XML document has to be exchanged and provides a pointer to the XML schema for the document.
- **Binary attachments** (e.g. EXCEL spreadsheets): These also have to be specified by in the CDL conversation definition. Currently this has to be done in the XML document schema, as CDL itself only specifies XML documents as payload.

The meta information about the message, i.e. routing, property and manifest information is carried in the SOAP header. The XML document payload is part of the SOAP body. SOAP header and SOAP body, wrapped into the SOAP envelope, are the first part of a multi-part MIME structure. Binary attachments are additional parts of the multi-part MIME structure. The following figure shows the basic SFS message structure:

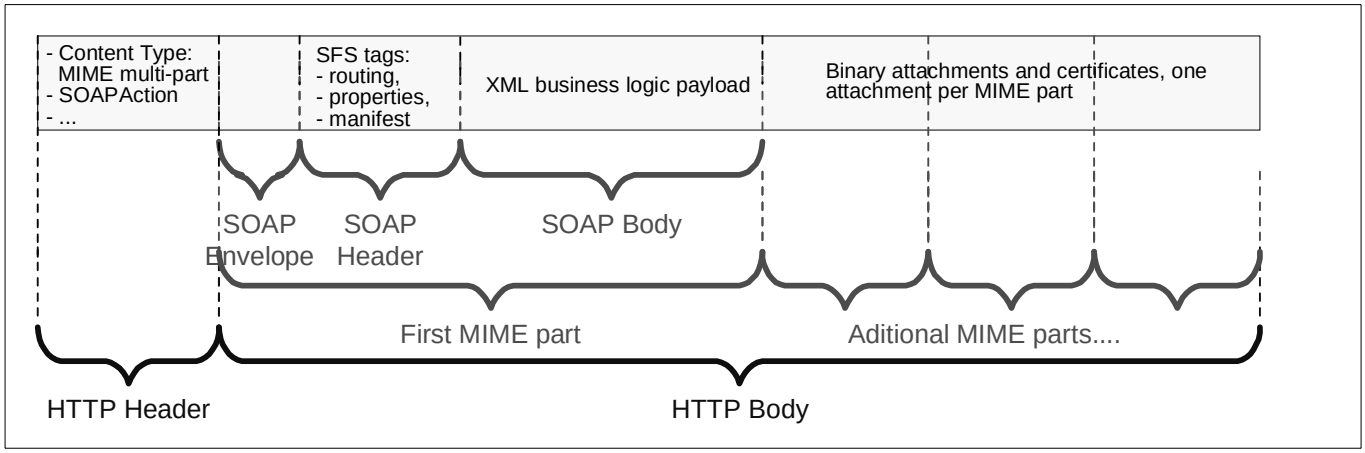


Figure 1: Figure: SOAP/MIME structure of an SFS message

The following example shows the structure of an SFS message when mapped to the HTTP transport protocol:

```
POST /StockQuote HTTP/1.1
             Host: www.ordering.com
             Content-Length: nnnn
             MIME-Version: 1.0
Content-Type: Multipart/Related;
             boundary="-----------1234567890";
             type=text/xml;
SOAPAction: ""
-----------1234567890
Content-Type:text/xml;
Content-Transfer-Encoding: 8bit
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope encodingStyle="http://schemas.xmlsoap.org/soap/encoding"
          xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope”
          xmlns:TRANSP=”http://schemas.e-speak.net/transport”>
   <SOAP-ENV:Header>

             <!—Here go the various SOAP header tags defined by SFS. ->

   </SOAP-ENV:Header>
   <SOAP-ENV:Body>

             <!—Here goes the actual XML payload as defined by CDL. ->

   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
-----------1234567890
```

3.1.2 SFS Message tags: Overview

In this section we describe the SFS XML tags that provide meta information about a message. They are part of each SFS message and are put into the SOAP header.

Routing: Conversations, Senders and Receivers

The Route element contains information about the sender, receiver and their conversation.

From the messaging point of view, **conversations** are stateful business communication channels whose lifetime is independent of the underlying communication and transport framework. The lifetime is solely controlled by the service as specified in the conversation definition. Conversations may last for a long time (e.g. several days), therefore several transport sessions may be needed to carry out a conversation. Furthermore, the participants in a conversation must be able to keep the state of the conversation over a long time (e.g. persistent memory), and to continue old conversations with new transport sessions.

The SFS message contains the name of the conversation as well as the id of the conversation instance. One specific service may carry out the same conversation with different partners at the same time, having several concurrent conversation instances of the same conversation type. Therefore conversation instances must be identified uniquely for both parties in a conversation.

The routing element also contains the name in form of the **URI** of both, the sender service and the receiver service.

Properties

The SOAP header may have a Properties element that contains further information about the message. SFS specifies the following subelements of Properties, which are all optional:

- **MessageID**: This ID is provided by the sender, and can be used by the receiver to make sure it does not process duplicate messages (e.g. due to resends after time-outs).
- **SentAt**: Time the message has been sent.
- **ReceivedAt**: Time the message has been received.
- **ExpiresAt**: Time when this message is no longer relevant and can be directly discarded upon receipt by the receiving server, i.e. the business logic would never see this message.

Binary data, attachments

If there are any attachments with the message (e.g. binary documents), the attachments are carried additional part of the MIME structure. The SOAP header may contain a **manifest** element that lists and describes the various documents attached. The manifest is a messaging layer flag and helps the messaging layer software in the receiver to correctly handle attached documents and to hand them over to the service or conversation controller layer software as needed. The SOAP body itself may also contain references to the attached documents (href attribute of SOAP body elements, the href reference do not reference any other elements, they reference the MIME ID of the attachment).

An example of a message with a manifest and attachment is in the example section of the SFS Messaging chapter.

3.1.3 SFS Message tags: Details

All the XML tags specified here are part of the SOAP header of an SFS message.

Routing tags:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
Route	Yes	Once	Contains the routing information in the SFS message	To, From
To	Yes	Once	Uniquely identifies the receiver service and the current conversation instance this message is part of	URI, ConversationID, ConversationName
From	Yes	Once	Uniquely identifies the sender service and the current conversation instance this message is part of	URI, ConversationID, ConversationName
URI	Yes	Once in To, once in From	Name of the receiving or sending service. This is the ServiceName field of the service descriptor.	String containing URI of sender or receiver service
Conversation-Name	Yes	Once in To, once in From	Name of the conversation. This is the conversation name in the service descriptor and conversation definition of the service that has the listener role in this conversation.	String containing the name.
ConversationID	Yes	Once in To, once in From	Unique ID of the conversation instance.	String containing the ID.

Attributes of routing elements:

Element	*Attribute*	*Required*	*Occurs*	*Semantics*	*Value*
Route	mustUnderstand	Yes	Once	Required by SOAP standard because routing element is mandatory and needs to be handled by receiver of message.	“1”

ConversationName: The name of the conversation is needed in the first message of a conversation in order to notify the other party of the conversation which type of conversation is being

started. As soon as both sides have established a new conversation runtime instance and assigned a unique ID to this instance, the conversation name becomes redundant.

The conversation name in the TO element of the first message of a conversation has to correspond to the conversation name published in the service descriptor of the receiving service. The conversation name in the FROM element is the name used by the sender for this conversation type.

ConversationID: This is a unique identifier chosen by each party at the beginning of the message exchange. The identifier has to uniquely identify the conversation instance of the sender (in the the FROM element) and of the receiver (in the TO element). In order to be unique for both parties, ID is made up of two parts: both parties in the conversation provide an id that is unique for them, forming an overall unique id. These id's are sent back and forth and stay the same for the whole life-time of the conversation. Upon receiving a message, the receiver simply copies the conversation id from the FROM element (containing information about the sender) into the TO element (containing information about the receiver) of the response message, and vice versa. The first message only contains a value for the conversation id of the sender, the conversation id element of the TO element is empty.

The following figure shows how conversation name and id are created and exchanged:

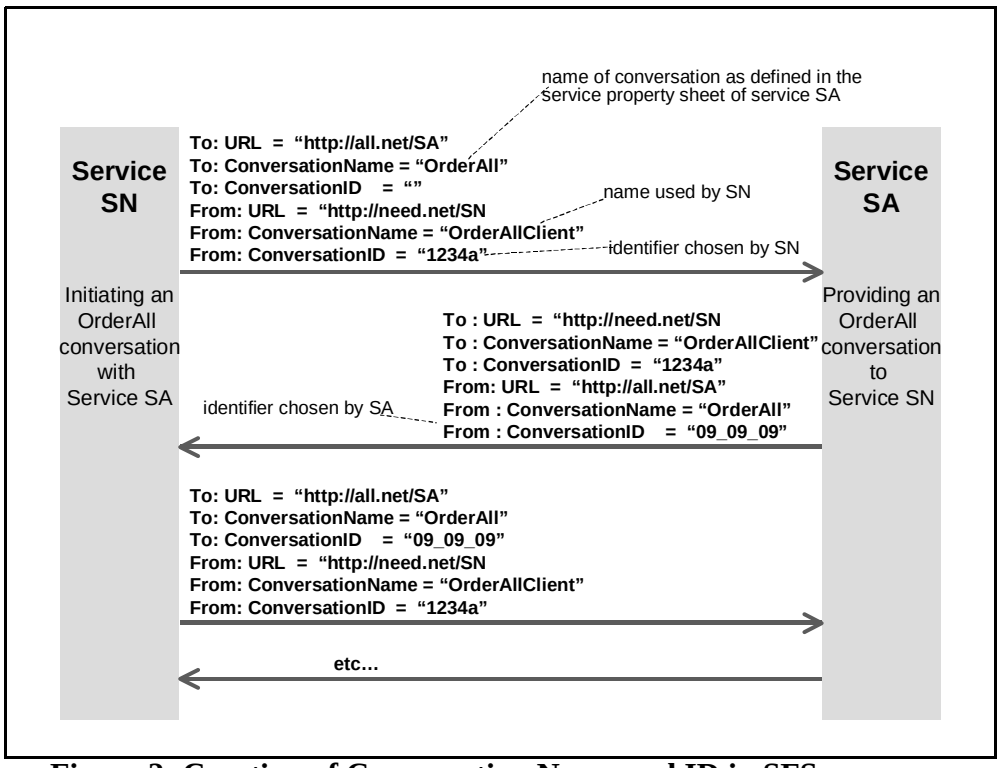


Figure 2: Creation of Conversation Name and ID in SFS messages

URI of sender and receiver service: These are the URIs that have to be used by the other party to respond to the message. They identify the end-points to the messaging layer of both services. The URI in the TO element of the first message of a conversation is the URI as defined in the ser-

vice descriptor of this message. If SFS is mapped onto HTTP, then the URI of the receiver in the SOAP header and the URL in the HTTP header may be identical, or the URI in the SOAP header may be more detailed e.g. if several services share a common servlet for dispatching, or the URL in the HTTP header may be completely different if it designates a SOAP server that will forward the message to its final destination.

Property tags:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
Properties	No	Once	Contains information about the message itself.	MessageID, Subject, SentAt, ReceivedAt, ExpiresAt
MessageID	No	Once	Identifier for this message. The identifier has to be unique for Sender within this conversation instance.	String
SentAt	No	Once	Time when the message is sent. This element is filled in by the sender of the message.	timeInstance data type of XML Schema
ReceivedAt	No	Once	Time when the message is sent. This element is filled in by the receiver of the message.	timeInstance data type of XML Schema
ExpiresAt	No	Once	Time when the message can be discarded without ever having been handled by the receiver. This element is filled in by the sender of the message.	timeInstance data type of XML Schema
Subject	No	Once	Classification of the message.	String

ReceivedAt: While the message is transported to the receiver this field is empty. It gets a value once the message has been received, and is of importance if the message gets saved or buffered.

ExpiresAt: This is a flag targeted at messaging software at the receiving end. If the message has been expired before it arrives at the receiver, or before the receiver has time to deal with it, then the messaging software at the receiving end can discard this message without even forwarding it to the business logic. Information about the validity of business data that is relevant for the business logic (e.g. until when an offer is valid) has to be part of the business data payload in the SOAP body. ExpiresAt is a pure messaging layer tag.

Subject: The exact semantic and content of this field is not defined by SFS, therefore it has to be defined by the eco-system and is of limited value.

Manifest tags:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
Manifest	No	Once	Contains information about binary attachments.	Reference
Reference	No	Multiple, one for each attachment	References the attachments with are in additional MIME parts of the message.	Description
Description	No	Once	Additional information about attachment.	String

Attributes of routing elements:

Element	*Attribute*	*Required*	*Occurs*	*Semantics*	*Value*
Reference	URI	Yes	Once	References the appropriate MIME part containing this attachment.	String containing name of attachment.

URI attribute of Reference element: This value of this attribute corresponds to the value of the Content-ID tag of the MIME part containing the corresponding attachment.

3.1.4 SFS Message tags: Schema

```
<?xml version='1.0'?>
<schema name = "SFSMessaging"
        targetNameSpace='http://www.e-speak.net/Schema/header'
        xmlns='http://www.w3.org/1999/XMLSchema'
        xmlns:ES-HEADER='http://www.e-speak.net/Schema/header'>
  <annotation>
    <documentation>SOAP header elements for SFS messages</documentation>
  </annotation>

  <!-- address type  -->
  <complexType name='addressType'>
   <element name='URI'  type='uri-reference' minOccurs='1' maxOccurs='1'/>
   <element name="ConversationName" type="String" minOccurs='1' maxOccurs='1'/>
   <element name="ConversationID" type="String" minOccurs='1' maxOccurs='1'/>
  </complexType>

  <!-- Route information -->
  <element name='Route' minOccurs='1' maxOccurs='1'>
    <complexType>
      <element name='To' type='ES-HEADER:addressType' minOccurs='1' maxOccurs='1'/>
      <element name='From' type='ES-HEADER:addressType' minOccurs='1'
             maxOccurs='1'/>
    </complexType>
  </element>

  <!-- Message properties -->
```

```
  <element name='properties' minOccurs='0' maxOccurs='1' content='elementOnly'>
    <complexType>
      <!-- a unique ID to identify the message -->
      <element name='MessageID' type='uri-reference' minOccurs='0' maxOccurs='1' />
      <!-- time when the message is sent -->
      <element name='SentAt' type='timeInstance' minOccurs='0' maxOccurs='1' />
      <!-- time when the message is received -->
      <element name='ReceivedAt' type='timeInstance' minOccurs='0' maxOccurs='1' />
      <!-- time when the message expires -->
      <element name='ExpiresAt' type='timeInstant' minOccurs='0' maxOccurs='1' />
      <!-- synchronous or asynchronous message -->
      <!-- the subject -->
      <element name='Subject' type='string' minOccurs='0' maxOccurs='1' />
      <!-- any extra elements -->
      <any minOccurs='0' maxOccurs='unbounded' namespace='##any' processCon-
             tents='lax'/>
    </complexType>
  </element>

  <!-- Manifest of the MIME message body -->
  <element name='manifest' minOccurs='0' maxOccurs='1' >
    <complexType>
      <!-- refer to a sub-part in the body -->
      <element name='reference' minOccurs='0' maxOccurs='unbounded' >
        <complexType>
       <element name='description' type='string' minOccurs='0' maxOccurs='1'/>
          <attribute name='uri' type='uri-reference' use='required'/>
        </complexType>
      </element>
    </complexType>
  </element>

</schema>
```

3.2 Mapping to Transport layer

SFS does not mandate any specific transport protocol. SFS specifies the mapping to HTTP as an example. An alternative to using SOAP/MIME/HTTP is to use ESIP for the messaging and transport layer. ESIP is the e-speak engine to engine communication protocol. If the services communicating with each other are all hosted by e-speak engines, then using ESIP is the easiest approach for the service provider and user. All the details of mapping SFS messages to ESIP are taken care of by the engine, and are hidden from the user.

3.2.1 Mapping of SFS messages to HTTP

SFS uses HTTP-POST with multipart MIME format. The body of the HTTP messages carries the SFS message. The first MIME part of the body carries the SOAP message, additional parts can carry attachments if there are any. The following example shows the HTTP header of a typical SFS message mapped to HTTP:

Each document exchange defined in the conversation (i.e. in the CDL description of the conversation) is mapped to an HTTP-POST, independent of the document exchange being part of an asyn-

chronous or synchronous interaction. Each HTTP-POST gets answered by an HTTP-RESPONSE without payload. This HTTP-RESPONSE either acknowledges the receipt of the message, or reports an error. The format of these responses is either simply according to SOAP standard, or extended by reliable messaging information if a reliable messaging layer is added on top of the HTTP transport layer. The conversation layer is not aware of any HTTP-RESPONSE messages, i.e. the conversation controller and business logic software does not receive any HTTP-RESPONSE messages, these are entirely handled by the transport software, and, if necessary, by and additional reliable messaging layer on top of transport. Of course, if the messaging layer cannot deliver the message, it may raise an exception to the higher levels like conversation controller and business logic.

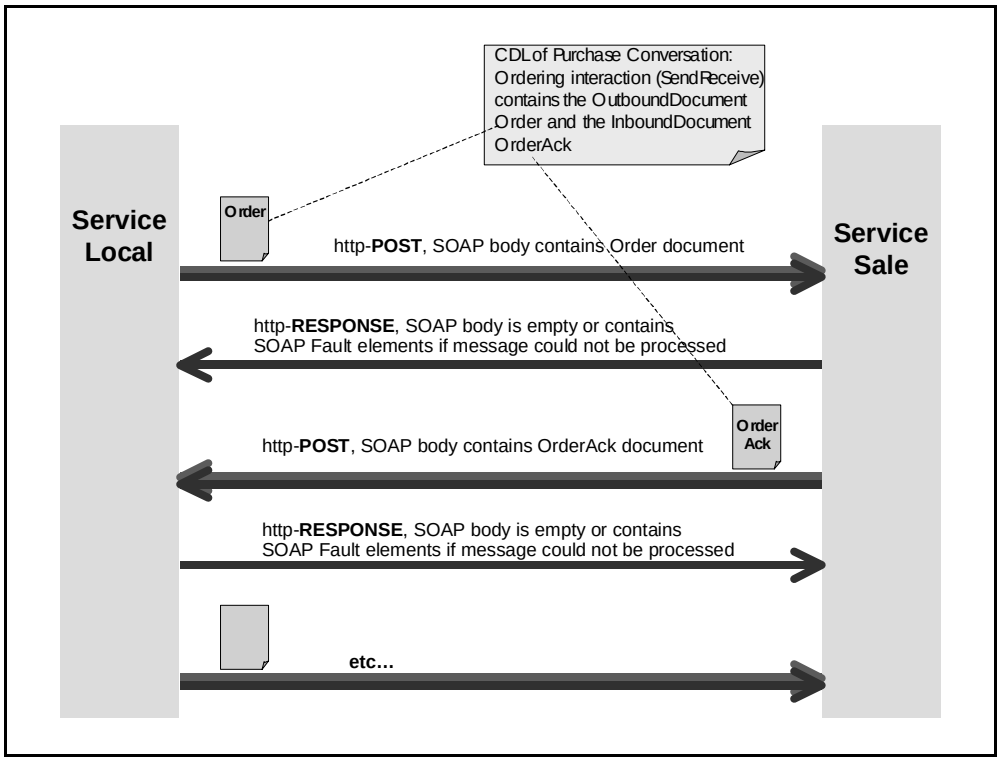


Figure 3: Mapping of business documents to HTTP messages

If the business logic needs to have explicit acknowledgement of the receipt of a message by the other party (e.g. if the answer to a request could take some hours, but the sender wants to know before if the other party actually has received the message), then the conversations definition in CDL must contain explicit acknowledge messages. Also the appropriate time-outs must be defined in the conversation definition (not yet possible with CDL 1.0). These business level acknowledgements are again mapped to HTTP-POST messages. There is a clear distinction between the acknowledge documents exchanged on the business level and defined in the conversation definition, and the HTTP-RESPONSES received and discarded on the messaging level without any influence on the conversation and business logic.

3.2.2 Mapping of SFS messages to HTTPS

In order to achieve secure transport, SFS messages can be mapped to HTTPS instead of HTTP. The mapping is analogue to the one shown for HTTP.

3.3 SFS Messages: Example

SFS message mapped to HTTP

The following message contains a PurchaseOrder document and sends it in the body of a SOAP message mapped to an HTTP-POST to the server www.ordering.com who hosts the ordering service offering the purchase conversation.

```
POST /StockQuote HTTP/1.1
             Host: www.ordering.com
             Content-Length: nnnn
             MIME-Version: 1.0
Content-Type: Multipart/Related;
             boundary="-----------1234567890";
             type=text/xml;
SOAPAction: ""

-----------1234567890
Content-Type:text/xml;
Content-Transfer-Encoding: 8bit
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope encodingStyle="http://schemas.xmlsoap.org/soap/encoding"
      xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope”
   xmlns:TRANSP=”http://schemas.e-speak.net/transport”>
  <SOAP-ENV:Header>
    <TRANSP:Route mustUnderstand="1">
      <TRANSP:From>
        <TRANSP:URI>http://www.SmallCompany/procurement/</TRANSP:URI>
        <TRANSP:ConversationName>purchase</TRANSP:ConversationName>
        <TRANSP:ConversationID>354</TRANSP:ConversationID>
      </TRANSP:From>
      <TRANSP:To>
        <TRANSP:URI> http://www.BestOrder.com/OrderService</TRANSP:URI>
        <TRANSP:ConversationName>purchase</TRANSP:ConversationName>
        <TRANSP:ConversationID>76</TRANSP:ConversationID>
      </TRANSP:To>
    </TRANSP:Route>
    <TRANSP:Properties>
      <TRANSP:SentAt>Oct/23/2000::23/10</TRANSP:SentAt>
      <TRANSP:MessageID>345-ertert-34m6s3</TRANSP:MessageID>
    </TRANSP:Properties>
    <TRANSP:Manifest></TRANSP:Manifest>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <MYCOPO:PurchaseOrder xmlns:MYCOPO=”http://myco.com/schemas/po”>

   <!—Here is where the actual PO document is represented. ->

    </MYCOPO:PurchaseOrder>
  </SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>
-----------1234567890
```

A SFS message with attachments

The following message contains not only XML payload in the SOAP body, but has also binary payload as attachments in additional MIME parts of the message:

```
POST /StockQuote HTTP/1.1
             Host: www.ordering.com
MIME-Version: 1.0
Content-Type: Multipart/Related;
             boundary="-----------1234567890";
             type=text/xml;
SOAPAction: ""

-----------1234567890
Content-Type:text/xml;
Content-Transfer-Encoding: 8bit

<?xml version='1.0' ?>
<SOAP-ENV:Envelope  xmlns:SOAP-ENV='http://schemas.xmsoap.org/soap/envelope/'>
  <SOAP-ENV:Header   xmlns:TRANSP=”http://schemas.e-speak.net/transport”>
    <TRANSP:Route  mustUnderstand =1>
      <TRANSP:TO     :::::      </TRANSP:to>
      <TRANSP:FROM>    ::::       </TRANSP:FROM>
    </TRANSP:Route>
    <!--Manifest about attachments for the dispatcher -->
    <TRANSP:manifest xmlns:TRANSP='http://www.e-speak.net/Schema/header/'>
      <TRANSP:reference uri='order-form'>
        <TRANSP:description>The order form</TRANSP:description>
      </TRANSP:reference>
      <TRANSP:reference uri='catalog'>
        <TRANSP:description>The catalog document</TRANSP:description>
      </TRANSP:reference>
    </TRANSP:manifest>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <a:Order xmlns:a='http://www.ServiceB.com/Schemas/order/'>
      <!-- references to attached documents, because the hrefs do not appear
                in the XML part, the receiving conversation controller knows to look
                for them in the attachments-->
    <a:Item href = ‘catalog’ />
   <a:SignedOrder href = ‘order-form’ />
      <!-- other content of the order -->
         ::::::::::::
    </a:Order>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

-----------1234567890
Content-Type: application/msword
Content-Transfer-Encoding: binary
Content-ID:<order-form>
::::::: <!—the actual data, MIME encoded -->
-----------1234567890
```

```
Content-Type:application/pdf
Content-ID:<catalog>
::::::: <!—the actual data, MIME encoded -->
-----------1234567890
```

4 Conversation Definition Language CDL

4.1 Introduction

The goal of CDL is to provide a standard way of describing conversations, and thus to enable rich interactions amongst services. These interactions include the introspection of services, the match-making of services by service directories. CDL is used for specifying predefined conversations in SFS, but also for specifying conversations by vertical and domain specific standards bodies or by the participants of a specific eco-system.

A conversation specification in CDL is itself an XML-document. Using XML for specifying conversations has the big advantage that specifications are formal and can be easily exchanged e.g. in introspection conversations and be interpreted by software. CDL describes interactions that contain documents relevant for the business logic. When conversations described by CDL are mapped to a specific transport protocol.

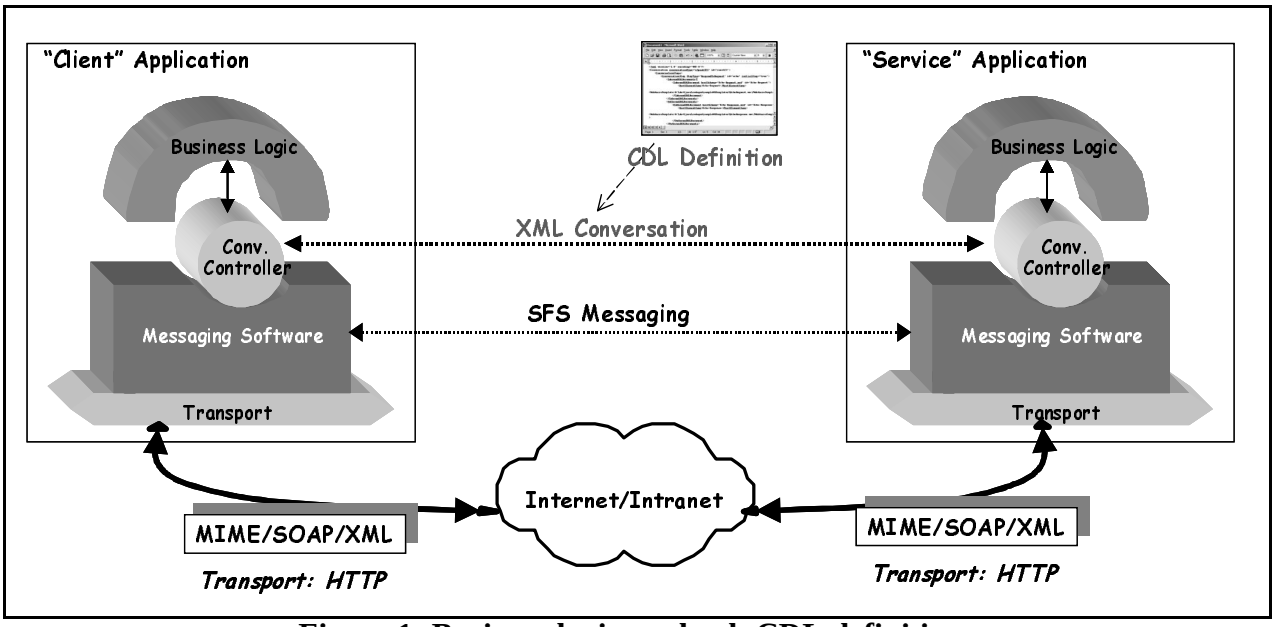


Figure 1: Business logic payload: CDL definitions

Projected description of conversations:

CDL describes a conversation from the point of view of a specific participant, i.e. it is a projected description. A conversation definition done in CDL specifies all the incoming and outgoing document for one of the participants in the conversations, normally for the one taking on the role of a service provider. This definition is than distributed to everybody else interested in either providing the same type of service or in using the service. In order for a potential client of the service to implement the conversation described for the server role, the client has to map the description to his own role, a task easily achieved by a tool.

One specific service can participate in several conversations, either in several instances of the same conversation at the same time, or by taking on different roles in different conversations.

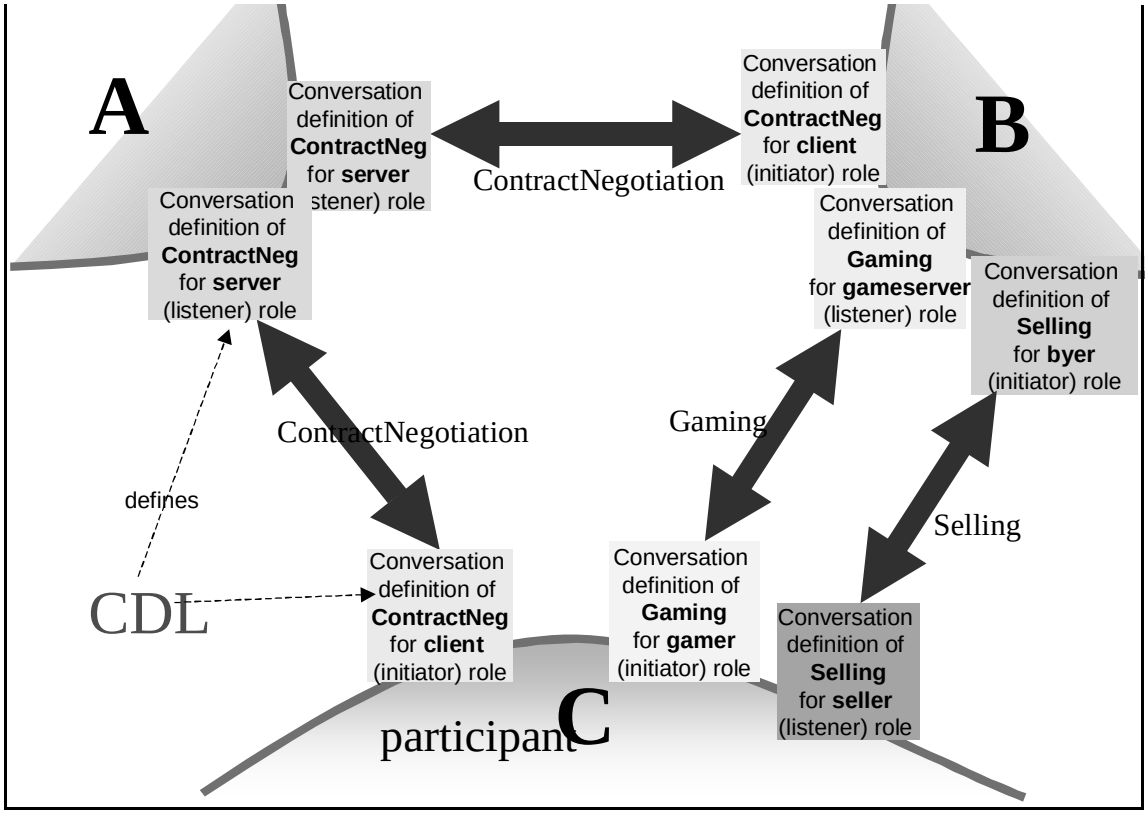


Figure 2: Defining conversations for specific roles of a conversation

Above figure shows an eco-system with three participating e-services, some of them participating in more than one conversation instance. All the conversations are two party conversations, with each participating e-service being either in the role of the *listener* or *initiator.* The initiator is the party that sends the first message in the conversation. Participant A is involved in two instances of a ContractNegotiation conversation, each time in the role of the listener. B is involved in a ContractNegotiatioin conversation, a Selling conversation, and a Gaming conversation, in the first two conversation instances as an initiator (client for negotiation, byer), in the third one as the listener (gameserver).

By default, two party conversations get defined and published for the *listener* e-service (darker colors in diagram). When *initiator* e-services get implemented they derive their conversation definition from the published listener conversation definitions. In an eco-system all conversation definitions for the listener role should be published and registered in a service registry along with the e-services realizing these conversations. Both, listener and initiator conversations carry the same name, yet they can be distinguished by their initial interaction. Listener conversations have an initial interaction that is a Receive or a ReceiveSend interaction.

Content and limitations of the current CDL:

For two parties to communicate with each other the following elements have to be specified in their conversation definitions:

- the **interactions** that can occur between the two parties,
- the format and content of the **documents** exchanged in the interactions (i.e. their xml-schemas),
- the possible **orders** of the interactions (i.e. the state machine that controls the conversation).

The current version of CDL can only define two-party conversations. Conversations where more than two participants are involved can be described by future extensions to CDL. Also not yet included in the current CDL are time-outs between messages, transactions, and events.

The following sections of this chapter contain the specification of CDL and an example of a conversation definition.

4.2 The elements of CDL

4.2.1 Overview

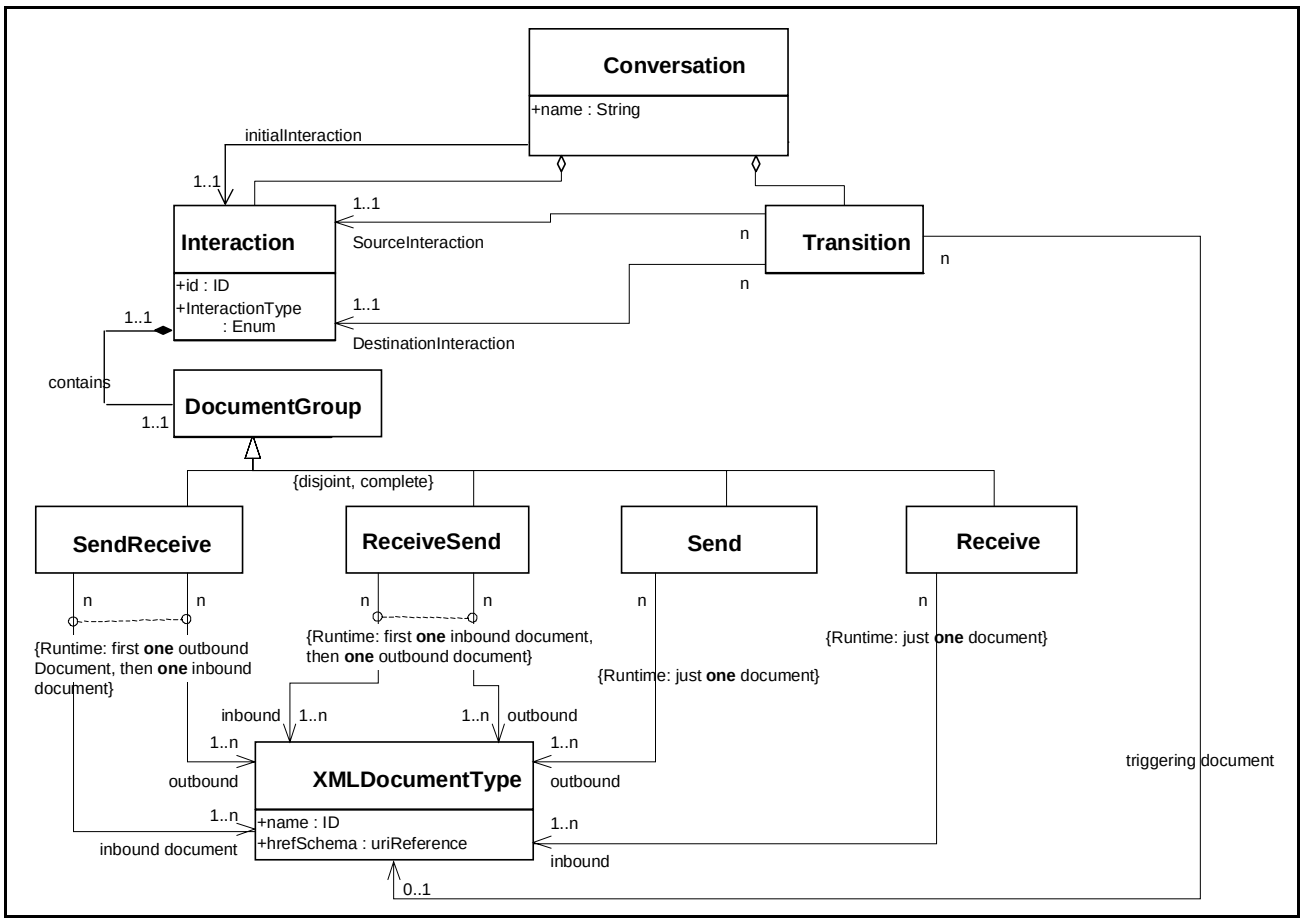


Figure: UML model of the CDL language

A conversation definition document in CDL essentially contains two parts:

- a **list of possible interactions** (and the documents they exchange)
- a **list of the possible transitions** between these interactions (i.e. the allowed orders of these interactions).

The schemas of the documents exchanged are not part of the CDL specification document, they are separate XML documents and are referenced by their URL in the interaction elements of the conversation specification.

4.2.2 Interactions

In CDL, four types of **interactions** that can be specified. These are:

- **Send**: an interaction that contains exactly one message/document that is sent out (an *OutboundXMLDocument*)
- **Receive**: an interaction that contains exactly one message/document that is received (an *InboundXMLDocument*)
- **SendReceive**: an interaction where the participant first sends out a document and then receives a document
- **ReceiveSend**: an interaction where the participant first receives a document and then sends out a document

Interactions are always specified from the point of view of one participant, therefore we distinguish between inbound and outbound documents. For all the interactions the conversation specification can list more than one possible inbound or outbound document. However, at runtime exactly one document of all the possible documents in the list is exchanged. Send and Receive interactions are one-way interactions.

The **transition** elements represent the possible orders of interactions, each transition element containing one possible transition from one specific interaction (the *SourceInteraction*) to another one (the *DestinationInteraction*). Because an interaction can list several possible documents to be exchanged, a transition element also has to state for which document of the source interaction it is valid (the *TriggeringDocument*). In case of the source interaction being a SendReceive or ReceiveSend interaction, the triggering document has to refer to the second document exchanged. A *Default Transition* or *Exception Transition* without a triggering document can be specified for any of the defined or any unexpected documents.

In order to mark the beginning of the conversation, the conversation specification marks one interaction as the *initialInteraction*. For a conversation specification to make sense, the interactions and transitions should form a complete graph where all the interactions can be reached from the initial interaction.

In the following we describe these elements into more detail.

4.2.3 Document Types

The interaction between service-consumer and service-provider is achieved through XML document exchange[1]. A conversation definition language must be able to define all the input and out-

put document types. The document type definitions refer to the schema that the document corresponds[1] to and also serve to declare an id for the document that can be used within the rest of the conversation definition. For example, the following definition defines an input document that conforms to a purchase order schema defined in PurchaseOrderRQ.xsd.

```
<InboundXMLDocument
    hrefSchema="http://foo.org/PurchaseOrderRQ.xsd"
    id="PurchaseOrderRQ">
</InboundXMLDocument>
```

In a conversation definition, the document types are declared within the interaction definitions. There are two types of document type declarations in the conversation definitions depending on whether the document is expected as input in an interaction (InboundXMLDocument) or whether the document is produced as output in an interaction (OutboundXMLDocument).

4.2.4 CDL Interaction Definition

An interaction is an exchange of one or two messages between a service and its client. Currently there are two types of interactions that are supported: one-way (Send interaction, Receive interaction) and two-way (SendReceive interaction, ReceiveSend interaction).

One-way Interactions:

These interactions represent a single one-way message being sent or received by a participant. There are two sub-types of one-way interactions: Receive, and Send. The Send interaction represents a message sent out by a participant, and on the other hand a Receive interaction represents a message being received by a participant.

The following two interactions represent a Receive and a Send interaction. The following interaction represents a simple Receive interaction that receives a purchase order.

```
<Interaction interactionType="Receive" id="PO">
  <InboundXMLDocuments>
     <InboundXMLDocument id="PurchaseOrderRQ"
        hrefSchema="http://foo.org/PurchaseOrderRQ.xsd">
     </InboundXMLDocument>
    </InboundXMLDocuments>
</Interaction>
```

Note that in each of these interaction definitions, there is the ability to select one from a set of incoming or outgoing documents. For example, a receive interaction can receive a document from a possible set of documents, and a send interaction can be defined to send a document from a set

1. Binary attachments are also supported by SFS, yet they are not defined by the CDL of a conversation. The ability to attach binary documents is specified in the XML schema for the payload, addresses this issue (see also the Manifest tags in 3.1 “SFS Messages“).
1. CDL only supports XML schema specifications of payload, as schemas seem to become the prevailing means of describing data exchanged on the internet. Therefore SFS compatible servers only have to support schemas. Existing DTD specifications can easily be translated into XML schemas.

of documents. Therefore, the CDL of a conversation can define several Inbound- or Outbound-XMLDocuments for one interaction. At execution time, only one of them can be sent or received.

An interactionType of Receive must be associated with InboundXMLDocuments. An interaction-Type of Send must be associated with OutboundXMLDocuments.

Two-way Interactions:

Two-way interactions can have one of two forms: SendReceive or ReceiveSend corresponding to whether the participant sent out a message for which it got a response (SendReceive) or whether the participant responded to a request that it received (ReceiveSend).

ReceiveSend Interactions: Each such interaction is the logical unit of receiving a request and then returning a response. The interaction is not complete until the response has been sent.

```
<Interaction interactionType="ReceiveSend" id="Quotation">

  <InboundXMLDocuments>
    <InboundXMLDocument id="PurchaseOrderRQ"
         hrefSchema=”http://foo.org/PurchaseOrderRQ.xsd”>
    </InboundXMLDocument>
  </InboundXMLDocuments>
  <OutboundXMLDocuments>
    <OutboundXMLDocument" id="InvoiceRS"
         hrefSchema="http://foo.org/InvoiceRS.xsd>
    </OutboundXMLDocument>
  </OutboundXMLDocuments>

</Interaction>
```

SendReceive Interaction: Each such interaction is the logical unit of sending a request and then receiving a response. The interaction is not complete until the response has been received.

```
<Interaction interactionType="SendReceive" id="Payment">

<OutboundXMLDocuments>
      <OutboundXMLDocument id="Receipt"
          hrefSchema="http://foo.org/ReceiptRS.xsd">
      </OutboundXMLDocument>
    </OutboundXMLDocuments>

  <InboundXMLDocuments>
     <InboundXMLDocument" id="Payment"
           hrefSchema="http://foo.org/Payment.xsd>
      </InboundXMLDocument>
    </InboundXMLDocuments>

</Interaction>
```

As with Send and Receive interactions, SendReceive and ReceiveSend interactions can specify multiple inbound and outbound documents. At run time exactly one inbound and exactly one out-

bound message from these sets will be exchanged. For example, the following is a definition of a simple ReceiveSend interaction that has multiple inbound documents.

```
<Interaction interactionType="ReceiveSend" id="Start">

  <InboundXMLDocuments>
    <InboundXMLDocument id="LoginRQ"
        hrefSchema="http://conv123.org/LoginRQ.xsd">
    </InboundXMLDocument>
    <InboundXMLDocument hrefSchema="RegistrationRQ.xsd"
        id="RegistrationRQ">
    </InboundXMLDocument>
  </InboundXMLDocuments>
  <OutboundXMLDocuments>
    <OutboundXMLDocument id="ValidLoginRS"
         hrefSchema="http://conv123.org/ValidLoginRS.xsd" >
    </OutboundXMLDocument>
    <OutboundXMLDocument id="RegistrationRS"
         hrefSchema="http://conv123.org/RegistrationRS.xsd" >
    </OutboundXMLDocument>
  </OutboundXMLDocuments>

</Interaction>
```

Note that a conversation that breaks up this single interaction into separate interactions each with a single incoming and a single outgoing document may not have the same semantics as the example. In the example, the business logic may choose to change the outgoing document without changing the definition of the conversation.

The Send interaction is the dual of the Receive interaction, and *vice versa*. Similarly, the SendReceive interaction is the dual of the ReceiveSend interaction and *vice versa*. The notion of duality is important when two entities have to interact. Essentially, two or more entities can successfully interact if the conversation definitions that the two entities use are duals of each other.

4.2.5 Transition:

A conversation can proceed from one interaction to another according to the legally defined interactions in the service conversation definition document. The ordering amongst the interactions is defined in the transition element (the attribute transitionType is optional for the basic transitions).

```
<Transition transitionType=”Basic”>
     <SourceInteraction href="#Invoice"/>
     <DestinationInteraction href="#Receipt"/>
     <TriggeringDocument href=”#invoiceRS”/>
</Transition>
```

The *SourceInteraction* references an interaction that can precede the DestinationInteraction when the conversation is executed. Similarly, the *DestinationInteraction* references one of the interactions that can follow the SourceInteraction when the conversation is executed. All transitions together thus specify all possible sequences of the interactions.

The *TriggeringDocument* is an additional constraint on the transition, needed because interaction specifications can list several possible documents to be exchanged. The TriggeringDocument references which document must have been exchanged in order for this transition to happen.

Note that there is a correspondence between the triggering document and the Source interaction definition. If the source interaction is a SendReceive interaction, the triggering document has to be among the list of incoming documents that are defined in the InboundXMLDocuments group in the interaction definition. Similarly, if the interaction is a ReceiveSend interaction, the triggering document has to be among the OutBoundXMLDocuments group. This restriction also holds for Send and Receive interactions. That is, if the source interaction is a Receive interaction, the triggering document has to be among the InboundXMLDocuments group that is defined in the interaction.

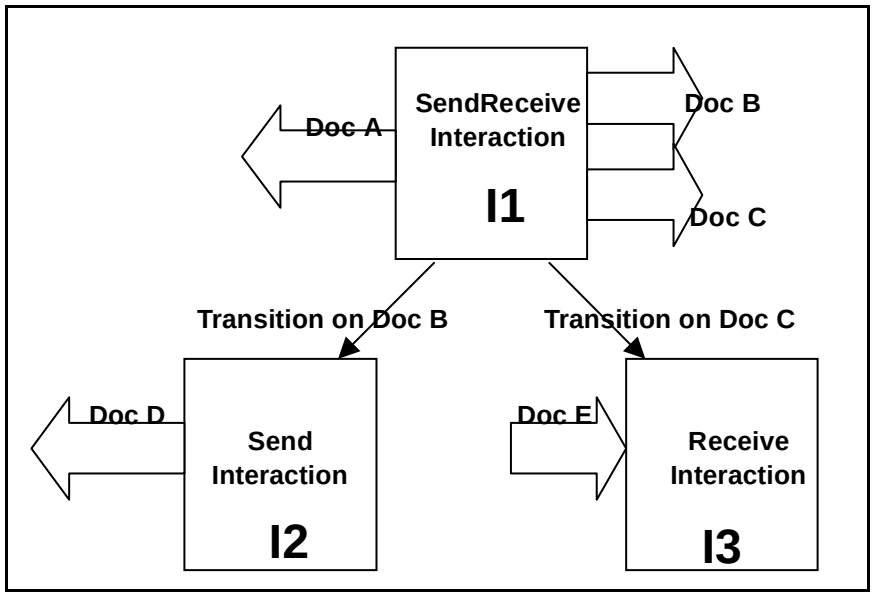


Figure 3: Transitions with triggering documents

The figure above shows the generic relation ship between interactions, transitions and triggering documents. Interactions are drawn as generic blocks with possible arrows representing the direction of the messages in the interaction. An arrow pointing to the right is a document that is received, and an arrow pointing to the left is a document that is sent. Furthermore, the actions on the left of the interaction box occur before the actions on the right of the box. Returning to the figure above, interaction I1 is a SendReceive Interaction that sends out one document and expects one of two documents as replies. However on each of those possible replies, the next allowable interaction can be different. Therefore, in the figure above, on receiving document B in Interaction I1, the next possible interaction is interaction I2 and on receiving document C the next possible interaction is interaction I3.

In CDL 1.0 there are two special transitions: Default Transition and Exception Transition (see section Exceptions). For each source interaction, the CDL definer can also specify one *default transition*. This is a shortcut in case the same transition can be triggered by more than one trigger-

ing document. This transition takes place if a document is received that is defined in the source interaction yet that does not appear as a triggering document in any of the other defined transitions for this source interaction. No TriggeringDocument is specified. There can be at most one default transition definition per SourceInteraction. Default interactions are defined as follows:

```
<Transition transitionType=”default”>
    <SourceInteraction href=”#Invoice”/>
    <DestinationInteraction href=”#InvExpected”/>
</Transition>
```

4.2.6 Exceptions

Similarly to the default transition the *exception transition* does not define any triggering document. Yet the exception transition happens when a document that is not expected at the current interaction is received while executing the conversation. Excpetion interactions are defined as follows:

```
<Transition transitionType=”Exception”>
    <SourceInteraction href=”#Invoice”/>
    <DestinationInteraction href=”#InvExpected”/>
</Transition>
```

In this case, any document not specified in the SourceInteraction of “#Invoice” will result in a transition to the interaction “#InvExpected”. These ExceptionTransitions are more likely to be defined for Receive, ReceiveSend, and SendReceive interactions because some document involved in the interaction potentially originates in another enterprise. Note that no TriggeringDocument is specified. There can be at most one exception transition definition per SourceInteraction.

Exceptions occuring while handling an expected document and payload signalling bussines logic exceptions are not treated specially in CDL. An exception is denoted simply as one of the outgoing documents. The implementation at each end can treat the exception document as it chooses. Consider for example a simple login interaction that accepts a login request (represented by LoginRQ.xsd) and returns a login response. In order to indicate that the login interaction can result in an InvalidPasswordException, the interaction definition contains the definition of the InvalidPasswordException document as one of the possible responses to the LoginRQ document.

```
<Interaction interactionType="ReceiveSend" id="Start" >
    <InboundXMLDocuments>
      <InboundXMLDocument id="LoginRQ" hrefSchema="http://conv123.org/LoginRQ.xsd"/>
    </InboundXMLDocuments>
    <OutboundXMLDocuments>
        <OutboundXMLDocument id="ValidLoginRS"
               hrefSchema="http://conv123.org/ValidLoginRS.xsd"/>
        <OutboundXMLDocument id="InvalidPassword”
               hrefSchema="http://conv123.org/BadPasswordExcp.xsd" />
     </OutboundXMLDocuments>
</Interaction>
```

4.2.7 Well-formed Conversation Definitions

Conversation definitions have one initial interaction, denoted by the attribute initialInteraction of the element Conversation. The possible end-states of the conversation are given by all those interactions that do not appear as a source interaction in any transition. Under normal circumstances a conversation instance is expected to execute until one of these interactions has been reached, upon which the conversation is terminated. Any interaction other than the initial interaction has to appear in at least on transition as the destionation interaction. Transitions of type default or exception are specified at most once per source interaction. Conversations fulfilling these conditions are considered as well-formed. Only well-formed conversations should be published and used.

4.3 Complete Conversation Example

This section contains a complete definition of a simple conversation. It is annotated to explain the various features.

There are two parties involved in this conversation: the *Listener* and (in this case the service provider) the *Initiator* (in this case the client).

```
<?xml version="1.0" encoding="UTF-8"?>
<Conversation xmlns=”http://www.e-speak.net/schema/conversation”
              name="http://conv123.org/conv123" initialInteraction=”#Start”>
```

The name is the shared piece of information needed by both parties so they realize the same conversation type in their service implementation. This name would also appear in the SFS messages exchanged between the two parties, and in the service descriptor of the listener party. Here, we’ve shown a URL, but a reference to a UDDI tModel would serve as well.

```
  <ConversationInteractions>
```

The conversation definition begins by specifying the set of interactions involved. Each has a type and a set of incoming and outgoing messages that depend on the type.

```
    <Interaction interactionType="ReceiveSend" id="Start">
```

The first interaction in the conversation, denoted by `initialInteraction` in the Conversation tag is normally of type Receive or ReceiveSend when the conversation is defined from the perspective of the service provider, in this case from the perspective of the listener. The Initiator need only reverse the sense of all tags to understand its role in the conversation. In this case, the Initiator knows that it starts the conversation with the complementary interactionType, here SendReceive. The type of the interaction also determines the document group that occurs within the interaction.

```
      <InboundXMLDocuments>
         <InboundXMLDocument id="LoginRQ"
            hrefSchema=”http://conv123.org/LoginRQ.xsd” />
         <InboundXMLDocument id="RegistrationRQ"
             hrefSchema="http://conv123.org/RegistrationRQ.xsd" />
      </InboundXMLDocuments>
```

The Listener is expecting a login or a registration request. The Initiator can parse the conversation to determine that it must send a message of one of these types.

```
<OutboundXMLDocuments>
   <OutboundXMLDocument id="ValidLoginRS"
       hrefSchema="http://conv123.org/ValidLoginRS.xsd">
   </OutboundXMLDocument>
   <OutboundXMLDocument id="RegistrationRS"
                hrefSchema="http://conv123.org/RegistrationRS.xsd">
   </OutboundXMLDocument>
   <OutboundXMLDocuments id="InvalidLoginRS"
       hrefSchema="http://conv123.org/InvalidLoginRS.xsd">
   </OutboundXMLDocument>
</OutboundXMLDocuments>
```

Documents of one of three types can be returned, one denoting a successful login, another an unsuccessful one, and one a successful registration. While the information about an unsuccessful login could be carried in the body of a single type of document, using different document types allows exception handling to be defined as part of the conversation.

The Listener also accepts a registration request from new users and returns a RegistrationRS document. In this case, the return document contains any necessary error messages.

```
</Interaction>
```

That ends the first interaction. The rest of the definitions follow the same pattern.

```
<Interaction interactionType="ReceiveSend" id="LoggedIn">
  <InboundXMLDocuments>
     <InboundXMLDocument id="CatalogRQ"
          hrefSchema="http://conv123.org/CatalogRQ.xsd">
     </InboundXMLDocument>
  </InboundXMLDocuments>
  <OutboundXMLDocuments>
     <OutboundXMLDocument id="CatalogRS"
         hrefSchema="http://conv123.org/CatalogRS.xsd">
     </OutboundXMLDocument>
  </OutboundXMLDocuments>
</Interaction>

<Interaction interactionType="ReceiveSend" id="Registered">
   <InboundXMLDocuments>
      <InboundXMLDocument id="LoginRQ"
          hrefSchema="http://conv123.org/LoginRQ.xsd">
      </InboundXMLDocument>
   </InboundXMLDocuments>
   <OutboundXMLDocuments>
      <OutboundXMLDocument id="ValidLoginRS"
         hrefSchema="http://conv123.org/ValidLoginRS.xsd">
      </OutboundXMLDocument>
   </OutboundXMLDocuments>
 </Interaction>

 <Interaction interactionType="ReceiveSend" id="Catalogued">
     <InboundXMLDocuments>
       <InboundXMLDocument id="QuoteRQ"
```

```
          hrefSchema="http://conv123.org/QuoteRQ.xsd">
        </InboundXMLDocument>
      </InboundXMLDocuments>
      <OutboundXMLDocuments>
         <OutboundXMLDocument id="QuoteRS"
            hrefSchema="http://conv123.org/QuoteRS.xsd">
         </OutboundXMLDocument>
      </OutboundXMLDocuments>
    </Interaction>

    <Interaction interactionType="ReceiveSend" id="Quotation">
      <InboundXMLDocuments>
         <InboundXMLDocument id="PurchaseOrderRQ"
            hrefSchema="http://conv123.org/PORQ.xsd">
         </InboundXMLDocument>
      </InboundXMLDocuments>
      <OutboundXMLDocuments>
         <OutboundXMLDocument id="InvoiceRS"
            hrefSchema="http://conv123.org/InvoiceRS.xsd">
         </OutboundXMLDocument>
      </OutboundXMLDocuments>
    </Interaction>

   <Interaction interactionType="SendReceive" id="Invoiced">
     <OutboundXMLDocuments>
         <OutboundXMLDocument id="ConfirmationRS"
           hrefSchema="http://conv123.org/ConfirmRS.xsd" >
         </OutboundXMLDocument>
      </OutboundXMLDocuments>

      <InboundXMLDocuments>
         <InboundXMLDocument id="AuthorizePaymentRQ"
          hrefSchema="http://conv123.org/AuthPaymentRQ.xsd">
               </InboundXMLDocument>
      </InboundXMLDocuments>
    </Interaction>

    <Interaction interactionType="ReceiveSend" id="end">

               <InboundXMLDocuments/>
      <OutboundXMLDocuments/>
    </Interaction>

  </ConversationInteractions>
```

The next part of the definition of the conversation is the actions that occur when different document types are received or sent. It is this part that imposes the ordering. In essence, the interactions define the interface, while the transitions define the protocol.

```
  <ConversationTransitions>
```

Most transitions have three parts, a source, a destination, and a document type that triggers the transaction. Note that it is a document type, not an interaction that triggers the transition. That's why we are free to combine several interactions into one with several incoming and outgoing document types without changing the semantics of the conversation.

```
      <Transition>
```

```
    <SourceInteraction href="#Start"/>
    <DestinationInteraction href="#LoggedIn"/>
    <TriggeringDocument href="#ValidLoginRS"/>
</Transition>
```

When the conversation is in the Start interaction, the Initiator will switch to the LoggedIn interaction when a ValidLoginRS message is received. The Listener will do the same when it sends a message of this type. In this way, both parties are guaranteed to be in a compatible state, reducing the chance of having badly formed conversations.

```
<Transition>
    <SourceInteraction href="#Start"/>
    <DestinationInteraction href="#BadLogin"/>
    <TriggeringDocument href="#InvalidLoginRS"/>
</Transition>
```

The above transition illustrates how nicely exceptions fit into the CDL; each simply becomes a different transition. We can also handle all other possible document types in the ineraction quite simply.

```
<Transition transitionType=”Exception”>
    <SourceInteraction href="#Start"/>
    <DestinationInteraction href="#Start"/>
</Transition>

<Transition>
    <SourceInteraction href="#Start"/>
    <DestinationInteraction href="#Registered"/>
    <TriggeringDocument href="#RegistrationRS"/>
</Transition>
```

The remainder of the transition should be clear.

```
<Transition>
    <SourceInteraction href="#Registered"/>
    <DestinationInteraction href="#LoggedIn"/>
    <TriggeringDocument href="#ValidLoginRS"/>
</Transition>
<Transition>
    <SourceInteraction href="#LoggedIn"/>
    <DestinationInteraction href="#Catalogued"/>
    <TriggeringDocument href="#CatalogRS"/>
</Transition>
<Transition>
    <SourceInteraction href="#Catalogued"/>
    <DestinationInteraction href="#Quotation"/>
    <TriggeringDocument href="#QuoteRS"/>
</Transition>
<Transition>
    <SourceInteraction href="#Quotation"/>
    <DestinationInteraction href="#Invoiced"/>
    <TriggeringDocument href="#InvoiceRS"/>
</Transition>
<Transition>
    <SourceInteraction href="#Invoiced"/>
    <DestinationInteraction href="#end"/>
```

```
            <TriggeringDocument href="#ConfirmationRS"/>
        </Transition>
    </ConversationTransitions>
</Conversation>
```

4.4 CDL Element Details

This section summarizes the various elements of a CDL description and their constraints:

Conversation tag:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
Conversation	Yes	Once	This is the root element of the conversation definition document	ConversationInteractions, ConversationTransitions

Attributes of Conversation element:

Attribute	*Required*	*Occurs*	*Semantics*	*Value*
InitialInteraction	Yes	Once	Indicates the initial interaction	href
Name	Yes	Once	Reference to conversation	String, URN or tModel key

ConversationInteractions tag:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
ConversationInteractions	Yes	Once	Encapsulates the set of interactions that this service can have.	Interaction

Interaction tag:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
Interaction	Yes	Many times	Denotes one interaction	InboundXMLDocuments, OutboundXMLDocuments

Attributes of Interaction:

Attribute	*Required*	*Occurs*	*Semantics*	*Value*

interactionType	Yes	Once	Type of interaction	Receive, Send, Receive-Send, SendReceive
ID	Yes	Once	Reference to this interaction	

InboundXMLDocuments tag:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
InboundXML-Documents	Yes	Once per Receive, SendReceive or ReceiveSend interaction	Lists the possible incoming documents in this interaction (at runtime only of the possible ones will be chosen).	InboundXML-Document

InboundXMLDocument tag:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
InboundXML-Document	Yes	One or many per InboundXMLDocuments tag	References the type of document to be exchanged	

Attributes of InboundXMLDocument:

Attribute	*Required*	*Occurs*	*Semantics*	*Value*
hrefSchema	Yes	Once	Schema for document	href
Id	Yes	Once	Root element of document	String

OutboundXMLDocuments tag:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
OutboundXML-Documents	Yes	Once per Send, SendReceive or ReceiveSend interaction	Lists the possible incoming documents in this interaction (at runtime only of the possible ones will be chosen).	InboundXML-Document

OutboundXMLDocument tag:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
OutboundXML-Document	Yes	One or many per OutboundXMLDocuments tag	References the type of document to be exchanged	

Attributes of OutboundXMLDocument:

Attribute	*Required*	*Occurs*	*Semantics*	*Value*
hrefSchema	Yes	Once	Schema for document	href
Id	Yes	Once	Root element of document	String

ConversationTransitions tag:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
Conversation-Transitions	Yes	Once	This defines the set of all ordering transitions (defining all possible sequences of the interactions of this conversations)	Transition

Transition tag:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
Transition	Yes	Zero, once or more	Defines each ordering transition	SourceInteraction, DestinationInteraction, TriggeringDocument

Attributes of Transition:

Attribute	*Required*	*Occurs*	*Semantics*	*Value*
transitionType	Only if it is not a basic transition	Once	Denotes the kind of transition.	Basic, Default, Exception. If the attribute is omitted, the value of the attribute is “Basic”.

SourceInteraction tag:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
SourceInteraction	Yes	Once per Transition	Identifies the interaction in the conversation can precede the DestinationInteraction	

Attributes of SourceInteraction:

Attribute	*Required*	*Occurs*	*Semantics*	*Value*
href	Yes	Once	Reference to Interaction	href

DestinationInteraction tag:

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
DestinationInteraction	Yes	Once per Transition	Identifies the interaction in the conversation can follow the SourceInteraction	

Attributes of DestinationInteraction:

Attribute	*Required*	*Occurs*	*Semantics*	*Value*
href	Yes	Once	Reference to Interaction	href

TriggeringDocument

Tag Name	*Required*	*Occurs*	*Semantics*	*Contains*
TriggeringDocument	Yes	Optional	Identifies a document that must have been exchanged as part of the SourceInteraction in order for the transition to be allowed to happen	

Attributes of TriggerDocument:

Attribute	*Required*	*Occurs*	*Semantics*	*Value*
href	Yes	Once	Reference to either an InboundXMLDocument (if SourceInteraction of type Receive or SendReceive) or an OutboundXMLDocument (if SourceInteraction of type Send or ReceiveSend) of the SourceInteraction	href

The schema of the CDL language can be found in the appendix.

5 Vocabularies, Service Description and Introspection

One of the most important predefined conversations in SFS is introspection. Every service in SFS supports the introspection conversation. The introspecton conversations allow a potential service consumer to ask another service about its properties and interfaces. The introspection covnersations return a service descriptor describing the properties of the service, and the conversation definitions for all the conversations supported by the service.

The service descriptor contains meta-information about the service. Some of the content of the service descriptor is domain specific. In order to define the content of the domain specific parts of the descriptor, vocabularies are specified.

In this chapter we first introduce vocabularies. Then we specify the structure of the service descriptor, and finally specify the introspection conversations.

5.1 Vocabularies

5.1.1 Purpose of vocabulary definitions

Many of the meta-data elements in a service descriptor or an offer are predefined by SFS. Their names, properties and possible values are defined by XML-schemas in the sections about service descriptors and offers. However, parts of the service descriptor and offers are domain dependent. These parts are, respectively, the service description and offer description. The attributes of the domain specific parts are defined in vocabularies. An offer or a service descriptor references the vocabularies it uses, and can use all the terms defined in these vocabularies for the offer description or service description. Because vocabularies are domain specific, they cannot be defined by SFS, but must be defined by the service providers or by domain specific standards bodies.

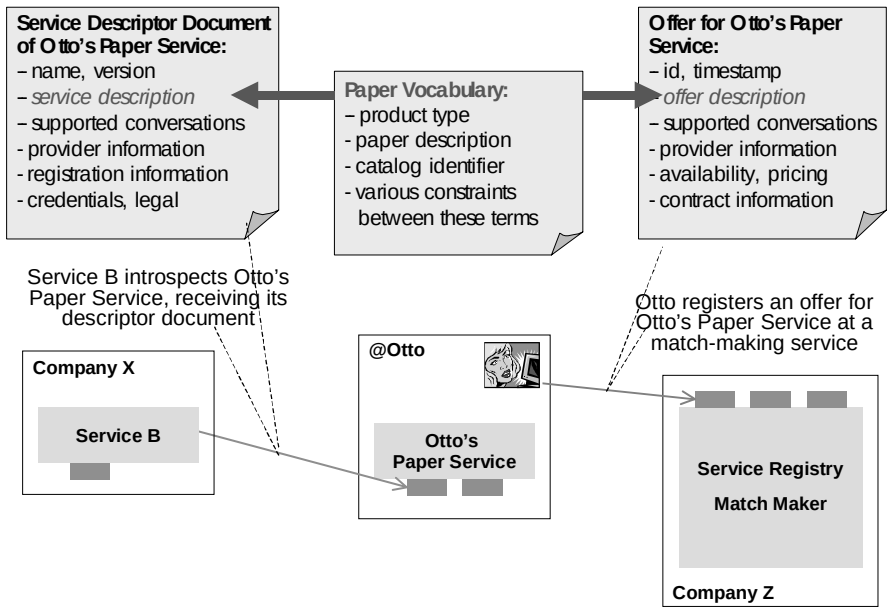


Figure 1: Vocabularies used in offers and service descriptors

In above figure, we have a service called “Otto’s Paper Service”, hosted by @Otto. @Otto registers an offer for this service with the service registry of company Z, where it can be queried by potential consumers. Furthermore, any entity that has the address of the service (e.g. after finding it at a match-maker or from previous use) can introspect the service and get back a service descriptor. Both, the service descriptor and the offer contain a description part whose meta-elements are defined by the vocabulary “Paper Vocabulary”.

Attributes and Vocabularies

An **attribute** is a meta-data element. It consist of the attribute name and the attriute value. In SFS, an attribute describes one specific aspect or property of a service or an offer.

A **vocabulary** defines the attributes that may or must be used in a service or offer description. The notion of a vocabulary definition language as introduced here builds on some of the notions that are prevalent in RDF schemas, and XML schemas. Essentially, a vocabulary is a framework for defining meta-data that can be used in service descriptions. XML-schemas define the type system and the structure of each meta-data entry. XML schemas define a rich extensible type system. XML schemas, however, only define the syntactic structure of the XML documents. We augment XML schemas with a simple constraint language to express simple constraints (such as some elements have to be non-null, references have to be valid, sum of line items plus tax should be total in PO, etc.) in order to quarantee the well formedness of a description.

Requirements for a vocabulary definition language

A language for defining vocabularies must fulfill the following requirements:

- It should be capable of defining the metadata for a wide variety of services. This means that if different vertical industries want to define their metadata for the services in their industry in different ways, the mechanism should allow that.
- It should allow the metadata to evolve in a gradual manner without changes to the backends or the enterprises that are using old versions of the metadata definition. This allows specific advertisers to differentiate their descriptions of the goods/services they sell with characteristics that enhance their advantage over their competitors.
- It should be compatible with existing metadata definition mechanisms so that existing web services can be advertised at matchmakers and discovered.
- It should identify portions of the metadata that are searchable.
- It should identify the portions of the offers that are visible to entities other than the creator of the offer. More specifically, it should provide for the specification of security rules that.
- It should support a rich type mechanism for modeling the services that it describes.

There are many ways to define the structure of meta-data. Standards bodies such as W3C are considering recommendations such as RDF (Resource Description Framework), that provide very flexible ways to define the meta-data for web resources. However, the RDF schemas themselves are not quite appropriate for defining the meta-data for e-services, and offers that e-services register with matchmakers. The main reason for this is that RDF is more about representing the relationships between web resources than the data types of individual attributes and their relationships.

5.1.2 Defining Vocabularies

The vocabulary definition language in SFS uses XML schemas to define vocabularies. The descriptions using these vocabularies are XML documents. Essentially, the relationship between vocabularies and any service or offer description is akin to the relationship between a table schema definition and rows in the table in database terms. The vocabulary defines the attributes that the description must or may specify. In addition, it also specifies the types of the values that any named attribute of the description can take.

The vocabulary definition language in SFS defines for each attribute:

- the name of the attribute
- the valuespace of the attribute defined by a datatype
- properties of the attribute

Furthermore, the vocabulary definition language defines the constraints that apply accross several attributes.

Vocabulary:

Each vocabulary is defined by one XML schema, containing one top-level element. This top-level element represents the vocabulary, its *name* is the name of the vocabulary. Each vocabulary declared in an XML schema has an associated *XML namespace*. The namespace is declared by a URI value of the targetNS attribute of a schema document. The URI value space includes URL's, URN's and other URI derivatives. For example:

```
<?xml version='1.0'?>
<schema targetNS="http://vocabularies.foo.com/paper">
  <element name=”PaperVocabulary”>
   <complexType content = "elementOnly">
   <--XML elements and XML attributes specifying the attributes of the vocabulary-->
     <element name="catalog-identifier" type=”string” required=”yes”/>
     <element name= "paper-description" type=”paper-description-type”/>
     <attribute name= "supplier-part-number" type=”number”/>
   </complexType>
  </element>
  <complexType name='paper-description-type'>
     <sequence>
       <element ref=”paper-desc” datatype=”string”/>
       <element ref=”paper-size” datatype=”integer”/>
       <element ref=”width” datatype=”integer”/>
       <element ref=”length” datatype=”integer”/>
     </sequence>
   </complexType>
</schema>
```

The vocabularies used in a description are referenced with their namespaces. Different vocabularies can be referenced in the same description as each XML fragment of the description can have its own namespace and thus its own vocabulary.

The meta-data elements or vocabulary attributes are defined as XML elements or attributes of the vocabulary element. In above example we only define a vocabulary that has only three attributes: catalog-identifier, paper-description, and supplier-part-number. The first meta-data element is

defined as an XML element of a simple data type, the second one as an XML element of a complex data type, and the third one as an XML attribute.

Datatypes of vocabulary attributes:

The type system used by vocabularies is the type system defined by the XML Schema data type recommendation and is available at http://www.w3d.org/TR/xmlschema-1/.

The datatypes defined can be put into one of the following categories:

- **Simple**: Simple data types are no further structured. XML Schema defines all the basic ones, and lets the user derive additional simple datatypes.
- **User defined aggregate**: XML Schema allows the user to define compex datatypes, consisting of several other complex or simple datatypes.

Properties of vocabulary attributes:

The vocabulary can also specify properties for the attributes. Possible properties are:

- **default**: contains the default value for the attribute, has to conform to the type specified for the attribute.
- **required**: a boolean that determines whether every this attribute has to be present in every description that uses this vocabulary.
- **ismultivalued**: a boolean that indicates whether the value of the attribute is a collections of values all of the same data type. Data types themselves can also be defined as lists. It is up to the user to choose between the two possibilities: multivalued attribute or a single valued attribute with a list as data type.
- **isreference**: a boolean that indicates whether the value of this attribute is a reference. If it is a reference it might be of XML Schema type URL, yet it might also be of a special user defined reference type. If a data type is a reference or not is important for consistency checks - references need to be resolvable, i.e. they need to point to a valid and accessable resource.

5.1.3 Schema of the vocabulary definition language

The schema of the vocabulary definition language as presented above is very similar to the schema for XML schema, as vocabularies are in essence schemas with some restrictions. The schema is described in details in the appendix of SFS 2.0.

5.1.4 Constraints of Attribute Values

Unrestricted schemas are limited in their ability to capture the meta-data of services. For instance, consider an offer description to sell a good that contains a reference to the owner of the good. We may want to ensure that the reference is a well-formed reference and not a dangling reference. The actual semantics of dangling depends on the type and representation of the reference. Using just schemas, it is not easy to specify the requirement of well-formedness. Another example is in the case where we want the portions of offers in a particular vocabulary to be unique to the offer relative to the other offers in the same vocabulary. In general, one can envisage a user specified constraint on the vocabulary that enforces an arbitrary well formedness constraint on each offer

that is registered in the vocabulary. We should note that the additional constraints that optional in the definition of a vocabulary, and in general, any XML schema qualifies as a vocabulary definition. At the outset, early implementations may support schemas that are flat, that is, those that do not contain any nested schemas, but the specification allows for arbitrary schemas with additional constraints as the basis for defining vocabularies.

The constraints that can be defined in vocabularies come in two types:

- **Integrity constraints** that validate the description using this vocabulary internally.
- **External constraints** that are imposed by the vocabulary definer in order to ensure the proper interoperation of offer or services descriptions created with it and registered in it.

These constraints are also prevalent in traditional databases and are well accepted.

Constraint specification based on horn clauses

We can use any standard language for encoding the constraints, examples include KIF, horn clauses, etc. The DTD presented here encodes the constraints as a sequence atoms that have to evaluate to true given a sequence of rules. These rules can be interpreted as horn clauses for the purpose of evaluating whether the atoms are true. In addition, it allows the specification of certain well-defined constraints captured by the element wdConstraint. A more detailed discussion of the constraint language will be added.

```
<?xml version='1.0' encoding='us-ascii'?>
<!ELEMENT constraint (goal*, rule*)>
<!ELEMENT goal (atom | batom)>
<!ELEMENT rule (head, body)>
<!ELEMENT head (atom)>
<!ELEMENT body (batom*)>
<!ELEMENT atom (predicate, args)>
<!ELEMENT batom (atom | wdConstraint)>
<!ELEMENT wdConstraint (atom)>
<!ELEMENT args (term*)>
<!ELEMENT term (constant | complexTerm| variable)>
<!ELEMENT variable #PCDATA>
<!ELEMENT constant #PCDATA>
<!ELEMENT complexTerm (functor, args)>
<!ELEMENT functor #PCDATA>
<!ELEMENT predicate #PCDATA>
```

Constraint specification based on first-order logic:

An alternative way of expressing constraints in a vocabulary is by using first-order logic expressions. The following DTD for the extension of a schema by constraints represents the notion that the constraint is a list of first-order expressions. A more detailed discussion of the constraint language will be added.

```
<?xml version='1.0' encoding='us-ascii'?>
<!ELEMENT constraint (expression)>
<!ELEMENT expression (and | or | not | forall | exist | simplePredicate)>
<!ELEMENT and (expression*)>
<!ELEMENT or (expression*)>
<!ELEMENT not (expression)>
<!ELEMENT forall (iterator, iteratorQuery, expression)>
```

```
<!ELEMENT exist (iterator, iteratorQuery, expression)>
<!ELEMENT simplePredicate (name, (subject| combinedSubject) ,
            (object | combinedObject))>
<!ELEMENT combinedSubject (iterator, query)>
<!ELEMENT combinedObject (iterator, query)>
<!ELEMENT subject (iterator, query)>
<!ELEMENT object (iterator, query, constant)>
<!ELEMENT iterator #PCDATA>
<!ELEMENT query #PCDATA>
<!ELEMENT name #PCDATA>
<!ELEMENT constant #PCDATA>
```

5.2 Example of a Vocabulary

The following vocabulary defines the meta-data needed for describing and registering paper sellers:

```
<?xml version='1.0'?>
<schema targetNS="urn:schemas-paperXchange-com:paperOffer.xsd"
   xmlns="http://www.w3.org/1999/09/24-xmlschema-1/structures.xsd"
   paper:xmlns=”http://paper.foo.com/paper_terminology.xsd>
<element name=”PaperVocabulary”>
  <complexType content = "elementOnly">
   <element name="catalog-identifier" type=”string” required=”yes”/>
   <element name="supplier-name" type=”URL” required=”yes” isreference=”yes”/>
   <element name= "supplier-part-number" type=”integer”/>
   <element name=”product-type” type=”paper:prodType” />
   <element name= "paper-description" type=”paper-description-type”/>
   <element name= "paper-pricing" type=”paper-pricing-type”/>
   <element name= "classification" type=”classificationType”/>
  </complexType>
<element>
<constraint>
  <goal> <batom> <wdConstraint> <atom>
    <predicate>unique</predicate>
    <args> <term>
      <variable>$paper/catalog-identifier</variable>
    </term> </args>
  </atom> </wdConstraint> </batom> </goal>
</constraint>
<constraint>
  <goal> <batom> <wdConstraint> <atom>
    <predicate>private</predicate>
    <args>
       <term>
         <variable>$paper/paper-pricing</variable>
       </term>
    </args>
  </atom> </wdConstraint> </batom> </goal>
</constraint>

<complexType name='paper-description-type' content=”elementOnly”>
       <element name=”paper-desc” type=”string”/>
       <element name=”paper-size” type=”integer”/>
        <element name=”width” type=”integer”/>
       <element name=”length” type=”integer”/>
</complexType>
```

```
<complexType name='paper-pricing-type' content="elementOnly">
       <element name="list-price" />
       <element name="list-uom"/>
</complexType>

<simpleType name="classType" base = "string">
  <enumeration value="UN/SPSC"/>
  <enumeration value="NAICS"/>
</simpleType>

<complexType name="classificationType" content="empty">
  <attribute name="classificationType" type="classType"/>
</complextType>

</schema>
```

Essentially, the above schema defines the notion of a vocabulary for a paper seller. The name of the vocabulary is "paper", and it defines the structures required for registering a paper seller description. It specifies that the catalog identifier is required for any paper seller to be registered with the matchmaker and the constraint is that the catalog identifier has to be unique. The vocabulary also indicates that the paper descriptions that are registered are such that the width of the papers are greater than 6 units. It also says that the supplier name must be a reference to an entity that should point to an entity registered in the business entity vocabulary.

5.3 Service Descriptor

The purpose of a service descriptor is to provide some of the meta-data of the service. This meta-data provides information relating to the description of the service, some of the binding information that determines the transport level protocols that the service supports, some of the security related information needed to inter-operate with the service, etc. The service descriptor document uses some of the concepts that emerging standards such as UDDI use in order to describe services. For instance, the notion of business Service, binding Template, and tModels are used by the service descriptor document in order to express the meta-data of the service.

This section represents the current thinking of the service descriptor and may change before the final version of the SFS 2.0.

5.3.1 Elements of the Descriptor document

The service descriptor includes all parameters necessary for describing an e-speak service. It consists of the following six parts:

- **Service Information:** Includes details such as service name, version, description, and type
- **Service Provider Information:** Includes information about the business that is hosting the service.
- **Service conversation Definition:** Includes the definition of the conversation or a reference to the definition.

The following represents an example of a service descriptor document that captures some of this information.

```
<ServiceDescriptor xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="C:\sriram\sdlcdlpdl\SDLSchema.xsd">
  <ServiceProperty>
    <AuthInfo />
    <BusinessService serviceKey="serviceKey" businessKey="businessKey">
      <Name />
      <Description />
      <BindingTemplates>
         <BindingTemplate>
           <AccessPoint urlType="http" />
           <tModelInstanceDetails>
             <tModelInstanceInfo tModelKey="tModelKey" lang="cdl">
                <Description />
                <ConversationDefinition id="ConvDef1" href="#ConvDef1" />
              </tModelInstanceInfo>
           </tModelInstanceDetails>
         </BindingTemplate>
       </BindingTemplates>
       <CategoryBag>
         <KeyedReference tModelKey="" keyName="" keyValue="" />
       </CategoryBag>
    </BusinessService>
  </ServiceProperty>
  <ServiceVariable>
    <ServiceURL />
    <ServiceURI />
  </ServiceVariable>
  <ConversationDefinitions>
   <ConversationDefinition id="ConvDef1" href="http://www.convserv.com/CDL1.xml" />
   <ConversationDefinition id="ConvDef2" href="http://www.convserv.com/CDL2.xml" />
  </ConversationDefinitions>
</ServiceDescriptor>
```

5.4 Registering Vocabularies, Conversation Definitions and Services

Registration of vocabularies

Vocabularies are made available as generally accessable resources on the web, e.g. on the web sites of the standard bodies defining the vocabularies. In addition, all the vocabularies used for service or offer descriptions are registered with the service registry.

More details about vocabulary registration will be added.

Registration of conversation definitions

More details about conversation definition registration will be added.

Registration of services

In order to facilitate dynamic discovery, services are registered with service registries, also called match makers. In SFS, this is done by using offers and the match-making conversations. If a ser-

vice provider wants to make a service available for discovery, the provider creates an offer for the service and registers this offer with a match-maker. If a consumer needs a service, he either queries a match-maker for existing offers to sell services, or creates an offer to buy a service and registers this offer with the match-maker.

The content of an offer and the match-making conversations are specified in the match maker chapter of SFS 2.0.

5.5 Introspection Conversations

There are two conversations that allow services to get information about another service they are interested in. These two conversations support the dynamic interaction by allowing a functionality that is similar to the notion of introspection in traditional object systems. SFS defines two conversations for introspection. The ServicePropertyIntrospection conversation provides general information about the service, the conversation it supports, its provider, and how to access the service. The ServiceConversationIntrospection conversation returns the complete description of the conversations as CDL documents.

Every service in SFS has to provide these too conversations. For practical purposes, the server hosting the service might actually take care of these conversations - the service provider simply registers the service descriptor with the e-speak compliant server when deploying a service on it.

5.5.1 Roles and scenarios for the introspection conversations:

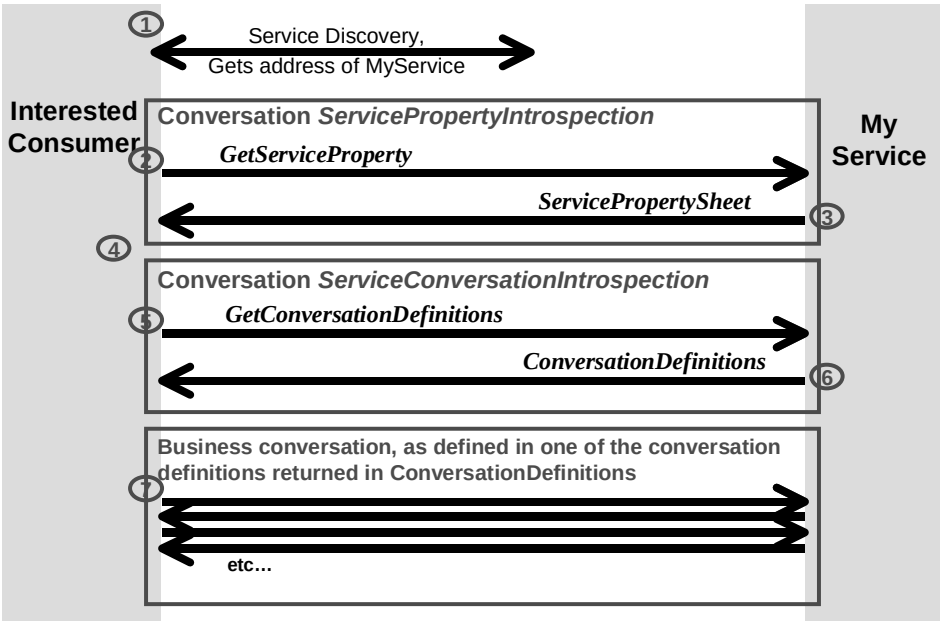


Figure 2: Introspection Conversations

During a typical introspection conversation between a client (service-consumer) and a service (service-provider), communication proceeds as follows (see also figure):

1. A client (interested consumer in above figure) discovers a service on the network.
2. It generates a service descriptor request message and passes it to the service. This message is an XML document that represents the client's request for the service descriptor.
3. The service receives this request from the client. It generates a service descriptor response message and passes it back to the client. This message is an XML document that contains all service descriptor information.
4. The client receives this response from the service. It examines the information contained in the descriptor and makes a decision as to whether or not it will use the service.
5. If the client decides to use the service, it generates a service conversation definition request message and passes it to the service. This message is an XML document that represents the client's request for the conversation definitions of the service.
6. The service receives this request from the client. It generates a service conversation definitions response message and passes it back to the client. This message is an XML document that contains all service conversation definition information.
7. The client receives this response from the client, and at this point may begin its interaction with the service.

5.5.2 ServiceDescriptorIntrospection Conversation

The ServiceDescriptorIntrospectio conversation is a very simple conversation. It just consists of one single ReceiveSend interaction. The interaction contains one possible inbound document and one possible outbound document:

- **GetServiceDescriptor**: requests information about the service
- **ServiceDescriptor**: returns the available information about the service as defined in the service descriptor specification.

The following CDL defines the **ServiceDescriptorIntrospection** from the point of view of the service that is being inquired:

```
<?xml version="1.0" ?>
<Conversation name="ServiceDescriptorIntrospection"
              xmlns=”http://www.e-speak.net/schema/conversation”
              initialInteraction=”#ServiceDescriptorIntrospection” >
   <ConversationInteractions>
      <Interaction id="ServiceDescriptorIntrospection"
             interactionType="ReceiveSend" >
         <InboundXMLDocuments>
            <InboundXMLDocument id="GetServiceDescriptor"
                          hrefSchema="GetServiceDescriptor.xsd" />
         </InboundXMLDocuments>
         <OutboundXMLDocuments>
            <OutboundXMLDocument id="ServiceDescriptor"
                          hrefSchema="ServiceDescriptor.xsd" />
         </OutboundXMLDocuments>
      </Interaction>
  </ConversationInteractions>
  <ConversationTransitions>
  </ConversationTransitions>
```

```
</Conversation>
```

The following is the schema of the **GetServiceDescriptor** document:

```
<?xml version="1.0" ?>
<schema name=”GetServiceDescriptor”
              targetNamespace="http://www.e-peak.net/schema/SFS”
              xmlns="http://www.w3.org/1999/XMLSchema"
              xmlns:sfsc=”http://www.e-speak.net/ServiceFrameWorkCommand>

   <element name=”GetServiceDescriptor” type=”sfsc:GetServiceDescriptorType”/>
   <!—Define empty GetServiceDescriptorType --!>
   <complexType name=” GetServiceDescriptorType” content=’empty’>
   </complexType>
</schema>
```

The following example shows the business payload of a GetServiceDescriptor message:

```
<SOAP-ENV:Body>
   <GetServiceDescriptor xmlns=”” >
   </GetServiceDescriptor>
</SOAP-ENV:Body>
```

The schema of the **ServiceDescriptorSheet** document has already been specified in section 5.3 “Service Descriptor“.

5.5.3 ServiceConversationIntrospection Conversation

The ServiceConversationIntrospection conversation is a very simple conversation. It just consists of one single ReceiveSend interaction. The interaction contains one possible inbound document and one possible outbound document:

- **GetConversationDefinitions**: requests detailed information about the conversations of the service
- **ConversationDefinitions**: returns the CDL specification of all the conversations supported. [1]

The following CDL defines the **ServiceConversationIntrospection** from the point of view of the service that is being inquired:

```
<?xml version="1.0" ?>
<Conversation name="ServiceConversationIntrospection"
              xmlns=”http://www.e-speak.net/schema/conversation”
              initialInteraction=”#ServiceConversationIntrospection” >
   <ConversationInteractions>
```

1. The ConversationDefinitions message returns all the CDL documents, yet it does not contain the schemas of the payload documents referenced in the CDL. The CDL document provides the URLs of the schemas.

```
        <Interaction id="ServiceConversationIntrospection"
               interactionType="ReceiveSend" >
            <InboundXMLDocuments>
               <InboundXMLDocument id="GetConversationDefinitions"
                                   hrefSchema="GetConversationDefinitions.xsd" />
            </InboundXMLDocuments>
            <OutboundXMLDocuments>
               <OutboundXMLDocument id="ConversationDefinitions"
                                   hrefSchema="ConversationDefinitions.xsd" />
            </OutboundXMLDocuments>
        </Interaction>
  </ConversationInteractions>
  <ConversationTransitions>
  </ConversationTransitions>
</Conversation>
```

The following is the schema of the **GetConversationDefinitions** document:

```
<?xml version="1.0" ?>
<schema name="GetConversationDefinitions"
              targetNamespace="http://www.e-peak.net/schema/SFS"
              xmlns="http://www.w3.org/1999/XMLSchema"
              xmlns:sfsc="http://www.e-speak.net/ServiceFrameWorkCommand>

    <element name="GetConversationDefinitions"
               type="sfsc:GetConversationDefinitions"/>
    <!—Define empty GetConversationDefinitionsType --!>
    <complexType name=" GetConversationDefinitionsType" content='empty'>
    </complexType>
</schema>
```

The following example shows the business payload of a GetConversationDefinitions message:

```
<SOAP-ENV:Body>
    <GetConversationDefinitions xmlns="" >
    </GetConversationDefinitions>
</SOAP-ENV:Body>
```

The following is the schema of the **ConversationDefinitions** document:

```
<?xml version="1.0" ?>
<schema name="ConversationDefinitions"
              targetNamespace="http://www.e-peak.net/schema/SFS"
              xmlns="http://www.w3.org/1999/XMLSchema"
              xmlns:conv="http://www.e-speak.net/schema/conversation>
    <element name="ConversationDefinitions"
               type="sfsc:ConversationDefinitionsType">
        <complexType name="ConversationDefinitionsType">
            <sequence>
               <element name="Conversations" minOccurs="1" maxOccurs="1">
                  <complexType>
                    <element name=ConversationDefinition
                               minOccurs="1"  maxOccurs="unbounded"
                               dt:schema="conv:ConversationDefinition"/>
                  </complexType>
```

```
            </element>
          </sequence>
        </complexType>
    </element>
</schema>
```

An example for the ConversationDefinitions document can be found in the appendix.

6 Match Maker Specification

6.1 Introduction

As mentioned earlier, SFS supports dynamic discovery of e-services. In this chapter we describe the SFS concepts related to registering and matchmaking, how service registries can support match-making, and the specification of the registration and match-making (lookup) conversations.

Matchmaking is the process of putting service providers and service consumers in contact with each other. The matchmaker is where services that want to be dynamically discovered register themselves, and where services that want to find other services send their request for matches. Some of the services advertising themselves through the matchmaker will be simple end-providers, while others may be brokers, auction houses and marketplaces which offer a locale for negotiating with and selecting among many potential providers. The matchmaker is a very simple, foundational, service on which the rest of the service framework rests, and should be as neutral as possible. It is the web-spider search engine of the e-services world. Other value-added services can take output from matchmakers to assist a service consumer in their selection of a business partner - they could provide recommendations over the set of results based on quality data, past performance metrics and prices, current state of the market etc -however, this is outside the scope of the SFS, but is instead a set of value-added trust services which can be built on top of it. For example, a service consumer wishing to place an order for DRAM may send a message to the matchmaker, and receive pointers to direct suppliers such as INTEL, an industry exchange such as E-HITEX and excess inventory auction sites such as fastparts.com.

Matchmaking is the process of connecting up the buyers with sellers, or service-consumers and providers. The **matchmaker** is where services that want to be dynamically discovered register themselves (i.e., their advertisements, or offers), and where services that want to dynamically find other services send their requests for matches. Essentially, the matchmaker itself is a service that participates in the services framework. Matchmaker and service registry are used as interchangeable terms within SFS.

Tasks of the matchmaker service

Service providers can register their offers, i.e. information about their services with the matchmaker. In addition to service provider data, matchmakers may contain registrations from service consumers who wish to be dynamically discovered by providers.

A matchmaker receives a lookup request containing a service description. The service description specifies the nature of the service the initiator wishes to trade and can potentially express constraints associated to each of the aspects of the description. The job of the matchmaker is to return:

- compatible service descriptions registered by other service providers or service consumer along with contact their information
- compatible agreement templates, each with associated negotiation locales (see chapter on negotiation)

- compatible contract template, each with associated market governance services (see chapter on contract)

The matchmaker returns all service descriptions that are compatible with the service description specified. Two service descriptions are compatible if they can refer to at least an instantiation of the service that is common to them.

As far as agreement templates and contract templates are concerned, the matchmaker returns all agreement templates and contract template that specify that they are able to reach agreements on the given service

Usage scenario of a match maker service

At the outset, we outline the high-level steps for dynamic discovery of services. It should be noted that these steps are meant as a guideline and are included here to provide a grounding for some of the concepts introduced in this chapter.

1. Service provider gets reference to a service registry acting as matchmaker.
2. Service provider optionally establishes a contract with matchmaker for hosting its offers.
3. Services are registered by their providers with a service registry using an offer that contains at least a short service description. This step uses the registration conversation.
4. If the service description uses a vocabulary not yet known to the service registry, the vocabulary also has to be registered with the service registry. This step uses the vocabulary creation conversation. However, we expect vocabularies to be created by standards bodies or by operators of matchmakers.
5. Consumers gets reference to a service registry acting as matchmaker.
6. Consumers optionally establishes a contract with matchmaker for hosting its offers.
7. Consumers request from the service registry a domain specific vocabulary, using the querying the matchmaker for vocabularies.
8. The consumer then queries the service registry for suitable services, the queries are formulated using the vocabulary. This step uses the the look-up conversation.
9. Optionally the consumer now introspects the services it has been referred to by the service registry, using the introspection conversations. Part of the introspection may be to receive the conversations supported by the service, in case the requied conversations have not been part of the conditions in the look-up.
10. Once the consumer has found a suitable service, it starts interacting with it using the conversations supported by the service.

On-line and off-line matchmaking:

Matchmakers interact with their clients in the following two manners: on-line and off-line. Matchmakers that support on-line interactions respond to queries in a synchronous manner. That is, when a client submits a matching request, the matchmaker returns the list of matched service providers immediately. However, a matchmaker that supports off-line interactions allows the query from the client to be executed in some time frame specified by the client. Such interactions are especially relevant in the case of complex matching mechanisms where matching is sophisticated. Furthermore, the off-line matching ability also allows the matchmaker to support discon-

nected and mobile clients. Off-line matchmakers provide mechanisms that allow entities to interact using event notification mechanisms. Essentially, one can define the notion of event vocabularies to categorise various kinds of events. Event subscribers subscribe to events by sending queries in a vocabulary to the matchmaker that further constrain the events. Event publishers publish events by sending offers registrations to the matchmaker that also conform to a vocabulary. Events and off-line matchmaking are not yet part of SFS 2.0, SFS 2.0 does not offer any predefined publish-subscribe service or messaging.

Simple Match Making

In simple matching, the eventual buyer or consumer drives the matching process. For example, in the case of a simple catalog management system each entry in the catalog can be treated as an offer to sell the good or service described in the catalog. The client who is interested in buying goods browses the catalog and picks a supplier. Another example is when the buyer initiates an offer to buy some good or service and providers of the good may bid on the offer and the buyer may choose amongst the bidders in some manner. Such a protocol is popular in a business to business scenario, where a company interested in purchasing some good creates a request for quotes (RFQ) that suppliers respond to. The above example assumes that the suppliers register their content with the matchmaker in order to make up the catalog.

This model of operation has challenges because suppliers have to update a catalog that is hosted by a matchmaker. If the suppliers want to maintain the portions of their offers under their control, they need only register their location with the matchmaker that can get the contents of their offers from the registered location. Such a mode of operation, where the catalog is hosted at the supplier is mandated when one uses a framework such as UDDI. On the other hand, there are situations where greater efficiencies are enabled by having the matchmaker host the offers on behalf of the clients. For instance, if the service provider and clients want to be mobile and still make use of the matchmaking facility, it may be more appropriate for the matchmaker to host the offers. In the rest of this section, we will not distinguish between the case where the matchmaker hosts the offers and the case where the offers are hosted by the offer creator unless specifically stated. We will assume that even if the offers are hosted by the offer creator, they conform to exactly the same schema as the case when the offers are hosted by the matchmaker. When the offers are hosted by the offer creator, the matchmaker acts like a search engine on the web today. However, on the other hand, when the offers are hosted by the matchmaker, it can support a additional features such as mobility, disconnected operation, etc.

6.2 Offers

6.2.1 Overview

The matchmaker can be viewed as managing **offers** that serve as the basis of communication between service providers and consumers. Services that want to dynamically interact with other services do so because they are interested in either providing services to, or consuming services from, other services in the economy. Offers themselves generally have two major flavors: **offers to buy** (i.e. to consume or use another service) and **offers to sell** (i.e. to be used by other services).

In addition, the offers are also classified according to the **vocabulary** that they conform to. The vocabulary provides a framework for specifying the metadata of services. The vocabulary mechanism allows offer creators to specify the information that they want to specify in their offers. Furthermore, the vocabularies also allow clients who are searching for offers to structure their queries in a manner so as to find the service of interest to them.

There may be other requirements from the matchmaker that affect the information that is conveyed in an offer or a lookup request. Some of these requirements are:

1. Security: The security mechanisms required by the matchmaker can be quite comprehensive. For instance, buyers who want to create offers to buy may want to restrict the visibility of this offer to buy to the list of preferred vendors. Furthermore, the security component enforces the rule that only the owner of any offer can change the contents of the offer if permitted by the terms and conditions between the matchmaker and the service provider. The creator of the offer defines the security rules to be used when matching this offer against a query issued by a client.

2. Matchmaker Business Rules: The matchmaker provides a service to both service providers (by hosting their offers), and clients (by searching for their requirements). Therefore, it may bill its clients for the services provided. The billing rules can be quite sophisticated as the matchmaker may charge its clients in many different ways. For example, the matchmaker may charge the service providers a fee depending on how many times the service provider's offer was forwarded as a potential match to a client's request. In some other deployments, the matchmaker may choose to not bill for hosting ads or performing queries. In addition to billing, the contract between the creator of an offer and the matchmaker can stipulate the length of time the offer is going to be hosted by the matchmaker. The offer that is created in the matchmaker can optionally refer to a business contract between the creator of the offer and the matchmaker that outlines the terms and conditions for hosting the offer at the matchmaker. These business rules may also apply to queries that are sent to the matchmaker especially if the queries are off-line queries.

3. Offer Owner Reference: When a client requests a match from the matchmaker, it expects in return a reference to the service provider that created the offer. If the matchmaker is part of a financial intermediary, the financial intermediary may present itself as the owner of all the offers that are available on it.

4. Query Owner Reference: The matchmaker can require query owners to include a reference to themselves in the queries that they send to the matchmaker. This is required for the off-line operation of the matchmaker.

5. Offer Owner Business Rules: Matchmakers can allow creators of offers some control over how their offer is to be managed by the matchmaker. For instance, suppose enterprise A creates an offer to sell pencils. Furthermore, suppose another enterprise searches for pencil providers in order to buy pencils worth $100,000 (some large amount). Enterprise A can ask the matchmaker to send a message to enterprise A if a potential buyer is interested in pencils in either a large enough quantity or dollar value. Essentially, these rules can be thought of as being a generalized event subscription rules. Another use of these business rules is to incorporate inventory, pricing, and QOS parameters into the offer.

6. Query Owner Business Rules: Just as creators of offers can associate business rules with their offers, query owners can also associate business rules with their queries that can trigger messages to the client when some condition is satisfied. These are particularly relevant when issues such as inventory, price, and QOS parameters are parts of the query.

7. Offer Owner Preferences: The creator of the offer can choose amongst alternative clients when certain conditions are met. For instance, if the offer is an offer to sell steel, and the supplier has only 100 tons of steel, and two clients want to buy 75 tons of steel each, the supplier has to choose whose query she would rather fulfill as she cannot fulfill both. These preferences associated with offers allow the creator of the offer to specify the rules for selecting one client over another in case of conflict.

8. Query Owner Preferences: The initiator of a query may also specify preferences that she wants satisfied. These preferences provide a way for the client to choose amongst alternative providers whose offers match her query based on criteria that she defines.

9. Owner interfaces: Each offer also has a reference to the list of interfaces that the owner of the offer supports in relation to this interface offer.

6.2.2 Detailed Structure of Offers

In this section we define the various elements of the XML document for an offer. In the description we mostly refer to offers to sell, yet the same elements need to be specified when creating offers to buy. The elements are grouped into *public elements* and *private elements*. Private elements contain information only relevant to the matchmaker service, this information is never sent to the consumer, and naturally, can never be searched by the consumer. Public elements contain the information available to the consumer.

The matchmaker associates a unique identifier with each offer that it hosts. This identifier can be requested by the clients when looking up offers and be used to refer to them later. Also, the matchmaker may pass along these identifiers to other parties when it wants to provide them with a reference to the offer.

In the following description we have grouped the elements into general information about an offer, information about provider, and special information and rules used for matchmaking.

General information about the offer:

Tag Name	Required?	Occurs	Description	Semantics
`<offer-description>`	Yes Public Searchable	Once	Describes the offer	Public searchable attributes of offer

<offer-info>	No Public, Non-searchable	Once	Other public info associated with offer	Public non-searchable attributes of offer
<offerID>	No, Public, Searchable	Once	The unique id associated with offer by matchmaker	
<offer-type>	No Public, Non-searchable	Once	The type of offer	This captures the kind of auction/exchange bid this offer is.
<time-of-creation>	No	Once	Time stamp of creation of offer	
<offer-validity>	No	Once	Time line for which offer is valid	

The **offer-description** element contains the describes the service to which this offer belongs. The service description uses the attributes defined in vocabularies, an offer might use several vocabularies at the same time. The content of the offer-description element can therefore be any XML document that conforms to the selected vocabularies. The offer-description element is one of the basic elements used in the lookup queries.

The service description in the offer can be compared to the service description in the service descriptor. They are both based on vocabularies, and they both serve the same purpose, to give further information about a service. Yet they might not be the same, they might have different attributes, there content may overlap or one description may be a subset of the other one, or they may even use different vocabularies.

The **offer-info** element contains other public information that clients who find the offer can peruse. For instance, in the case of a catalog, the public information can contain pictures of the item that is being sold.

The **offerID** element is associated with the offer by the matchmaker. It is not specified by the creator. However, for the consumer it looks like any other public searchable element of the offer.

The element **offer-type** determines the type of offer. For instance, if this offer is an offer to sell in an auction, additional information about the auction can be specified here. Clients who discover the offer can inspect this element to determine the kind of offer.

The **time-of-creation** element contains a timestampt giving the time the offer was registered with the service registry.

The **offer validity** element determines the period of time for which the offer is valid. If unspecified, the offer is assumed to be valid until it is explicitly removed by the creator of the offer. Clients who discover this offer can inspect this field in order to determine the length of time for which the offer is valid.

Interface and provider information::

Tag Name	Required?	Occurs	Description	Semantics
<owner>	Yes Public, Non-searchable	Once	Owner of offer.	Reference to owner, can be url, etc.
<offer-interfaces>	Yes Public, Non-searchable	Once	Conversations supported by the service advertised in the offer	List of conversation names
<matchmaker-contract>	No, Private	Once	Contract between matchmaker and creator of offer	This contract contains the terms and conditions for hosting the offer.
<**private-info**>	No Private	Once	Private info associated with offer	Private info that the creator can associate with this offer.

The **owner** field provides a reference to the creator of the offer, i.e. the provider of the service. It may, for example, point to a URL of the creator. This element is also a public element, but is not searchable. It is a required element within each offer, because the match-maker typically returns the contents of this element within an offer as the result of a lookup.

The **offer-interfaces** element lists the conversation supported by the service.

The **matchmaker contract** element contains a reference to the contract between the creator of the offer and the matchmaker. This element is also optional. It is likely to be present in complex matchmakers and likely to be absent in simple matchmakers.

The **private-info** element contains information that is private to the owner of the offer and is associated with the offer. The creator of the offer can interpret this in any way she pleases. This information is typically in addition to the security information and is optional. Other clients who find this offer do not get to see this information.

Special elements for match-making

Currently, the concrete syntax (schemas) for some of these elements is not defined by SFS. It is up to the matchmaking service to define which syntax it wants to use for these fields. It is assumed that a matchmaking service might provide more than one possible syntax for some of the fields, and the service provider can use the once most appropriate for his kind of offer.

Also some of these elements are dynamic in nature, i.e. they need to get updated very frequently. There are three possible ways to do this:

- *Modify-offer conversation*: whenever the element changes, the provider notifies the matchmaker using the modify-offer conversation.
- *Events*: whenever the element changes, the provider notifies the machmaker by sending an event over a publish-subscribe mechanism (currently not supported in SFS)
- *Callbacks*: whenever the machmaker needs to know the value of the element, it asks the provider for the up-to-date value using a conversation predefined by the matchmaker. SFS currently does not define this conversation, nor how this conversation is described in the offer. It is up to the matchmaking service to specify the mechanisms it wants to use.

Tag Name	Required?	Occurs	Description	Semantics
<availability>	No	Once	Availability criteria	Rules that determine availability criteria.
<pricing>	No	Once	Pricing criteria	Pricing rules associated with this offer.
<fulfillment>	No	Once	Fulfillment rules	Has criterion for fulfillment
<owner-rules>	No, Private	Once	Rules that the creator associates with the offer	The matchmaker enforces these rules
<contract-templates>	No, Public	Once	Templates of potential contracts that can be reached with owner of offer	These are templates of contracts that can be reached amongst the e-services in eco-system.
<agreement-templates>	No, Public	Once	Agreement templates that govern this offer	These templates form the basis of the start of negotiations
<owner-preferences>	No Private	Once	Preferences of the owner	These are rules that specify conflict resolution strategies.

The **availability** element of the offer determines the mechanism by which the owner of the offer determines the current availability of the service or good being offered for sale. In the case of goods, the availability refers to the inventory. However, in the case of services, the availability can refer to the current load and incorporate QOS parameters. Service providers can register callbacks that the matchmaker can invoke when client requests depend on the availability.

The **pricing** element of the offer determines the mechanism by which the owner of the offer determines the current price of the service or good being sold. The service provider can associate interaction with its pricing module in order to determine the price. For instance, if this were an offer to sell airline tickets, the pricing can depend on when the airline ticket is being purchased.

The **fullfillment** element describes when and how this offer is going to be fulfilled (especially useful if physical goods are going to be shipped). It may, for example, have a link to the owner that can determine when the owner can fulfill the request.

The **owner-rules** element contains, among other things, the security information associated with this offer. The security element contains the visibility rules that the creator of the offer wants to associate with the offer to sell. For instance, the creator of the offer may want to make the offer visible to clients with a credit rating of 'B' or better, or HP employees. If a service provider does not associate any security information with an offer, it is assumed that there are no security restrictions, and the offer can be made visible to any client who sends a lookup request that matches the offer.

In addition, the creator of the offer can associate other business rules with the offer. For instance, these could specify additional conditions required by the owner for a match, or could specify conditions under which some message is sent to the owner of the offer. These rules are especially useful in complex matchmaking. For instance, in an auctioning engine, the rules could specify some of the rules that govern the auction.

The **contract-templates** element contains the templates of contracts that can be reached amongst the e-services that are participating in the eco-system for which the matchmaker is a registry. The actual contents of this element can contain references to the actual contract template documents if they are registered with the matchmaker. The reader is referred to the section on the contract formation and maintenance in SFS part II for details of the contract template.

The **agreement-templates** element contains various agreement templates that can be used by the parties in order to negotiate terms and conditions of their interaction. This specifies the different parameters of the negotiation, e.g., product type, price, supply date etc. Some of the parameters will be constrained within the template while others may be completely open. The agreement template is decided at the matchmaking stage - matchmaking is the process by which one party locates other parties with agreement templates compatible with their needs.

The **owner preferences** provides rules for conflict resolution. Suppose for instance that two clients want to buy some quantity of some good that cannot be simultaneously satisfied by the service provider, the owner preferences can dictate which client's lookup request the service provider would rather satisfy. Again, this element is more likely to be used in a complex matchmaker such as a commodity exchange.

6.3 Match Making Conversations

The following conversation definitions only support one vocabulary per offer. All of the conversations specified below consist of exactly one ReceiveSend interaction. We have therefore ommitted the various CDL definitions for the conversations.

6.3.1 Creating a business relationship with a matchmaker

The business relationship with a matchmaker is established using the negotiation and contract formation framework discussed in part II of the SFS. A service that needs to register at a matchmaker either has a reference to a matchmaker, or, it can search for a matchmaker at other matchmakers. The recursion ends at a root matchmaker such as HPs developer village hub.

6.3.2 Registration Conversations

There are three registration conversations for offers which all consist of one ReceiveSend intraction with exactly one possible inbound and one possible outbound document:

- **Register an Offer**:
  - inbound document: **offer**
  - outbount document: **offer-reply**
- **Modify an Offer**:
  - inbound document: **offer-modify**
  - outbount document: **offer-reply**
- **Retract an Offer**:
  - inbound document: **offer-retract**
  - outbount document: **offer-reply**

In the following sections we specify the XML Schemas of these messages.

XML document: Register an Offer

Services register offers with matchmakers using the by sending the matchmaker offer documents. The schema for the offers looks as follows:

The top level schema for the offer registration message looks as follows (for further details see previous chapter on offers):

```
<element name="offer" >
  <complexType content=”elementOnly” model=”open”>
   <element name="offer-description"/>
   <element name="offer-info"/>
   <element name="owner"/>
   <element name="owner-interfaces"/>
   <element name="availability"/>
   <element name="pricing"/>
   <element name="fulfillment"/>
   <element name="private-info"/>
   <element name="matchmaker-contract"/>
   <element name=”matchmaker-rules”/>
   <element name="owner-rules"/>
   <element name =”contract-templates” />
   <element name=”agreement-templates” />
   <element name="owner-preferences"/>
   <element name="time-of-creation" />
   <element name="validity"/>
   <element name="offerID"/>
   <element name="offerType"/>
  </complextype>
</element>
```

The precise contents of the some of the elements such as the availability and fulfillment tags can be determined by the matchmaking service. In the description element of an offer, one can include any XML document that conforms to a particular vocabulary. The schemas used in the descriptions are found in schema repositories or in the matchmaker. If there are multiple XML documents that correspond to different vocabularies, the service offer will be registered in all the vocabularies, and any client will be able to search for the service in any of those vocabularies.

XML document:Modify Offer

The modify offer request modifies an existing offer registered with a matchmaker. Clients may use this message to modify any aspect of the offer that is allowed according to the terms and conditions reached between the client and the matchmaker. In essence, the offer modify message sends another offer to the matchmaker to replace the existing offer. The result of sending an offer-modify message is another offer-reply message . In essence, the offer-modify message has the following schema.

```
<element name="offer-modify">
 <complexType content=”elementOnly” model=”open”>
     <element name="offer"/>
     <element name="offerID"/>
  </complexType>
</element>
```

XML document:Retract Offer

The offer-retract document can be sent by the creator of the offer in order to retract a previously registered offer. The structure of this document looks as follows:

```
<element name="offer-retract" >
 <complexType content=”elementOnly” model=”open”>
     <element name="offerID"/>
  </complexType>
</element>
```

Both the offer-retract and the offer-modify messages receive an offer-reply document as a reply that indicates whether the message was successfully processed.

XML document: Reply to an offer

The matchmaker replies to registration requests with an offer-reply document. This offer-reply document typically contains an offerID that the client who created the offer can use in order to make other changes to the offer.

```
<element name="offer-reply">
 <complexType content=”elementOnly” model=”open”>
    <element name="status"/>
     <element name="offerID"/>
  </complexType>
</element>
```

The status element determines, for instance whether offer was accepted by the matchmaker. The matchmaker can reject offers for a variety of reasons, some of which are:

- The offer document is not well formed. For example, some of the required elements may be missing, etc.
- The offer document does not comply with the contract under which it has been sent.

The offerID can be used by the client for subsequent requests to the matchmaker for modifying the contents of an offer.

6.3.3 Lookup Conversation and Queries

The **lookup conversation** consists of only one ReceiveSend intraction with extacly one inbound document (**lookup-request**) and one outbound document (**lookup-reply**).

Structure of Lookup Request

Lookup requests to the matchmaker have the following parts:

1. The constraints and preferences that make up the query (given by the vocabulary and by the various searchable elements of the offer schema).
2. The contract with the matchmaker under which the query is being executed, especially if this is an off-line query.
3. The business rules of the client issuing the query.
4. The reference to the client making the lookup request.

The various elements in the lookup request are:

Tag Name	Required?	Occurs	Description	Semantics
<**query**>	Yes	Once	The query	Contains the constraints and preferences of user.
<matchmakercontract>	Yes	Once	Contract with matchmaker	
<owner>	Yes	Once	The owner of the query	
<owner-rules>	No	Once	Rules associated with this query	
<validity>	No	Once	The time period for which the query is valid.	
<notification-method>	No	Once	The nofitication mechanism	

The **query** element represents the constraints and preferences of the query that the client uses to find offers. This essentially contains a document that conforms to the ESXQL schema [1]. In addition, it may have certain elements such as quantity, that represents the number of items being ordered. There can be a sequence of constraint elements that make up the query, and these can represent a more complex query that can be used in the context of combinatorial matching, etc. The query can also specify preferences that further refines the query by specifying additional requirements that offer or the owner of the offer may have. This allows clients to specify, for example, conditions such as given two potential offers, the client would rather consider the offer that has earlier shipping dates. Note that this is not just a min or max operation. Since this element represents the query itself, it is required.

The **matchmakercontract** element represents the contract that the client has with the matchmaker. For example, this allows the matchmaker to bill for the services it provides to the clients.

The **owner** element provides a reference to the query-owner so that the matchmaker can reply to the lookup request at a subsequent time, or forward the reference to service providers.

The **owner-rules** element provides a mechanism for clients to associate other rules that can affect the outcome of potential matches. For example, they can associate conditional callbacks that allow the clients to verify the contents of the offer before accepting it as a potential match.

If the query is an off-line query, the client can specify the time-line associated with the reply in the **validity** element.

If the query is off-line, the **notification-method** element has the notification method for notifying the owner. This allows the owner of the query to get off-band notifications such as e-mail, cell-phone calls, etc.

Query element of the lookup-request

The schema for the query element looks as follows:

```
<?xml version="1.0"?>
<schema xmlns='http://www.w3.org/1999/XMLSchema'
        xmlns:ES-CORE='http://www.e-speak.net/Schema/core/'
        targetNamespace='http://www.e-speak.net/Schema/core/'>
  <element name='query' minOccurs='0' maxOccurs='1'>
    <complexType>
    <!--the vocabulaies used in the query, if not present, use the Base Vocabulary-->
     <element name='vocabulary' minOccurs='0' maxOccurs='unbounded' content='empty'>
        <complexType>
          <!-- the prefix. If not presented, the current vocab is the default vocab-
              ulary -->
          <attribute name='prefix' type='string' use='optional'/>
          <!-- the name (e.g. URL) of the vocabulary -->
          <attribute name='src' type='string'/>
   </complexType>
  </element>

  <!-- result specifies the output of the query -->
  <element name='result' minOccurs='0' maxOccurs='1'> </element>
```

```
  <!-- the constraint -->
  <element name='where' type='constraintType' minOccurs='1' maxOccurs='1'/>
  <complexType name='constaintType'>
    <element name='condition' type='string'/>
  </complexType>
</schema>
```

The essential parts of the schema for queries above are the following:

- The vocabularies that define the definitions of the attribute names in descriptions against with the query is run against.
- The result that is expected from the query. Essentially, the result element allows the searchers to define the part of the offer that they want returned as a result of executing the query.
- The where element that captures the constraint that the results to the query satisfy.

In essence, the query element defined here is similar to standard query languages such as SQL, XQL, etc. In essence, the vocabulary definitions are akin to table definitions and the list of vocabularies are similar to the **from** clause in SQL queries. The result element of a query is similar to the **select** clause in SQL queries, and the where element is similar to the **where** clause in SQL queries.

The following is an example of a query for a service that offers letter size paper that costs less than 25 US dollars per unit.

```
<lookup>
<query xmlns="http://www.e-speak.net/Schema/core" >
   <vocabulary prefix="pv" name="paper-vocab"/>
<result>$offer/owner</result>
   <where>
       <condition>
           <and>
          <and>
             <equals>
                <left>pv:paper-description/paper-size</left>
                <right>&apos;letter&apos;</right>
             </equals>
             <lessThan>
             <left>pv:paper-description/paper-price/list-price</left>
             <right>25</right>
             <lessThan>
          </and>
          <equals>
          <left>pv:paper-description/paper-price/currency</left>
          <right>&apos;usd&apos;</right>
          </equals>
        </condition>
    </where>
</query>
<owner></owner>
<validity></validity>
</lookup>
```

Structure of Lookup Reply

The reply to a lookup request depends on the lookup request. In a generic lookup request, the client typically wants references to the service providers that match the constraints and preferences expressed in the lookup request. If the lookup request is an on-line lookup request, the reply will contain the result of the lookup. However, if the lookup request is an off-line lookup request, the reply will contain a lookupID that can be used by the client to subsequently determine the status of the lookup request.

```
<element name="lookup-reply">
 <complexType content=”elementOnly” model=”open”>
   <element name="result"/>
   <element name="lookupID"/>
  </complexType>
</element>
```

For instance, the lookup request asked for the references to the owners to the offers that matched the request for paper. Therefore, the reply from the matchmaker may look as follows:

```
<lookup-reply>
     <result>
   <esurl>es://server.acme-paper.com:8080/paperseller</esurl>
      <url>http://server.roadrunner.com:80/ps</url>
     </result>
</lookup-reply>
```

The client can inspect any aspect of the offer that is public.

7 Transactions

In this chapter, we introduce the problem of transactions over the open internet. The transaction problem for web services should first be limited to providing atomicity and not the other properties that transactionality in traditional distributed applications typically provide.

7.1 Introduction

The problem is to ensure two-party as well as multi-party atomicity on the Internet. The parties involved are e-services that may be owned by different organizations and composed dynamically. The challenge is to implement activities (or transactions) that span multiple e-services and still be atomic (either they happen entirely or not at all). As an example, consider the following three parties: a manufacturer, a supplier, and a shipment service. The manufacturer orders parts at the supplier and then organizes for these parts to be shipped by the shipment service. Figure 1 illustrates this scenario.

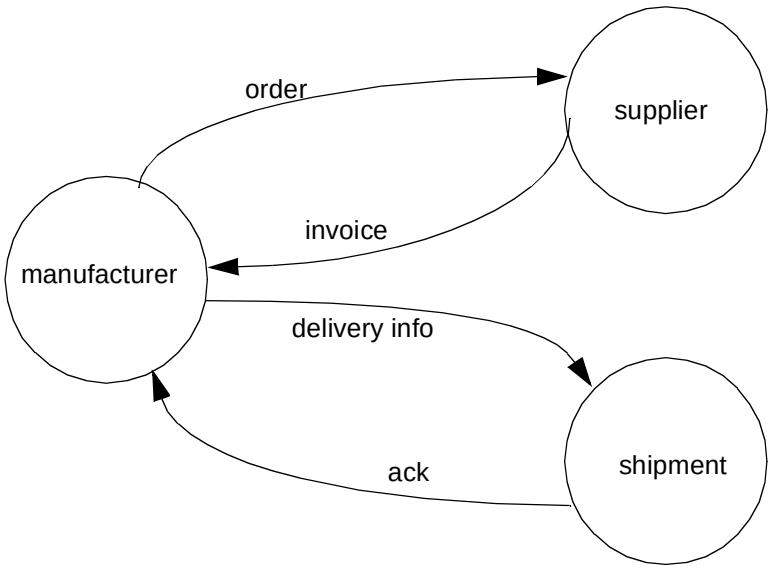


Figure 1. Three-party interaction between e-services

The manufacturer only wants to order the parts if shipment is possible, and vice versa. That is, the total order-shipment activity should be atomic. The atomicity should protect against failures (an e-service goes down) as well as user-level issues (shipping is impossible).

Each service typically has its own notion of transaction. These local (or internal) transactions update the data within a service. We can view the support for transactions as a composition mechanism for these local transactions to create a global (or public) transaction that have these local transactions as constituent parts. Traditional transactions usually satisfy consistency, isolation, and durability in addition to atomicity (ACID transactions). Initially, we are primarily concerned with providing the atomicity property for global transactions. We discuss two different ways to achieve global atomicity from local atomicity: two-phase commit and compensation. We also discuss ways to express the underlying protocols in terms of conversations. Using conversations allows us to make the protocols explicit rather than hard-wired, and we can (potentially) have multiple protocols co-exists in a system of e-services.

More specifically, the document is organized as follows. We first discuss the basics of two-phase commit and compensation. Then we touch upon various issues that arise when applying either two-phase commit or compensation in an Internet context. Finally we describe how two-phase commit and conversations play together with conversations and CDL.

7.2 Two-Phase Commit and XA

Overview

The standard solution to the atomicity problem, with multiple transaction participants, is the two-phase commit protocol. This is a two-round protocol in which a transaction coordinator takes votes among all transaction participants. If all participants vote yes, and therefore promise that they will be able to commit (i.e. make durable) the transaction results, the coordinator determines the transaction's outcome to be *commit*, and tells all participants to commit the transaction (this is the second round). If a participant votes no, or if the coordinator suspects a participant to have crashed, the coordinator determines the outcome to be *abort*, and informs all participants to abort the transaction. An example for a two-phase commit protocol is XA[1], which is the standard interface that transaction participants, such as databases and message queues, implement in tightly coupled object systems in order to participate in two-phase commit protocols.

Discussion

Two-phase commit and XA are standard means to achieve multi-party atomicity within an enterprise. The near universal support for XA makes it attractive to leverage XA into any standardization effort. The transaction participants in a global transaction would then be the per-service transaction coordinators who use XA to control local, intra-service transactions. The resulting (global) protocol would be a hierarchical two-phase commit protocol, where a (global) transaction coordinator would control a number of sub-ordinate transaction coordinators. This notion of hierarchical two-phase commit is at the core of the TIP protocol [2].

The main problem with using two-phase commit and XA on a global scale is that transaction participants (services) must hold locks on a transaction's data until the transaction terminates (abort or commit). Since the participants are independent services, the running time of a transaction is unpredictable and potentially long. Thus, a local transaction within a service may hold locks on data on behalf of a long-running global transaction. Because locked data cannot be accessed by other transactions,[3] holding locks for long-running transactions is usually not a good idea since it hampers concurrency and scalability.

Global transactions may be long-running transactions for several reasons:

---

1. Distributed Transaction Processing: The XA Specification, X/Open Snapshot, 1991. XA also supports one-phase commit, but here we only consider two-phase commit.
2. K. Evans, J. Lyon, and J. Klein, "Transaction Internet Protocol, Version 3.0," *Network Working Group RFC 2371*, The Internet Society 1998.

3.The actual rules for data sharing in the presence of locking depends on the isolation level. For example, it is usually possible to share read-only data (data held by read locks) between transactions.

- The completion time for a transaction is limited by the slowest participant. The running time of individual participants is less predictable on the Internet than within an enterprise.
- Transactions may now span entire conversations between distributed parties. For example, a multi-service transaction on the Internet may span an entire order-fulfillment process.
- Timeouts for failure-detection on the Internet typically have to be longer because of the unpredictable performance of the underlying network. Thus, it may take longer for a transaction to be rolled back if one of the participants fail.
- Finally, the two-phase commit protocol itself (when using XA) requires two round-trip messages with each participant. The latency of these interactions may be significant on the Internet.

The XA interface provides the option for transaction participants to unilaterally terminate transactions. That is, if a transaction participant is in its uncertainty period because it has voted yes to a transaction, it may decide by itself to either commit or abort the transaction as long as it remembers what it decided. The ability to decide unilaterally is provided so that transaction participants do not remain blocked if the transaction coordinator crashes and does not recover. One question is whether we want to allow unilateral decisions. Furthermore, if we choose to allow such decisions, should the circumstances be the same as in XA?

Another troubling aspect of using two-phase commit for the Internet is its failure in the past. In particular, TIP has been proposed 2 or more years ago, but has not seen wide spread adoptation (as far as we know). Is this simply because TIP was ahead of its time, or were there fundamental problems with TIP.

7.3 Compensation

Overview

The standard solution to ensuring atomicity for long-running transactions is to use compensating actions[1]. A compensating action is a piece of user-defined logic that cancels the effects of a previously committed transaction. For example, if the transaction transferred $34 between two bank accounts, the compensating action would be to transfer $34 between the same two accounts, but in the reverse direction. Compensating actions have to be user-defined because the state of the system may have changed since the original transaction was committed.

We can use the basic concept of compensation for long-running transactions. The idea is to break the long-running transaction into a number of sub-transactions. These sub-transactions are sometimes called (transactional) steps. Each step now has a compensating action. With this setup, we can commit each step as soon as it has completed, we do not have to wait for the global, long-running transaction to commit. If we cannot execute step *n*, we simply compensate every step up to *n* to cancel out the side-effect of the transaction so far.

Discussion

1. Y. Breitbart, H. Garcia-Molina, and A. Silberschatz, “Overview of Multi-Database Transaction Management,” *VLDB Journal*, Volume 1, Number 2, October 1992.

Traditional transactions hold locks on data until they terminate. This allows system-level cancellation of the transaction's update: we can cancel a transaction by re-establishing the values that existed before the transaction executed. With compensation, we cancel the effect of a transaction after the locks have been released, which means that another transaction may have seen the effect that is been cancelled. Thus, with compensating actions, we cannot cancel effects by re-establishing the pre-transaction data values. Instead, we have to explicitly program an "inverse" transaction. Furthermore, since we cancel updates that may have been seen by other transactions, we obtain a weaker notion of atomicity with compensating actions. This notion is called "semantic atomicity."

Nevertheless, the notion of compensation is actually alive and well on the Internet today. It is used primarily in the B2C space, where customers can submit an order and then later cancel it within a given time window. What likely goes on within such a B2C site is that an order is placed in a shipping queue and is not actually shipped for a certain period of time. Compensation now amounts to removing the order from the queue (and adding the items back to the inventory database).

The primary drawback of compensating actions is that they are user-defined. Thus, it is up to the programmer to verify that a given action in fact compensates for another action. Another drawback is that we may expose an update that is later rolled back (compensated). For example, if a customer books the last seat in a flight, and then later compensates (cancels the reservation), another customer may see a full flight, which turns out to be a transient state. Although this is a drawback from an ACID transaction point of view, travel reservation systems usually have this property.

As is usually the case with B2C transactions, the compensating actions most likely will have an expiration time. That is, the option of cancellation is only possible for a given period of time after the original transaction committed. There are at least a couple of reasons for having expiration times:

- The ability to compensate may be due to "delayed" server-side effect rather than truly compensatable server-side effect. For example, if the transaction orders an item that is then shipped to the customer, we can only delay the shipment (at least it is complicated to cancel a shipment once UPS takes over).
- Compensation may require a service to store some information that allows it to compensate. For example, if we are compensating a non-deterministic action, the service has to remember the non-deterministic choices made during the original transaction. We do not want the service to store such information forever.

The compensation period may even be a QoS aspect of services where some clients negotiate longer expiration times because they are willing to pay more, or because they are preferred clients for some other reason.

Finally, we notice that compensating actions have been utilized in some environments, such as workflow, that have some similarity to the Internet-based e-services scenario. In particular, workflow systems must deal with long running actions that make holding resource locks infeasible similar to what we described with the arguments against two-phase commit previously. Further, most approaches to e-services development provide more semantic information about the services, such as how they are invoked and the sequence of operations expected by the services (via conversation descriptions or service "choreographies"). This more detailed semantic information may help to make implementation of compensating actions easier and more reliable.

7.4 Internet Issues

Security

Both two-phase commit and compensation assumes that the various parties are well-behaved (or trusted). For example, two-phase commit assumes that participants vote "honestly" and that they do as instructed (commit or abort). Furthermore, the notion of compensation also assumes that a participant actually executes a compensating action if instructed to do so. With two-phase commit, each participant also trusts the coordinator to be in control of the protocol—the protocol is inherently asymmetric because the coordinator knows the outcome before any of the participants.

The issues of trust are magnified when we deploy the atomicity protocols on the Internet. The protocols concerned with fair exchange[1] may provide inspiration for tackling some of the issues, but at this point we merely identify the issue rather than propose solutions.

Recovery Assumptions

For two-phase commit and compensation protocols to ensure atomicity, some entities in these protocols need to make assumptions about the recovery of other entities. For example, a participant in two-phase commit will have to make assumptions about the recovery of the coordinator—if the participant suspects the coordinator to have crashed, it must either wait for the coordinator to recover or it must unilaterally decide an outcome for the transaction. In a tightly coupled system, where all entities are within the same domain of control, the recovery assumptions can remain implicit parts of the system and be hard-wired policies. In an open Internet-based environment, we need to be more explicit about such assumptions.

Notice that recovery assumptions also affect global atomicity with compensation. For example, consider a client that starts a global transaction, and crashes after executing half of the steps. To ensure atomicity, we need to assume that the client recovers and compensates for the executed steps.

Composition Issues

If we have nesting of services, which we are very likely to have, a tree structure will be a natural way to organize the resulting transactions. Each node in the tree will correspond to a local transaction within a service, and the global transaction is to provide atomicity over the entire tree. The previous discussion applies to homogeneous trees where all parent-child relationships are based on either two-phase commit or compensation. One question is what happens when we combine the two ways to implement global atomicity.

For simplicity, but without loss of generality, let us consider a two-level, heterogeneous tree. We consider two types of trees in Figure 2. The first scenario is two-phase commit (2pc) with compensation. The service `a` exports 2pc as its notion of atomicity. In turn, `a` uses services `b` and `c`, and `b` exports compensation and `c` exports 2pc. This type of composition seems to be possible. If `a` is asked to vote, it passes on `c`'s vote if `b`'s actions completed successfully, and votes no

1. J. Garay amd P. MacKenzie, "Abuse-Free Multi-Party Contract Signing," *Distributed Computing (DISC)* 1999, LNCS 1693.' S. P. Ketchpel and H. Garcia-Molina, "Making Trust Explicit in Distributed Commerce Transactions," *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1996.

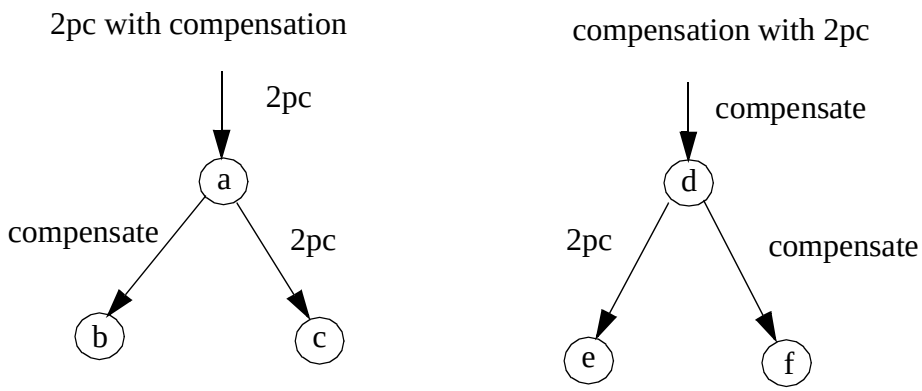


Figure 2. Composition of atomic actions

otherwise. If the outcome is abort, `a` compensates the action at `b` and propagates the outcome to `c`. Of course this assumes that the ability to compensate `b` has not expired when `a` is instructed to abort.

Consider now the second scenario in Figure 2. A service `d` exports compensation, and relies on compensation from `f` and 2pc from `e`. The question is: when should `e` commit its action. Ideally, we would like to commit `e`'s action when the global transaction is complete. However, since the compensation approach is inherently optimistic, `d` will not know when the global transaction is complete, it will only know if it needs to compensate its action. Thus, `e` cannot commit its action until `d` knows that it no longer needs to be able to compensate its action. That is, for this case to work, we need expiration times on the ability to compensate, and we need to hold the locks at `e` for that entire period.

If the ability to compensate is unbounded in time (there is no expiration time), we can implement 2pc on top of compensation (the left-hand side of the figure). If, on the other hand, compensation is time bounded, we can implement compensation in terms of 2pc (the right-hand side of the figure).

7.5 Conversations

We want support for transactions in the context of CDL conversations. Since services use CDL specifications to export information about their behavior, we would like CDL specifications to reflect the atomicity support provided by a service. Furthermore, we would like to use CDL to capture the conversations about atomicity that services engage in in order to collectively ensure global atomicity. Integrating atomicity with CDL allows multiple services to agree on the atomicity model (compensation or two-phase commit) up front when they bind to each other based on which conversations they support. In this section we examine more closely what it might mean to integrate atomicity (specifications) with CDL.

If we consider the example from Figure 1, the manufacturer engages in two conversations: one with the supplier and one with the shipment service (CDL supports 2-party conversations only). We illustrate the corresponding state machines in Figure 3.

We depict interactions as transparent boxes with black connectors. The connectors capture the inbound and outbound documents for an interaction (e.g order and unavailable are document types). The inbound documents are on top of the box, and the outbound documents are on the

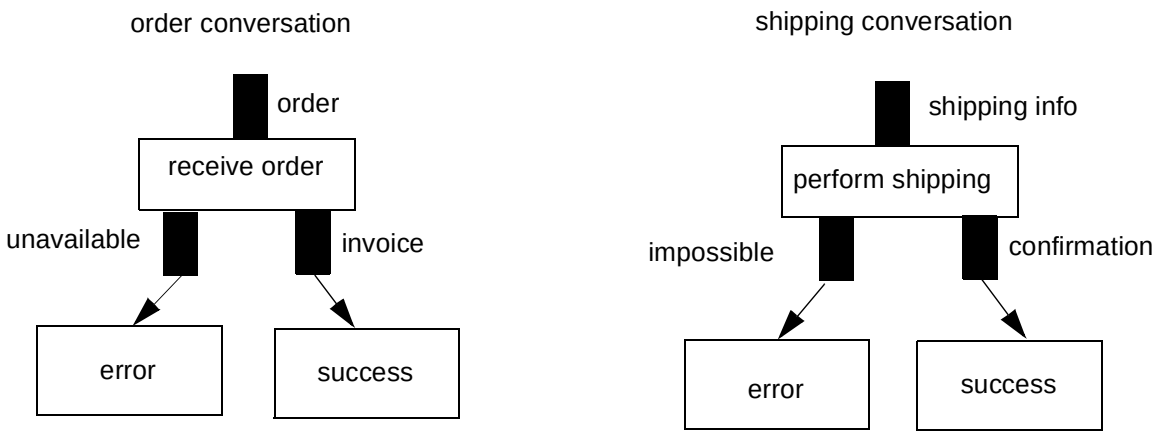


Figure 3. State machines for the order-ship conversations

bottom. The text within an interaction box is the name of the interaction. The arrows depict transitions. A transition connects one interaction with another. A transition is triggered by the sending or reception of documents. Both conversations contain a single ReceiveSend interaction, and the triggering documents for the end states are the outbound documents from the ReceiveSend interaction.

Compensation and CDL

The basic idea is to make compensation an explicit part of conversations. From a CDL perspective, interactions that compensate are no different from "regular" interactions. Whether a particular interaction updates the state of a service or compensates for a previous update is up to the service implementer. In particular, we do not provide a systematic way to generate a compensating conversation from a regular conversation. The fact that a given interaction compensates for one or more previous interactions may only be evident from the names and types of the documents that trigger the compensating interaction.

To illustrate the notion of compensation in a CDL context, consider Figure 4, which shows the order conversation from Figure 3 enhanced with a notion of compensation. The left-hand side is the conversation without any notion of compensation; the right-hand side shows a conversation in which an order can be cancelled. The cancellation can either trigger submission of another order or it can simply terminate the conversation. Thus, there are two types of documents that trigger compensation: redo order and cancel. If the compensation fails, the conversation terminates in an error state. A manufacturer may invoke one of these compensating actions if it is impossible to ship the part from this supplier (the manufacturer may then contact another supplier).

The compensation in Figure 4 cancels the effects of the previous interaction. In general, compensation may "reverse" an ongoing conversation an arbitrary number of steps, perhaps all the way back to the beginning. That is, we envision conversations where a single compensation document triggers cancellation of the entire conversation.

In general, we may have nested conversations. For example, the interaction in one (high-level) conversation, may be implemented as a (lower-level) conversation with another set of services. The nesting of conversations gives rise to the nesting of compensation. Thus, for a service *s* to

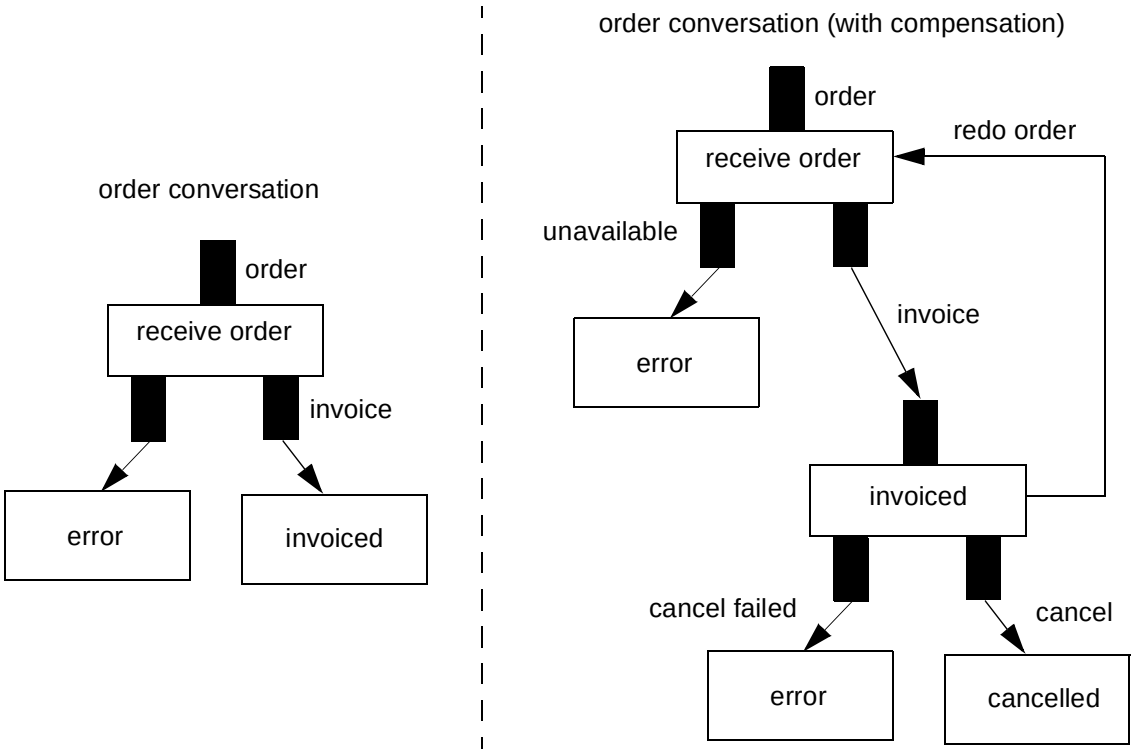


Figure 4. The order-placement conversation with compensation

compensate a particular interaction with a client, *s* may have to compensate for a conversation it had (or has) with another service *s'*. To perform this type of conversation, s will first of all have to remember the identity of *s'*, and it will have to remember the exact conversation it engaged in with *s'*.

Because compensation inherently takes place at the semantic level, and because we do not want to systematically construct compensating conversations from "regular" conversations, tracking the dependencies between nested conversations is an application-level matter. Hopefully, we can build an infrastructure to help applications track such dependencies. The need to remember, in order to compensate, means that the ability to compensate is likely to be time-bounded. As we remarked previously, the expiration time may be subject to negotiation between the services.

Two-Phase Commit and CDL

Let us first recapitulate the use of two-phase commit in an internet web services context to provide global atomicity. We assume that the parties in a global transaction are independent e-services that communicate using CDL conversations. Each service may have its own internal transaction-processing system, with a transaction coordinator that terminates local (intra-service) transactions. The role of the global transaction co-ordinating mechanism, in terms of two-phase commit, is to orchestrate the communication between the local transaction coordinators in order to accomplish atomicity for the global transaction. The global picture is that the transaction coordinators are organized in a tree, where the root is the global transaction coordinator who determines the outcome of the global transaction. Another role of the global co-ordinator is to facilitate the construction of this tree of coordinators.

We can use CDL to express the communication between the various transaction coordinators as conversations. These conversations will then be concerned with building the tree structure and with executing a hierarchical two-phase commit protocol in this tree structure. The alternative to using CDL to describe these conversations is to decide on a particular protocol between transaction coordinators up front, and hard-wire this protocol into all services that wish to participate in global transactions. With CDL, we can have a more flexible scheme in which the inter-coordinator protocols are defined explicitly. For example, we can add new protocols to the system later on, and if these new protocols conform to the old protocols, we can support evolution with backward compatibility. One starting point may be to express TIP in CDL, and then make that the basic two-phase-commit conversation.

Each global transaction has an *initiator*[1], which is the service (or client) that demarcates the transaction. Each service within the transaction has a local transaction coordinator. These coordinators can play two roles relative to the global transaction: *master* or *slave*. The (global) notions of master and slave are simply re-incarnations of the (local roles of) coordinator and participant. A coordinator can be both a master and a slave, this happens if it is neither root nor leaf in the tree of coordinators. The root of the tree is only master, and the leaves of the tree are only slaves. The master-slave distinction determines who gets to vote (the slaves) and who collects the votes and determines an outcome (the masters). The root master determines the outcome of the global transaction; intermediate masters determine the outcome of sub-transactions.

We may not want the initiator to always be the global master. For example, the initiator may be a client (e.g. a browser), and we may not want a client to control a global transaction. Moreover, we may want to allow third-party entities, who are not even services within the transaction, to be the global master. For example, we may want to define a global transaction-termination service (run by HP of course) that can function as master (because it is trusted to be up and impartial).

There are many ways to instantiate the above concepts in CDL. Here we outline one such way in order to make the concepts slightly more concrete. The starting point is a number of conversations, for example the conversations from the order-ship example. The question is how to extend these ‘‘regular’’ conversations with transactional semantics. Our proposal here is for a service to export a conversation that defines its role (master or slave) relative to clients (or other services). If a service is willing to play both roles, it can export multiple conversations.

In Figure 5, we illustrate how to enhance conversations with transactional semantics. The left-hand side of the figure shows how to enhance a server-side conversation with the role of a master. The client first sends a document to the service to ask the service to establish a transactional context. The service establishes a context, becomes master for it, and returns the context to the client. The client can then use the context to enroll other services in the transaction (if those services are willing to be slaves relative to this particular context). After sending the context to the client, the service engages in the regular conversation. When the regular conversation is complete, the service waits for a document that terminates the context. The service cannot terminate the context until explicitly told so because the context may still be active at other services. The server-side implementation of the terminate interaction is to activate the transaction coordinator within the service to play the role of global master and execute a global two-phase commit with the other

1. Not to be confused with the initiator of the conversation. If A sends the first message in a conversation, then A is the initator of the conversation, and stays that for the rest of the conversation (see chapter about CDL). Yet during the conversation any participant can initiate interactions.

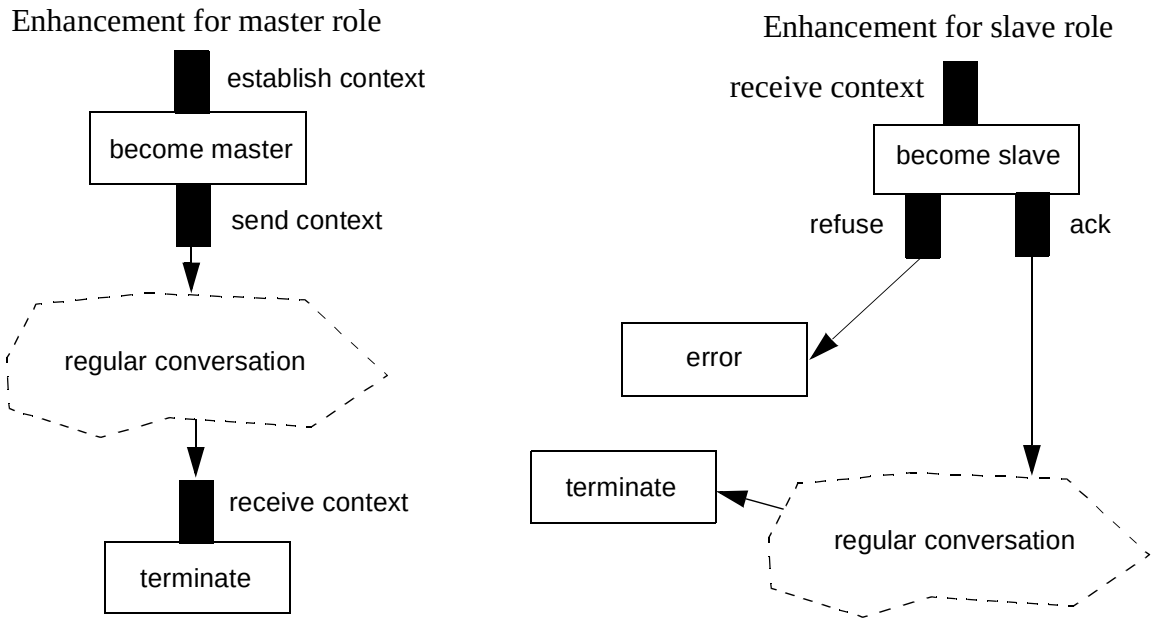


Figure 5. Enhancing conversations with transactional semantics

transaction coordinators. The other transaction coordinators have used the context to register, directly or indirectly, with the global master. If the tree of coordinators have more than two levels, we have indirect registration.

The right-hand side of the figure depicts a way to describe transactional enhancements for a service that plays the role of a slave. Because it is a slave, it receives a transactional context from the client. After receiving the context, the service becomes a slave by instructing its transaction coordinator to register with the coordinator identified by the received context. The service may refuse to do so (if it does not like or trust this particular coordinator). If the service becomes a slave, it then continues with the regular conversation. When the regular conversation is complete the service activates its local coordinator to engage in a two-phase commit protocol as slave. Notice that a slave enters its part of the two-phase commit without further interaction with other services. This is because it is a passive entity in the two-phase commit protocol, simply waiting for a vote request.

The “become master”, “become slave”, “terminate”, “error” conversations shown in Figure 5 are predefined two-phase commit conversations that can be described by CDL like any other conversations. These predefined conversations are then added to any regular conversations carried out between two services needing two-phase commit. These predefined two-phase commit conversations (also called inter-coordinator conversations, as the participating services take on the roles of coordinators) have two aspects:

- A service that takes on the role of a slave coordinator executes an enrollment conversation with another service that acts as its master coordinator. The enrollment is relative to a specific transaction context.
- All services engage in a two-phase commit conversation to terminate the transaction. The global master starts the commit conversation based on the outcome of the regular conversation with the initiator. Of course, the initiator and the global master may be the same service, in which case noone uses the conversation with the master role.

We do not give here CDL descriptions for the inter-coordinator conversations. These conversations will largely follow the conventional two-phase commit pattern. However, there are a couple of things worth noting:

- Two-phase commit is inherently a multi-party protocol. A master transaction coordinator communicates with a number of slave coordinators to obtain their vote and instruct them how to terminate their (part of the) transaction. Currently, CDL is a two-party conversation model. Thus, we either have to make CDL a multi-party conversation model, or we have to describe the two-phase commit conversation as a number of two-party conversations that are composed in a manner that is not captured in CDL.
- Two-phase commit enhanced conversations can be seen as having two conversations in parallel (the regular conversation and the inter-coordinator conversation) that are highly interrelated and need to be coordinated by the internal logic of the service. Or two-phase commit enhanced conversations can be seen as one conversation aggregated from the regular conversation and the inter-coordinator conversations.
- As inter-coordinator conversations will be predefined, it could be feasible to simply enhance CDL and other conversation description languages by attributes denoting which interactions are transactional and which two-phase commit protocol is used.
- A given transaction coordinator can simultaneously engage in conversations as both slave and master, even for the same transaction. This will happen if it resides neither at the root nor at the bottom of the tree.

7.6 Conclusions

Our primary conclusion is that providing multi-party atomicity on the Internet is a complex problem that is unlikely to have a single solution that covers all cases. Simply re-coding XA in XML is not flexible enough. This is not surprising since XA was designed for tightly-coupled systems. On the other hand, completely discarding two-phase commit as a possible solution for certain scenarios is probably too extreme. The wide-spread support for XA makes it a highly attractive basis for a more global, wide-reaching transaction/atomicity protocol. If we can build a protocol where the XA operations are taken as the fundamental building blocks, we very quickly make much existing infrastructure compatible with transactional web services. If we are willing to live with the drawbacks of two-phase commit, the TIP protocol seems to be a promising way to glue together XA-based transaction managers into a global hierarchy that provides global atomicity.

To support long-running, global transactions, we need some ability to release local resources before the global transaction terminates. Compensation is one way to achieve this goal. In contrast to two-phase commit conversations that can be predefined, the various interactions needed for componsation need to be added explicitly into each business conversation.

8 Managing an E-Service

This section defines the XML Application Response Measurement (XAM) specification. It follows in the footsteps of, and aims to be backward compatible with, the Application Response Measurement (ARM) standard as defined by the Open Group (http://www.opengroup.org). Like ARM, XAM is designed to measure the availability, performance, usage, and end-to-end transaction response times of distributed applications. This document defines a measurement hierarchy, a protocol for XML-based document exchange between application and measurement agent. It focuses on ways in which measurement information is collated, aggregated, delivered, and controlled. It assumes the reader to be familiar with the terms and concepts presented in the ARM specification, available at http://www.opengroup.org/management/l2-arm.htm.

8.1 ARM

The Application Response Measurement (ARM) standard as defined by the Open Group, is a measurement standard that allows application developers to instrument their code with little overhead. The programmer defines transactions that are specific to the application. In the simple case, calls to the ARM routines are inserted when a new transaction is started and when the transaction terminates. The standard specifies the semantics of the six ARM routines and how measurement types can be associated with transactions. The implementation of the routines is left to the ARM provider and the manner with which the transaction data is stored and transported to the management stations is undefined.

ARM has the following good points:

- It is relatively simple to understand and use.
- It does not constrain programmers to use standard transaction types.
- It is widely used (for a measurement standard).

Why do we need something more than ARM?

- The ARM calls only provide for instrumentation of one particular type of measurement the transaction. There are other valid and useful measurements that we would like to capture.
- The ARM standard assumes that a library such as a DLL, or a shared object is present on the target machine, possibly a daemon process, disk storage and that an unspecified communications path exists between the target machine and the management console. This is valid for intra-enterprise management but not for inter-enterprise management.
- The ARM standard does not specify how measurements are encoded for transportation to the management system, or how the management console controls which transaction types are recorded.
- The ARM standard does not specify what types of aggregation are possible and where the processing is performed.

XAM

This section is concerned with how the measurement information is collated, aggregated, delivered and controlled. The ARM calls could be used as defined in the standard for instrumenting

transaction response times. Other API's could be standardized for instrumenting other measurement types.

In this specification we generally follow two guiding principles that ARM advocates. These are:

1. Make it simple for the programmer.
2. Let the programmer define the measurement types.

Extending ARM to include more measurement types may be misunderstood as extending the measurement types defined in ARM to convey measurements about transactions. Some of the measurement types defined by ARM are gauge, counter, opaque etc. The purpose of XAM is not to extend transaction-related measurements but to introduce application-related measurements, of which transaction-related measurements are a subset.

The following entities will be used throughout this document:

- **E-service:** The application which has been instrumented.
- **XAM API:** The application programming interface to the XAM library.
- **XAM library**: The code the application invokes and which sends and receives the XML documents.
- **XAM service:** The remote entity which receives the measurement reports and which confiures the XAM library.

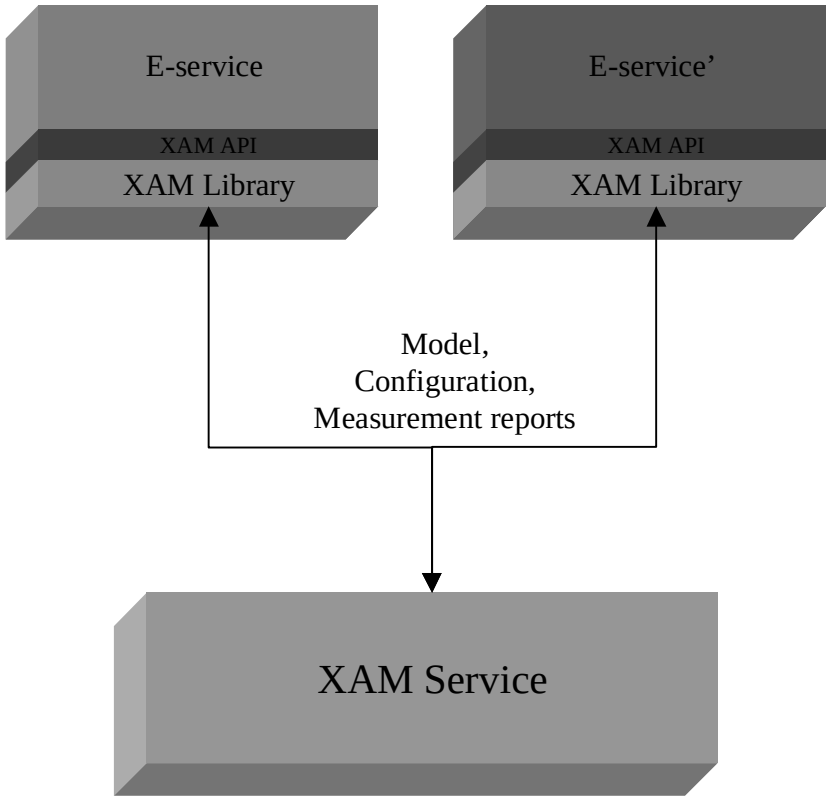


Figure 25: E-service, XAM API, XAM Library, and XAM service

The XAM specification assumes that a particular e-service/XAM library instance will have a single XML dialog with an XAM service. However several e-services may converse with a single XAM service instance. The XAM service may pass the measurement data to several other parties. Only one entity may converse with the XAM library at one time; the XAM library has one master.

8.2 Measurements

8.2.1 Measurement Hierarchy

Applications, services, transactions, and conversations need to be instrumented for management purposes. The instrumentation code makes various measurements. There should be a mechanism both for expressing (representing) these measurements and communicating them with management systems.

The measurement hierarchy is designed to allow expression of two types of measurement instrumentation:

1) Instrumentations that are placed into applications and frameworks at their design time. This instrumentation should be written to be generic and should be usable by multiple management systems.

2) Other types of instrumentation may be specific to a management system and are usually deployed into the application environment by the management system itself to collect additional measurements.

XAM can be used to express, control and deliver measurements in these two forms.

One important goal of a measurement hierarchy definition is that it expresses more semantics to the receiver of the measurements (management system in this case). The extreme way for bringing an agreement between the managed application and the management system is to define every possible measurement and its semantics. Since the number of possible measurements is very large, this is not feasible. Hence, a measurement category system provides a framework for defining real measurements that can be understood by a management system. Just using the type definitions in the hierarchy, the management system will be able to interpret, operate on, and display the various measurements. One of the main advantages of the measurement hierarchy is that the amount of code needed to be written is bounded by the size of the hierarchy and not the number of different types of measurement that applications produce.

Another important goal of the hierarchy is to accommodate measurements that are both primitive and aggregated. If the type hierarchy does not include any measurements that correspond to aggregates, then a lot of data has to be passed between two parties, and it involves performing the same computations everywhere. However, the number of possible ways in which aggregation could be done is not bounded. So, a compromise has been made on the kinds of aggregations that

are supported. These aggregate categories supported are count, sum, group and threshold these in turn support topN (the N most frequently occurring items from a set).

8.2.2 Measurement Type System

This document uses a three level system for defining measurement categories, measurement types and measurement instances. Each level follows from the last. We expect the top level (which defines the different categories of measurement) to be static. The second level, which defines measurement types, is where application programmers and standards organizations will specify well-known types. The third level is where the actual measurement values come in.

For example an application that wishes to measure the names that new files are created under could use the atomic category and define a atomic measurement type with the type ID field equal to "fileCreated". When an instance of this measurement is sent out it may have the fileName value of "/docs/index.html".

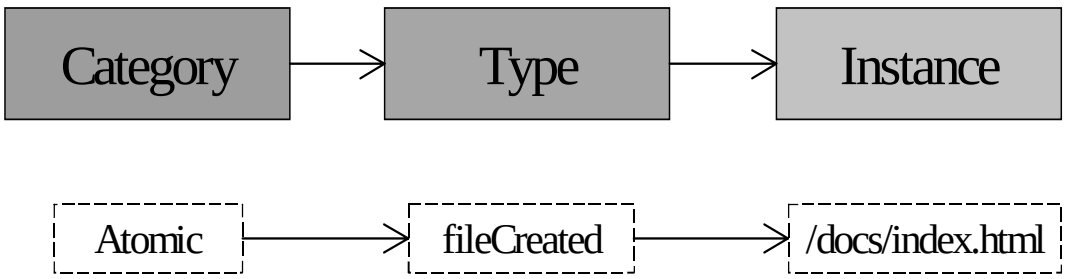


Figure 26: Example Application

8.2.3 Measurement Categories

What categories of measurement do we need?

For the 1.0 release of XML Application Measurement (XAM) we will specify three main categories of measurement, each with some levels of sub-categories. Later versions of XAM could add extra sub-categories if needed; the XML is excellent for allowing such additions.

At the top of the measurement hierarchy is the category Measurement. The three important sub categories of Measurement are Polling, Occurrence and Aggregate. Each differ in the way that they treat time. Polling measurements are those where a sample must be taken to retrieve an actual value. Occurrences are asynchronous events that may occur at any time and we report each individual happening. The Transaction sub category also has a duration field, which is the time interval between the start of the transaction and the end of the transaction. Aggregate measurements report things that either do not have discrete time based events or need to be sampled or counted to reduce the data rate. Aggregate measurements can perform aggregation over occurrence variables and polling variables.

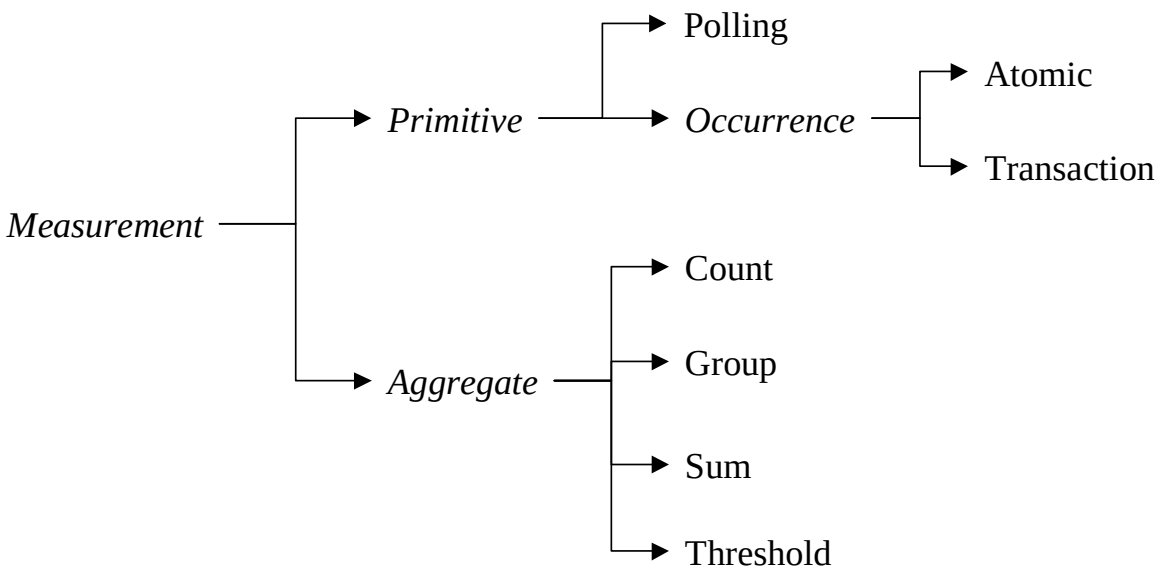


Figure 27: Measurement Category Hierarchy

The figure above shows the measurement category hierarchy. Three of the categories are abstract and simply form part of the information model. These categories are Measurement, Occurrence and Aggregate.

A definition of each category can be found below:

Category	Description
Measurement	The root of the category hierarchy.
Primitive	The root of the measurements that are supplied by the application.
Occurrence	Measurements of incidents that happen asynchronously. I.e. some action performed by the application that could occur at any time.
Aggregate	Measurement over time period which collate or sample the primitive measurements. The primitive measurements are Polling and Occurrence and their sub categories. All sub categories of Aggregate can be configured to either report all data over the time period or to report the topN of the data over the time period (see description of topN below)

Polling	Measurements of things that either are changing to rapidly which must be sampled to at some rate to create an aggregate measurement. E.g. free memory. Or measurement data, which is available on request but cannot for some reason be asynchronously supplied to the library.
Atomic	The concrete subcategory of Occurrence. Can be used to report application actions or measurable events.
Transaction	A specialization of Occurrence which includes a time duration which is calculated by the XAM library.
Count	A concrete subcategory of aggregate. Can be used to report things such as counts of instances of an Occurrence measurement type.
Group	A concrete subcategory of aggregate which reports the frequency of a group of Occurrence measurement types. Each Occurrence type in the group has its own frequency.
Sum	A concrete subcategory of aggregate. Can be used to report summations of variable values from occurrence instances. Can be used to report the average, min, max and standard deviation of the variable values.
Threshold	A concrete subcategory of Aggregate. Is used to differentiate between good and bad data and count in which case each instance of a value falls.

Of the seven concrete categories, three can have specific types defined by the application programmer; the others are auto-generated from the three primitive categories. The category that corresponds to ARM is Transaction. The ARM API's have also been used to produce Atomic measurements by calling arm_start() and then arm_end() immediately.

TopN

The idea behind the topN is to sample in such a way as to report the most important information. For example, an application may have several hundred different error conditions which may occur during normal operation. If the frequency occurrence of these errors is such that reporting each one individually is to costly, then the topN is the solution. The topN reports the N most numerous errors in each reporting period. Sampling techniques can make calculating topN's less costly while retaining the statistical properties of the data.

8.2.4 Measurement Variables

All XAM measurement types have a list of variables that are specified by the application when they are defined. When the application gives actual measurement data it must supply values for the variables defined. When defining a variable the following information is supplied:

Field	Possible values	comment
Name	Any combination of alphanumeric characters and '-', '.', '_','':'. Must start with a letter. E.g. "user_name".	':' is reserved for use by the XAM system and may not be used in type names specified by the application. Alphanumeric includes Unicode characters not in the ASCII set.
Class	One of **string**, **long** or **double**	
isKey	Either **true** or **false**	Default false. Variables of class double may not be keys.
Alphabet	Either an enumeration of the valid string values or a range list for the long or double. E.g. Enum = **{ "ready", "running", "stopped" }** Range = { "**11","12","15","32..126"}**	Optional
Unit	A string describing the unit of the variable. E.g. "kilos", "sessions", "requests", "seconds"	Optional. If unspecified the name field is used.
Desciption	Can contain any string, might contain a URL to a localization service and a message ID and. E.g "**the number of users logged in**" "**http://www.locals-are-us.com/msgId324432432**"	Optional

8.3 Measurement Type Information Model

For each measurement type defined by the programmer there is a set of fields that detail the specification of the measurement type. The fields for each measurement type include the fields defined for the measurement category that type was created from.

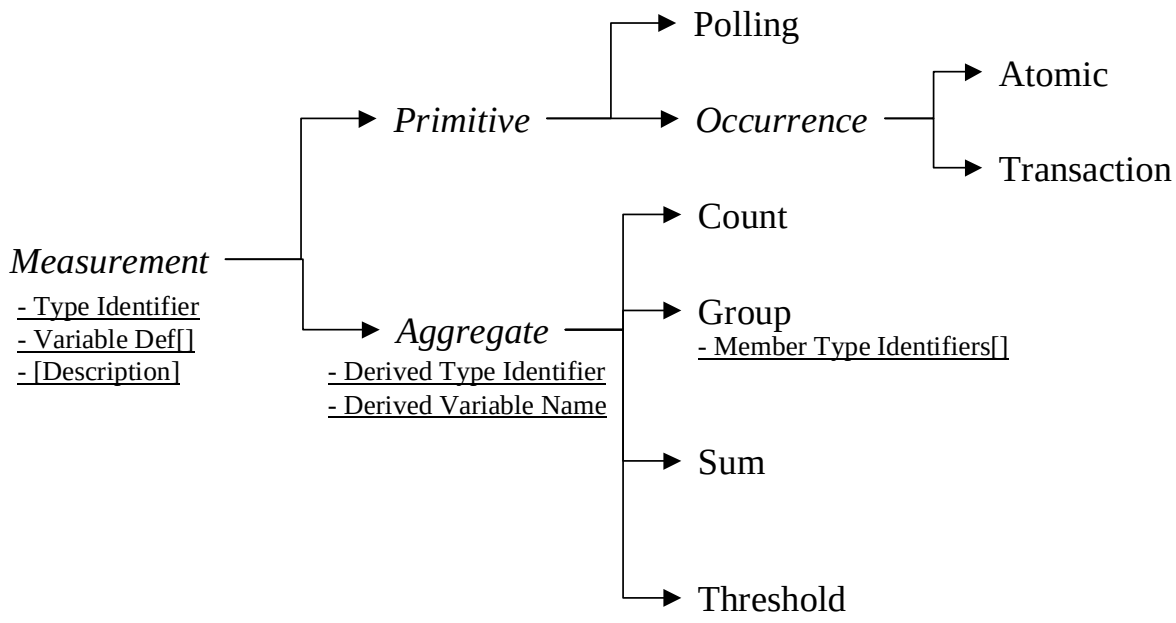


Figure 28: Measurement Category Hierarchy

Black underscore text is for model information, which describes the transaction types the application has instrumented for. This model information is sent by the application to the management system in the ‘measurement type catalog’ as part of the details document.

The type fields for each category are defined below:

Category	Name	Class	Description
Measurement	Type Identifier	String	A unique name with which the application identifies a measurement type. Same rules as for the measurement variable name field.
	Description	String	A human readable description of the measurement {optional}
	Variable Spec	Set of specs as defined in section 3.5.3.4	A set of variable specifications as described in the previous section. No two variables may have the same name within one type.
Aggregate	Derived type ID	String	The Type Identifier of the type which this measurement uses for source data.

	Derived variable name	String	The variable within the type which is used as source data for this measurement type. May be empty for certain aggregate categories.
Group	Member type IDs	String[]	An array of Type Identifier names one for each member of the group.

For each measurement type registered by the programmer (polling or occurrence) there will be a number of aggregate measurements automatically generated. The XAM library will give the XAM service a catalog of available measurement types. The measurement type catalog will contain all the occurrence sub types defined plus any automatically generated aggregate types.

8.3.1 Measurement Request Information Model

For each measurement type in the catalog the XAM service may request a measurement report to be generated. The XAM library holds a list of requested measurements. Each item in the request list corresponds to one measurement type's configuration data. The configuration data stored depends on the category of the measurement type being requested. The request configuration data for each category is described below.

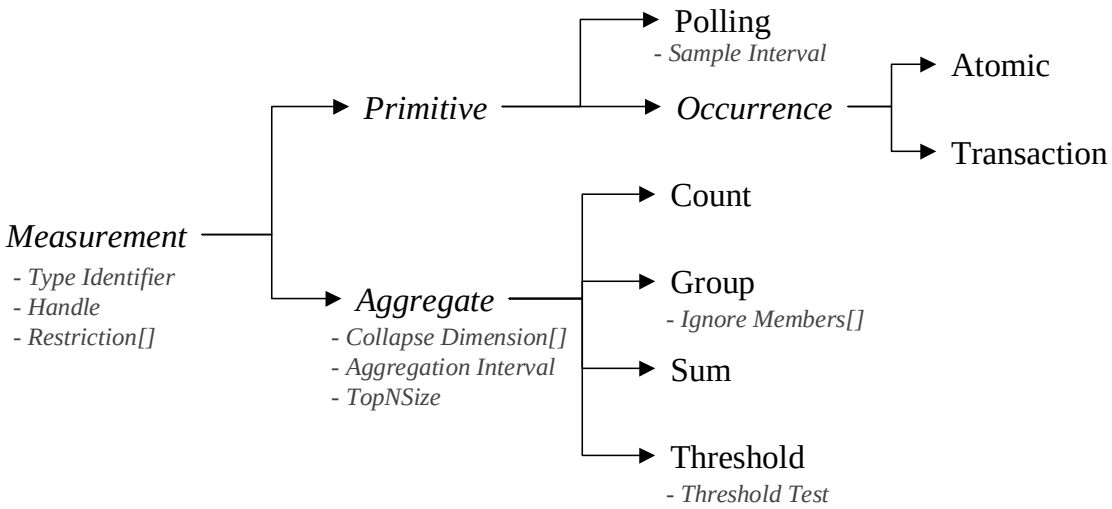


Figure 29: Request Configuration Data

Each category inherits its parent’s variables as per normal. For example: configuration data for a measurement type of category Atomic will have the following fields: Type Id, Handle and Restriction[].

Category	Name	Class	Description
Measurement	Type Identifier	String	A unique name with which the application identifies a measurement type
	Handle	String	A handle set by the XAM service and which is set along with the measurement results.
	Restriction	String[]	For each variable there may be a restriction. For definition of restriction see below.
Aggregate	Collapse Dimension	Boolean[]	Indicate to the XAM library if the key field in question is to be used when cutting the data. Default true. See 3.5.3.12.2 for example of usage.
	Aggregate Interval	Double	The time over which this aggregate measurement is to be taken before the aggregate result is calculated. Unit is seconds.
	TopN Size	Long	The maximum number of measurements results to be returned in any one reporting interval. Range = -1,1..200
Group	Ignore Members	Boolean[]	Indicates whether each of the types that make up the group should be included in the group for reporting purposes.
Threshold	Threshold Test	A restriction	The test which discriminates the values.
Polling	Sample Interval	Double	Seconds between samples.

Restrictions

When a measurement request is registered with the model a restriction set can be specified. For each variable defined a restriction of one the following forms can be used:

Class of Variable	Restriction	Description
String	Enumeration	The value of the variable must match one of the elements in the enumeration. E.g. **{ 'ready', 'running', 'stopped' }**

	Prefix	The first characters of the value of the variable must match the complete prefix string.
	Postfix	The last characters of the value of the variable must match the complete postfix string.
Long or Double	Range set	The value of the variable must lie within the ranges specified.

The restriction sets use the same format for enumeration and range as the measurement variables in section 3.5.3.4.

If the variable does not match the restriction then the instance data is not used for this measurement. See 3.5.3.12.1 for examples of restrictions. If no restriction is specified for a variable then no checking against the value will take place.

8.3.2 Measurement Instance Information Model

For each measurement request registered by the XAM service, XML measurement reports will be sent out. The report data delivered depends on the category of the measurement type being reported. The report for each measurement type includes the fields defined for the measurement category that type was created from.

The measurement report fields are shown below:

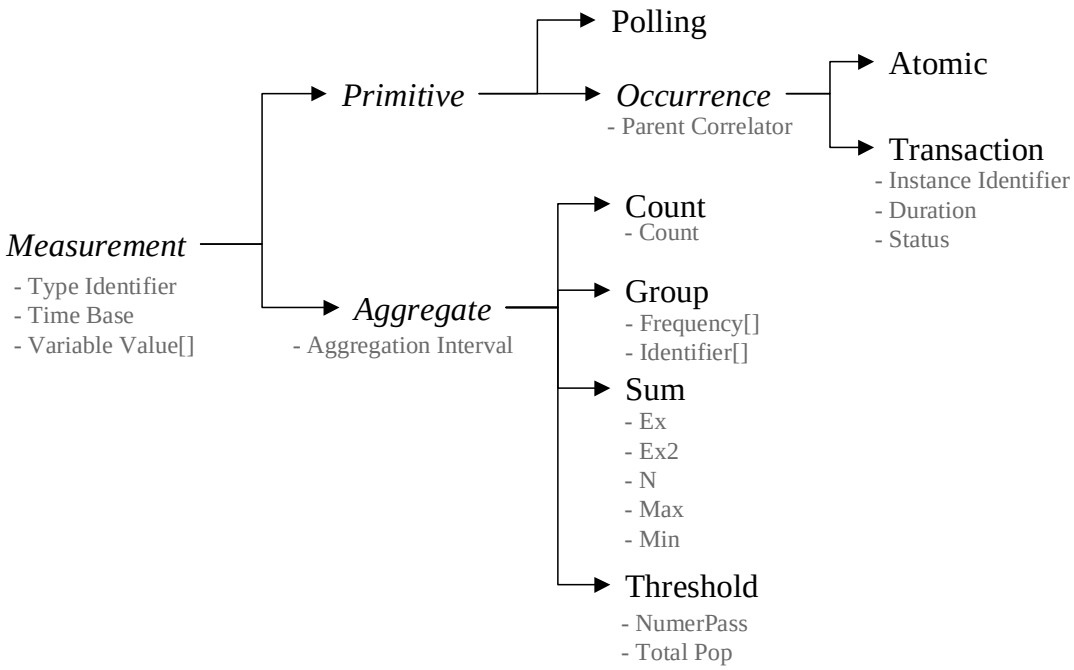


Figure 30: Measurement Report Fields

Sub categories include all of the fields of their parents. For example reports for measurement types of category Transaction have the following fields: TypeId, TimeBase, ParentInstanceId, InstanceId, and Duration.

The reporting fields for each category are defined below:

Category	Name	Class	Description
Measurement	Type Identifier	String	This field is defined as part of the measurement type information model and is sent along with every report.
	Time Base	Time	The time when this measurement happened. For a transaction this corresponds to the start time. For a polling value it will correspond to the time when the callback method is called. [UTC time string]
	Variable Value	Value[]	Either a String Long or a Double.
Occurrence	Parent Correlator	String	A field which links several different transactions together. Can be used to link up transactions which occur on different entities but which are initiated by the same source. This field may be empty.
Transaction	Type Instance Identifier	String	This field is created in by the XAM library and may be used as a valid value for part of the Parent Correlator field. Any Unicode character is valid except '!'. Should be world and time unique.
	Duration	Double	The time between the call to start() and the call to end(). This value is calculated by the XAM library.
	Status	String	Whether the transaction was completed sucessfully on not valid values are "ok" or "failed"
Count	Frequency	Long	The number of instances of the occurrence type delivered by the application during the reporting period.
Group	Frequency	Long[]	As above for the type within the group.
	Identifiers	String[]	The type Identifier's that go with the frequencies above.

Sum	Ex	Long	The summation of value of the variable over the reporting period.
	Ex2	Long	The sum of the squares of the variable over the reporting period.
	N	Long	The number of variables summed over the reporting period.
	Max	Long	The highest variable during the reporting period.
	Min	Long	The lowest variable during the reporting period.
Threshold	NumberPass	Long	The number of instances in which the value passed the threshold test.
	Total Pop	Long	The number of instances of the value that were tested.

8.3.3 Correlator

The correlator is used to create aggregate measurements, which tie two or more transactions together. The correlator from one transaction may be provided when a sub or daughter transaction is started. If the sub transaction is started on another process then the correlator of the parent must be transferred to the remote process (probably along with the action which starts the transaction). The correlator can be used by the XAM library to create further aggregate measurements.

The correlator contains the following fields:

1. System Identifier (defined in 3.5.3.8)
2. System Instance Identifier (defined in 3.5.3.8)
3. Type Identifier (defined in 3.5.3.5)
4. Type Instance Identifier (defined in 3.5.3.7)

The four fields above are encoded into a string, which can be passed around from one application to another. The format is a Unicode string of the following form:

"System Identifier! System Instance Identifier! Type Identifier! Type Instance Identifier"

The '!' character is explicitly not a valid character in the four sub fields. Any or all of the four fields can be empty for example "!!!" is a valid correlator field and the default if one is not specified.

Calculated Variables

The Transaction category is unique in that is has the ‘duration’ variable, which is very similar to a variable supplied by the application but that it is calculated by the XAM library. Because the library calculates this variable the application programmer cannot calculate further variables from

the duration. The XAM library calculates these variables if required. For the measurement types of category transaction a number of calculated variables are created these variables are not directly visible but cause extra aggregate types to be created. The following calculated variables are created.

For all transaction measurement types (***T***) {

Within ***T*** For each non string variable (***V)***:{

Register a variable in T

with Name = '***V***_by_Duration**'**

with Key = false.

with unit = '***V.unit***/second'

with class = double.

}

}

These calculated variables are never directly delivered to the XAM Service.

8.3.4 Measurement System Information Model

Each application has a set of fields that affect the collection and reporting of all the measurement types which the application produces.

System field name	description	class	Change-able
Enable	Boolean field, which controls if any measurement reports are generated by the application. Default is true.	Boolean	Yes
Service Identifer	The name of the application or service will be the same for all instances of the service that ever run. Eg "virtual file system version 1.5.4". Any Unicode character is valid except '!'	String	No
Service Instance Identifier	A string that will uniquely identify the instance of the application or service. Eg. "15.144.69.2:pid=1234". Any Unicode character is valid except '!'	String	No
Service Description	A description of the service or application or a URL to a localization service and a message ID	String	No
Clock Resolution	The smallest period of time that the library can measure in seconds.	Double	No

minReportInterval	Time is seconds between the application sending measurement reports. If this is set to zero then a report will be sent as soon as measurement data is available. Values above zero allow the application to bundle up several measurements into a single XML report document. Default is 5 seconds.	Long	Yes

8.3.5 Time

Measurement reports are sent out at least every minReportInterval and contain the measurement data collected since the last report was sent. Atomic instances are sent in the next report. Transactions are reported when they finish. Aggregates also produce measurement report information at the end of their AggregateInterval. Polling measurement types are sampled according to the sampleInterval. If AggregateInterval and sampleInterval are the same the polling measurement type will be sampled once only. Depending on the semantics of the polling measurement this may or may not make sense.

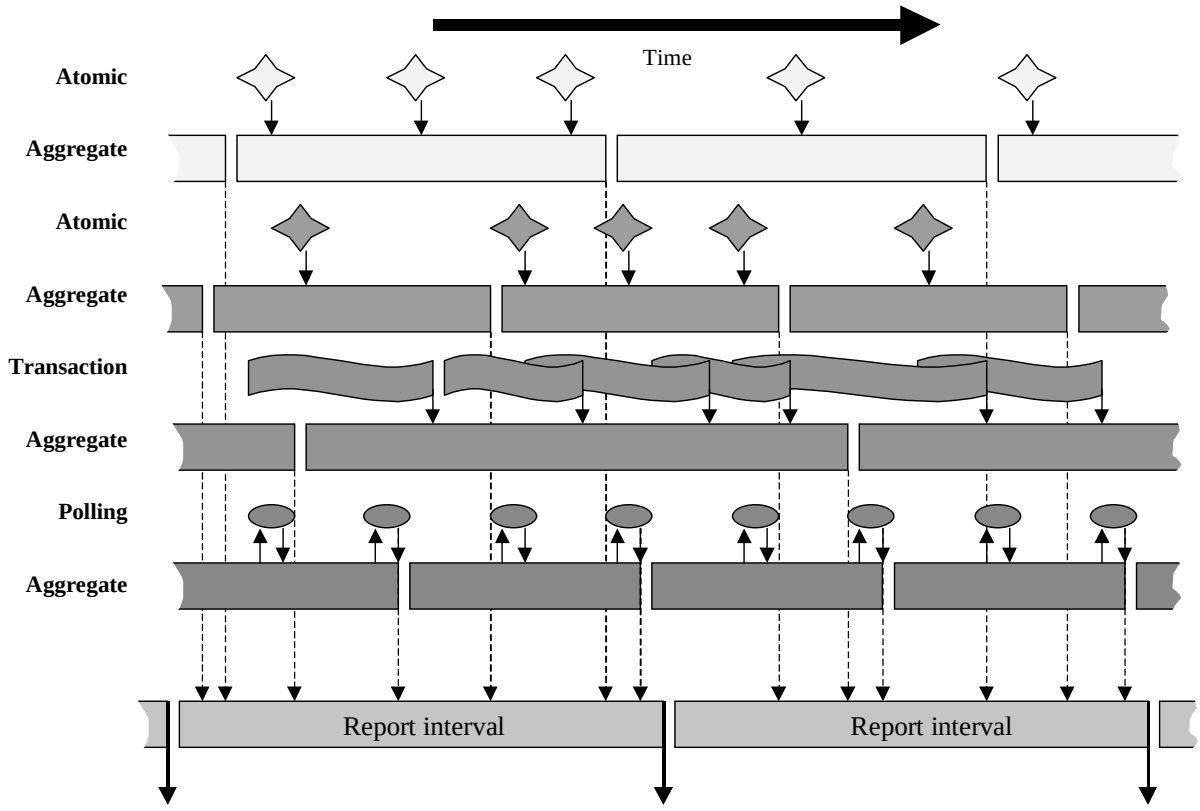


Figure 31: Measurement Reports sent over Time

The diagram above shows three occurrences types being aggregated and one polling measurement being sampled. The data from these four aggregates is shown in relation to the report interval and is packaged into the reports send at the end of the report interval.

8.3.6 Aggregation

There are four sub-categories of the aggregation category:

Count – each instance of a measurement type of category ‘occurrence’ can be counted.

Group – a set of measurement types of category can be grouped together and counted. The frequency of each measurement type in the group is stored.

Sum – each occurrence of a measurement provides an value which is summed, The sum of the values and the sum of the squares of the values is calculated. From these fields the average and standard deviation can be calculated.

Threshold – each occurrence of a measurement provides an value which is compared against a restriction and the number of instances which pass the threshold are counted along with the total number of occurrences.

In every case some time interval must be chosen over which the aggregation is performed.

Key Dimensions

Every aggregate type can have a number of key variables, which are transferred over to the aggregate from the occurrence or polling measurement type. Only the variables that are keys are transferred to the aggregate. The instance variable key information will be retained for every key variable that has a "collapse dimension" field value of 'false' in that measurement request. This means that at the end of a reporting period (for an aggregate measurement type with one or more key variables) an unspecified number of measurement instance reports may be sent from the XAM Library to the XAM service.

For an example of an aggregate type with multiple keys see section 3.5.3.12.

If the size number of keys is large, the number of measurement instance reports may approach the number of Occurrence instances in the reporting period and there will be no advantage to the aggregation. In this case, using the topNSize is efficient way to reduce the unknown set too a finite set. See section 3.5.3.3.1 for a description of topN's.

Automatic Derivation of Aggregate Types

The XAM library automatically generates a catalog of measurement types, which includes a set of aggregate measurement types that are derived from the occurrence and polling types, which have been registered by the programmer. The following aggregate types are derived.

For all polling measurement type (***T***) {

Within ***T*** For each non key non string variable (***V)***:{

Register a measurement type of category Sum

with TypeId = '***T***-Sum-***V***'

with Variables = key variables from ***T***

with DerivedTypeID = ***T***

with DerivedVaraiableName = ***V***

Register a measurement type of category Threshold

with TypeId = '***T***-Threshold-***V***'

with Variables = key variables from ***T***

with DerivedTypeID = ***T***

with DerivedVaraiableName = ***V***

}

}

For all atomic measurement type (***T***) {

// same as for polling plus…

Register a measurement type of category Count

with TypeId = '***T***-Count'

with Variables = key variables from ***T***

with DerivedTypeID = ***T***

}

For all transaction measurement type (***T***) {

// same as for atomic plus…

Register a measurement type of category Sum

with TypeId = '***T***-Sum-duration'

with Variables = key variables from ***T***

with DerivedTypeID = ***T***

with DerivedVaraiableName = 'Duration'

Register a measurement type of category Threshold

with TypeId = '***T***- Threshold -duration'

with Variables = key variables from ***T***

with DerivedTypeID = ***T***

with DerivedVaraiableName = 'Duration'

// don't forget the calculated variables derived from Duration

// which will fall into the non key non string case for atomic above.

}

A compliant XAM implementation is not required to support any of the above derivations but its utility will be significantly reduced.

Example Measurements

The table below gives an example of each of the three types of measurement that an application programmer can define i.e. Polling, Transaction, and Atomic.

Category	Type name	Variables				
		Name	Class	IsKey	Alphabet	Unit
Polling	Disk_utilization	Utilization	Dou-ble	No	0..100	percent
		Disk	Long	Yes	1..i	
		User	String	Yes		
Transaction	Download_file	File_name	String	No		
		File_type	String	Yes		
		Size	Long	No		bytes
		Invoker_ID	String	Yes		
		Duration	*Dou-ble*	*No*	*0..i*	*second*
		Size_by_dura tion	*Dou-ble*	*No*		*bytes/second*
Atomic	Security_Authorizatio n_failure	Conversation _type	String	Yes		
		Document_ty pe	String	Yes		
		Invoker_ID	String	Yes		
		Recipient_ID	String	Yes		
		Reason	String	Yes		

If an application programmer defined the three measurements above then the following aggregate measurements would be automatically created by the XAM lib.

For the ‘disk_utilization’ measurement type:

Category	Type name	Variables

		Name	Class	IsKey	Alphabet	Unit
Sum	Disk_utilization:Utili-zation	Disk	Long	Yes	1..i	
		User	String	Yes		
Threshold	Disk_utilization:Thres hold:Utilization	Disk	Long	Yes	1..i	
		User	String	Yes		

For the ‘Download_file’ measurement type:

Category	Type name	Variables				
		Name	Class	IsKey	Alphabet	Unit
Count	Download_file:count	File type	String	Yes		
		Invoker ID	String	Yes		
Sum	Download_file:sum:si ze	File type	String	Yes		
		Invoker ID	String	Yes		
Sum	Download_file:sum:d uration	File type	String	Yes		
		Invoker ID	String	Yes		
Sum	Download_file:sum:si ze_by_duration	File type	String	Yes		
		Invoker ID	String	Yes		
Sum	Download_file:sum: duration_by_size	File type	String	Yes		
		Invoker ID	String	Yes		
Threshold	Download_file: threshold:size	File type	String	Yes		
		Invoker ID	String	Yes		
Threshold	Download_file: threshold:duration	File type	String	Yes		
		Invoker ID	String	Yes		
Threshold	Download_file: threshold:duration _by_ size	File type	String	Yes		

		Invoker ID	String	Yes		
Threshold	Download_file: thresh-old:size_by_duration	File type	String	Yes		
		Invoker ID	String	Yes		

For the ‘Security_Authorization_failure’ measurement type:

Category	Type name	Variables				
		Name	Class	IsKey	Alphabet	Unit
Count	Security_Authorizatio n_failure:count	Conversation type	String	Yes		
		Document type	String	Yes		
		Invoker ID	String	Yes		
		Recipient ID	String	Yes		
		Reason	String	Yes		

Restriction Examples

For the ‘Disk_utilization’ atomic measurement here are some valid restrictions:

Utilization = “80..100” // only measurements with a disk utilization over 80% will be considered.

Disk = “1,2,3” // only measurements for disks one two and three will be considured.

User = “{“root”,”sysadmin”}”

For the ‘Download_file’ measurement type:

File name = “prefix:/dev/” // any file under /dev

File type = “{“pdf”,”doc”,”jpg”,”gif”}”

Size= “5000..i” // only files larger than 5000 bytes.

Collapse Dimension Examples

The aggregate measurement types, which are given above all, have some number of key variables. For each key variable there is a collapse field which controls if the key is used to cut the data i.e. if different measurement reports will be sent for different values of the key variable.

For the ‘Disk_utilization-SumSample-Utilization’ measurement type defined above 4 dimen-sional cuts are possible.

These are by:

None
Disk
User
Disk,User

In the first case *None* only one measurement report will be sent. In the other cases, an unknown number of reports will be sent. If the topNSize field is set to a value other than –1, the number of measurement reports sent will be limited to the topNSize value.

For the ‘Security_Authorization_failure-Count’ measurement type defined above, 32 dimensional cuts are possible. What these are is left as an exercise for the reader.

8.4 Protocol

We have two parties involved: the application and the measurement service. We assume each application has only one measurement service and that they share a communications channel down which they may exchange XML documents. This paper does not specify what form communications channel takes nor what addressing is used nor how the application comes by the services address. The specifics of the communications channel will depend on which distributed middleware solution the application uses

.

When the measurement service and the application communicate three types of information travel between them, these are Data, Control and Model.

There are seven documents that make up the protocol and these are: { init, request, details, configure, report, close, info }. The protocol has six sub-parts A-F. These are described below:

A) Initialization: The application sends an “init” document to the measurement service. The service will usually move to sub-part B or if it is not unwilling, move to sub-part E.

B) Model acquisition: The measurement service sends a “request” document to the application, the application will reply with a details document which will contain the measurement system information model, measurement type information model and the measurement request information model. This sub-part can take place any time after sub-part A and before sub-part E. If the measurement type information model changes after being sent to the measurement service the library may send out a “details” document asynchronously to the measurement service.

C) Configuration: The measurement service sends a “configure” message to the application. The application will update its internal state to reflect the document content.

D) Reporting: Depending on the application configuration, reports will be sent periodically from the application to the measurement service. If no measurements are enabled then no reports will

be sent. Reports contain bundles of measurement data collected over the last system reporting period.

E) When either party wishes to terminate the conversation they send a “close” message to the other party which replies with another close message.

F) An “Info” message can be sent by either party to the other at any time to inform the other party that something has gone wrong but the session does not need to be closed.

The protocol must start with sub-part A and end with sub-part E.
The diagram below shows a normal conversation between an application and a measurement service.

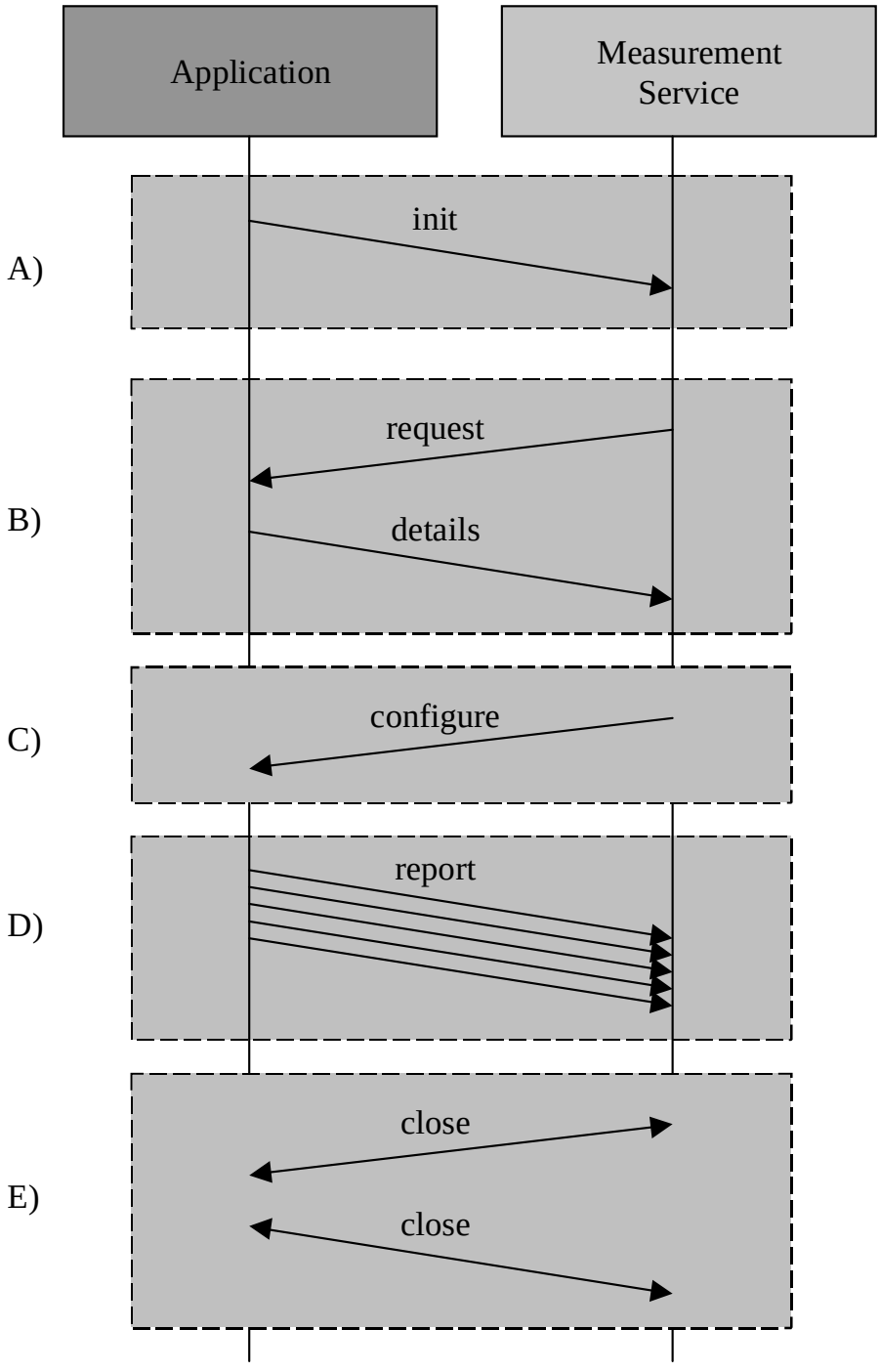


Figure 32: Normal conversation between an application and a measurement service.

When an application talks to a measurement service for the first time, all its measurements should be disabled. The measurement service will then enable the measurements it requires. If communication between an application and its measurement service fail, and the application then finds another measurement service, it should reset the configuration state to the defaults before sending the first "init" document

8.4.1 Errors

An info message containing an error description can be sent at any stage and can indicate one of the following:

1) That the message that was last received from you
   a) Contained Invalid XML (i.e could not be passed).
   b) Did not match the schema.
   c) Contained invalid data e.g. referencing of non-existing types or variables.

The error description will indicate whether the previous document was ignored or partially processed.
2) An internal error occurred inside the XAM library:
   a) A buffer overflow or out of memory quota condition arose and some data was lost.
   b) A null pointer exception or other non recoverable exception was caught.
   c) Some part of the library failed to load.

3) The application made incorrect use of the XAM library.

4) The data being sent to the XAM service from the library has exceeded the channels bandwidth:
   a) Some data was discarded.
   b) Some measurement requests has been removed as a consequence.

If the library cannot continue it will send the error description in a close message.

8.5 XAM Summary

XAM has been designed to provide a means for a programmer to instrument their code in such a way as to allow for intra-enterprise management without the need for proprietary libraries to be installed on the local machine. A measurement hierarchy has been defined which extends the range of measurements beyond those provided by ARM. The measurement hierarchy is a balance between simplicity and completeness. More measurement types could be added later with ease. The measurements defined are backwards compatible with the ARM measurements. Existing applications could be made to be XAM compliant by replacing the proprietary ARM library with a standard XAM replacement. XML has been used as the messaging format to allows for maximal interoperability and extensibility. The document dialog is simple with only six document types, each document in turn has a simple structure. XAM supports ganged messaging reports to reduce messaging overhead. Few assumptions have been made about the messaging middleware layer being used to transport the XML documents beyond reliable delivery. Possible transports include TCP, HTTP, E-Speak service bus, MQ series, RMI, CORBA, COM, etc. XAM is positioned to move ARM into the Internet services arena.

Service Framework Specification, Part II
Version 2.0

9 Negotiation

The negotiation framework aims to provide infrastructure that allows two or more independent entities to interact with each other over time to reach agreement on the parameters of a contract. It is aimed primarily, though not exclusively, as a means to reach trade agreements. It can be used both by automated entities, and by users via appropriate software tools.

Its value to *negotiation participants* is that it is a prerequisite to provide decision support or automation of the negotiation, and hence make the process more efficient. Furthermore, they can be confident that the basic rules of interaction in any negotiation are standardised, hence reducing the effort to automate many different kinds of business interactions. They are able to negotiate simple contracts, where only price is undetermined, and more complex contracts where many complex parameters depend on each other. Furthermore, the protocols provide the participants with trust guarantees, that no party has access to extra information or is able to forge false information.

Its value to *negotiation hosts* such as auction houses and market makers is that it provides a standard framework that all potential customers can use to interact with them. However, it does not require a specific market mechanism, so allows the host to decide on an appropriate one. It not only provides standard off-the-shelf market mechanisms such as the English auction, but also allows custom mechanisms to be implemented for particular special needs such as the FCC auction for auctioning bandwidth.

In this section, we present a framework for negotiation that can be used to model various negotiation protocols that are commonly used in practice. In addition, the framework is extensible and allows customization of existing negotiation protocols to suit specific needs.

9.1 General Negotiation Framework

We now briefly outline some of the high-level requirements of the negotiation framework.

1. The framework should be sufficiently formal that automated entities (e-services) can interact with it.
2. The framework should allow negotiation about simple and complex objects.
3. The framework should be sufficiently general that a variety of different market mechanisms such as 1-1 negotiation, combinatorial auctions, exchanges, etc., can be expressed as specific instances of it.
4. The framework should be built on appropriate security mechanisms and protocols that participants can do business in a trusted way.
5. The framework should allow, but not require, the existence of a third party to arbitrate a given negotiation, for example an auctioneer in an auction.
6. The framework should support existing ways people do business, as well as permitting more radical approaches in the future.

9.1.1 What Can One Negotiate?

Negotiation is the process by which two or more parties interact to reach an agreement. Usually, this will be about some business interaction such as the supply of a service in return for payment. However, the concepts described in this section are sufficiently general and they can be used to negotiate other forms of agreement.

To be able to negotiate with each other, parties must initially share an *agreement template*. This specifies the different parameters of the negotiation such as the product type, price, supply date etc. Some of the parameters will be constrained within the template while others may be completely open. The agreement template is decided at the matchmaking stage – matchmaking is exactly the process by which one party locates other parties with agreement templates compatible with their needs. This is achieved in the matchmaking phase by allowing offers for services to refer to agreement templates and contract templates that are relevant to the offer. A client searching for a service can retrieve the agreement templates associated with the offer when performing the lookup.

Depending on what parameters a party is willing to negotiate on, it will adopt more or less constrained agreement templates. For example, a party that is willing to negotiate nothing will only advertise a fully instantiated agreement template, with a fixed price. An example of this is the case where a company hosts a catalog that contains the descriptions and the prices of the goods in it and is unwilling to negotiate any part of the catalog. A party willing to negotiate features of a product, such as colour, as well as price and delivery date, will leave these parameters unconstrained.

The process of negotiation is the move from an *agreement template* to an *agreement* that the agreeing parties find acceptable. A single negotiation may involve many parties, resulting in several agreements between different parties and some parties who do not reach agreement. For example, a stock exchange can be viewed as a negotiation where many buyers and many sellers meet to negotiate the price of a given stock. Many agreements are formed between buyers and sellers, and some buyers and sellers fail to trade.

After an agreement is reached, it is necessary to formalise this and to determine how the business processes will interact with each other. This is the process of moving from agreement to *contract*, and will be dealt with in the contract formation section.

In this document, we assume that all agreements are between two parties. However, the majority of the protocols described generalise in a straightforward way to handle agreements between more than two parties.

9.1.2 The General Negotiation Protocol

When discussing negotiation, it is important to distinguish between the *negotiation protocol* and the *negotiation strategy.* The protocol determines the flow of messages between the negotiating parties – who can say what, when – and acts as the rules by which the negotiating parties must abide by if they are to interact. The protocol is necessarily public and open. The *strategy,* on the other hand, is the way in which a given party acts within those rules in an effort to get the 'best' outcome of the negotiation – for example, when and what to concede, and when to hold firm. The strategy of each participant is necessarily private, and hence an exploration of appropriate strategies falls outside the scope of the Service Framework Specification.

The Service Framework Specification offers a general negotiation protocol, by which service buyers and service sellers can interact. This general protocol can be specialised to give specific kinds of negotiation – such as catalog purchase sites, auctions, exchanges and multi-attribute one-to-one negotiation. Any entity that is able to participate in the general protocol will be able to participate in a specialised form of it, though its strategy may need to be altered.

There are 2 main roles in negotiation – *participant*, and *negotiation host.* The participants are those who wish to reach agreement, and usually they are subdivided into *service buyers* and *service sellers*. The negotiation host is the role responsible for enforcing the protocol and rules of negotiation. The host is often a third party outside the negotiation. In the case of an auction, the host is the auctioneer. In the case of an exchange, the host is the market provider. However, the host may also be a participant - In 1-1 negotiation or catalog provision, this is usually the case.

The general negotiation protocol consists of 5 main stages;

1. Potential participants request the negotiation host for admission to the negotiation. If they are accepted, they receive the agreement template and rules specifying how the negotiation takes place. For example, the admission rules for an auction site are different from the admission rules for an exchange, or a simple catalog based purchase site.
2. Negotiation takes place by participants making proposals. These proposals consist of constrained or instantiated versions of the agreement template. Participants may make proposals only according to the rules of the negotiation received at admission.
3. During negotiation, the host informs participants of the current status of the negotiation, either by sending them current proposals, or by sending some form of 'digest' such as, the current 'best' proposal. The content of these messages is determined by the rules of the negotiation.
4. In circumstances determined by the rules, the negotiation host identifies compatible proposals, and converts them into agreements.
5. In circumstances determined by the rules, the negotiation host closes the negotiation locale, and determines any final agreements.

9.2 Dictionary

Before proceeding further with the detailed description of the negotiation framework, we first introduce some of the key concepts and terminology required for formulating the negotiation framework.

Negotiation Host	Entity responsible for creation and enforcement of rules governing participation, execution, resolution and termination of a negotiation.
Participant	Entity participating in a negotiation by posting proposals according to the rules provided by the negotiation host.
Negotiation locale	Location where negotiation proposals are posted according to the rules enforced by the negotiation host.
Infrastructure provider	Provider of the underlying communications infrastructure of the negotiation locale.
Gatekeeper	Sub-role of negotiation host. Responsible for enforcement of policy governing admission to a negotiation.
Participant credentials	Information, with appropriate trust guarantees, about a participant's attributes, capabilities, etc.
Admission policy	Policy used for determining who is allowed to participate in a given negotiation.
Proposal	Specification of potential value ranges in the agreement template a participant is willing to accept.
Negotiation table	A negotiation locale with 1 buyer and 1 seller.
Auction room	A negotiation locale with 1 seller and many buyers.
Exchange floor	A negotiation locale with many buyers and many sellers.
Proposal validator	Sub role of negotiation host: responsible for ensuring that a proposal is well-formed.

Agreement template	A prototype specifying what is to be negotiated, and the possible values different parts of the prototype can range over.
Proposal well-formed-ness	A proposal is well formed within a given negotiation if all prototypes within it are subsumed by the agreement template.
Proposal expiration time	A parameter in a proposal defining when it no longer can be considered valid.
Negotiation rules	Rules determining the mechanism by which negotiation proceeds – who may make a proposal when, how existing proposals affect what proposals may be made.
Posting rule	Negotiation rule determining in what circumstances a participant may post a proposal.
Visibility rule	Negotiation rule specifying whom, among the participants, has visibility over a submitted proposal.
Display rule	Negotiation rule specifying if and how the referee notifies the participants that a proposal has been submitted – could either be by transmitting the proposal unchanged or by transmitting a summary of the situation.
Improvement rule	Negotiation rule specifying, given a set of existing proposals, what new proposals may be posted.
Withdrawal rule	Negotiation rule specifying if and when proposals can be withdrawn from negotiation, and policies over the time expiration of proposals.
Termination rule	Negotiation rule specifying when no more proposals may be posted (e.g. a given time, period of quiescence, etc.).
Protocol enforcer	Sub-role of negotiation host. Responsible for ensuring that participant’s proposals are posted according to the negotiation rules.

Agreement	A configuration of the attributes associated agreement template that is understood by the participants as being fully defined, together with the identification of two (or more?) participants.
Agreement formation	The conversion of a set of proposals, at least one pair of which intersect, into a set of agreements.
Agreement maker	The entity responsible for agreement formation.
Agreement formation rules	Rules responsible for determining, given a set of proposals at least one pair of which intersect, which agreements should be formed.
Tie-breaking rule	A specific agreement formation rule applied after all others.

9.3 Negotiation Protocol

In this section, we first specify the XML messages and conversations associated with the general protocol. We then outline how this general protocol can be extended to handle some more specific instances such as purchasing at a catalog site, english auctions, and continuous double auctions.

We consider each of the five main stages of a generic negotiation protocol in turn.

9.3.1 Admission

In general, the admission procedure will involve authentication and authorization. In addition, to authentication and authorization, there may be other admission rules that may be applicable. For example, in the case of auctions, one can envisage two general types of admission rules.

1. Explicit rules: The creator of the auction can specify the criteria for selecting buyers. Examples of rules that restrict entry are: 'Invitation,' prior registration, explicit qualification rules etc.
2. Implicit rules – *Entry fees* (paid to participate in the auction) and *reserve prices* (lowest acceptable bids) induce some of the potential buyers not to participate in a given auction. In this sense, such aspects of the auction design indirectly determine the set of actual participants.

When admission is successful, the negotiation host sends a message specifying an identifier for the particular negotiation, the agreement template, and a document containing the rules of negotiation.

The following XDR schema captures the top-level structure of the negotiation-accept document.

```
<element name = "negotiation-accept">
 <complexType content="elementOnly" model="open">
      <element name = "negotiationID">
      <element name ="agreement-template" >
      <element name ="negotiation-rules">
 </complexType>
</element>
```

The agreement-template is the starting point for the negotiations and specifies the template that the two parties fill out during the negotiation process in order to reach agreement. The negotiation rules specifies the constraints that govern the negotiation process.

For example, the agreement template for the paper example from the matchmaker chapter can have the following entries:

- A list of possible catalog identifiers
- A list of possible supplier names
- A list of part numbers used by the suppliers to identify the goods
- Whether the product desired by the client is new, used, etc…
- Plain text description of the product
- Desired size of the paper
- Dunn and Bradstreet code or other such codes.

Agreement template validation rule

The agreement template validation rule enforces that any proposal has to match the validation template. A RFQ (and RFQ lookup request) kind of proposal will specify enough detail to identify what it is that the buyer (seller) wants to trade upon. Each of the proposals are validated against the template and accepted or rejected accordingly. A proposal can specify more information such as maximum buyer price, but it's not requested to. Also an RFQ kind of proposal will have to specify who is the requestor. When entering the phase of purchase order formation, sellers will have a handle to identify the buyer and targeting them with their purchase order kind of proposal. Examples of valid RFQ and counter-RFQ kind of proposal are seen at page 157-161.
Purchase order documents have to conform to the agreement template too. The examples in pages 162-164 are valid and their validity is guarantee by the fact that they follow pre-agreed RFQs.

More formally, the following DTD captures the structure of the agreement template in the paper example:

```
<!DOCTYPE paperProposal [
      <!ELEMENT proposal (paper, price, owner, purchase-order>
      <!ATTLIST proposal type (buy | sell)>
      <!ELEMENT paper (catalog-identifier, supplier-name, supplier-part-num-
```

```
      ber, product-type, product-desc, paper-size, commodity-code-name, com-
      modity-code)>
      <!ELEMENT price (numerical-constraint, currency)>
      <!ELEMENT owner esurl>
      <!ELEMENT purchase-order>
      <!ELEMENT …>
]>
```

Now that the agreement template is defines, we turn our attention to defining the negotiation rules that govern the negotiation. In general, the negotiation rules can be divided into the following categories:

Posting rule
This rule determines who can submit proposals at what time.
Visibility rule
The proposal submitter can identify a sub-set of participants who are allowed to see the proposal. Notice that it's in the interest of the submitter to have their RFQ visible by as many sellers as possible, but when submitting a purchase order kind of proposal, the visibility had better be reduced, possibly to a single potential seller.
Information filtering (digest) rule
Each of the proposals that are submitted is transmitted unchanged to the participants who are allowed to see them (see visibility rule). No data structures are defined that present a summary of the proposals.
Time-bounding rule
Proposals are hold valid for a definite time, or up to agreement formation, whichever is first.
Improvement rule
Improvement rules are defined to have the purchase order formation phase converge more easily (e.g. cannot go back on a proposal etc).
Termination rule
Negotiation ends at a certain time (decided by the marketMaker)

9.3.2 Negotiation

Negotiation consists of the sending of a series of proposals to the negotiation host. Proposals may be sent at any time. If a proposal does not conflict with the negotiation rules, the host will accept and process the proposal appropriately. If the proposal does conflict with the rules, the host may simply ignore the proposal or it may explicitly reject the proposal. However, in general, the information about the acceptance or rejection of the proposal can be infered from the information about the negotiation that is made available by the negotiation host to the participants of the negotiation.

In general, the steps for negotiation are:

1. PERFORM *Initiate negotiation*

2. REPEAT
2.1 PERFORM *Submit proposal*
2.2 IF Agreement possible
2.2.1 PERFORM *Agreement formation*
2.3 ENDIF
UNTIL Termination
3. PERFORM *Finalize negotiation*

A proposal can have the form of an RFQ, or an RFQ lookup request, or a (possibly partial instantiation of a) purchase order.

1) Proposals can come in at any time when negotiation is open.
2) Market maker validates incoming proposals against the agreement template. In order for a proposal to be accepted, all the mandatory fields described in the agreement template will have to be present in the proposal. Notice that at this stage we don't make a distinction between RFQ/RFQ lookup and purchase orders.
3) If the proposal is rejected, use case ends.
4) The referee then validates the proposal against negotiation rules. In this case, the rule that is applied is the improvement rule. Every proposal submitted by a participant is checked against previous proposals of the same kind submitted by the same participant. For example, when working to form a complete purchase order, the improvement rule makes sure the new proposal represents a step towards the formation of a complete purchase order, with the given counter-part.
5) If the proposal is valid, the Market Maker applies the visibility rule, which allows a submitter to specify who among the participants is permitted to see the proposal. The Market Maker forwards then the proposal to the intended addressees.

The following schema outlines the top-level schema of a negotiation proposal.

```
<element name = "propose">
 <complexType content="elementOnly" model="open">
      <element name = "negotiationID">
      <element name ="proposal" >
  </complexType>
</element>
```

The proposal consists of a constrained form of the agreement template, specifying attribute values or value ranges which are acceptable to the proposer. For example, a buyer may constrain the price attribute to be less than $100.

If the negotiation rules allow, a participant may withdraw a previously submitted proposal by sending the following message;

```
<element name = "proposal-withdraw" >
 <complexType content="elementOnly" model="open">
    <element name = "negotiationID">
    <element name ="proposal" >
  </complexType>
</element>
```

Information Display

The negotiation host sends information about the current status of negotiations to all participants, as defined by the information display rules. By default, this consists of a set of current proposals. However, in certain circumstances, other information could be sent as well or instead. Also, some participants may receive different information from others

```
<element name = "negotiation-status">
 <complexType content="elementOnly" model="open">
    <element name = "negotiationID">
    <element name ="proposal" >
  </complexType>
</element>
```

Participants may also withdraw from negotiations by sending a negotiation-withdraw message. Often it is possible to withdraw at any time, though in some negotiations, the negotiation rules will only allow withdrawing in certain circumstances. (For example, in an auction, it is not possible to withdraw if you have the best current bid.)

```
<element = "negotiation-withdraw" >
 <complexType content="elementOnly" model="open">
    <element name = "negotiationID">
  </complexType>
</element>
```

The host responds to a withdraw message by removing all proposals from that participant.

Agreement Formation

In general, agreement formation consists of the following steps:

1) The Market Maker looks at the current sets of proposals to see whether agreements can be made. As said above, it's not enough to verify that two proposals are compatible (i.e. define non incompatible value for each of the attributes). In order for an agreement to be possible, all the mandatory fields in the purchase order have to be specified. In other words, the two proposals have to be compatible with each other and have to be validated against the *agreement formation template,* described above.

2) Tie-breaker rules do not apply to this particular kind of negotiation
3) The Market Maker creates the agreement, as described in the example at page 164 (Contract between enterprise A and enterprise B).
4) The Market Maker sends a copy of the agreement to both participants.

Proposals such as RFQ and RFQ lookup request can be matched (as they are seen by anyone declaring an interest), but that's not enough to get to agreement formation.
The agreement template extends the negotiation template with purchase order information (deliver to, pack list requirements, ship from, product quantity). In general, the agreement template extends the negotiation template with all the information that is necessary to form an agreement but is not necessary in the RFQ exchange phase.
Agreement is formed as soon as two proposals that comply with the agreement template, match with each other.
Tie-breaking rule
In this kind of negotiation, is not necessary to define a tie-breaking rule, because the purchase order kind of proposals are directed to counter-parts that have been singled-out already. It's responsibility of the submitter to make sure that they can fulfil any proposal that they submit.

In circumstances determined by the agreement formation rules, the negotiation host checks existing proposals for compatibility. If all required attributes of a proposal are specified and agreed between at least 2 parties, the host determines which parties actually will form an agreement, and notifies successful parties of the outcome by sending them agreements with all attributes specified. In cases where more than one agreement is possible, the negotiation host uses the agreement rules to determine which participants have priority.

```
<element name = "negotiation-agreement" >
 <complexType content="elementOnly" model="open">
    <element name = "negotiationID">
    <element name ="agreement">
  </complexType>
</element>
```

The agreeing parties use the agreement in the contract formation phase. (See section ??)

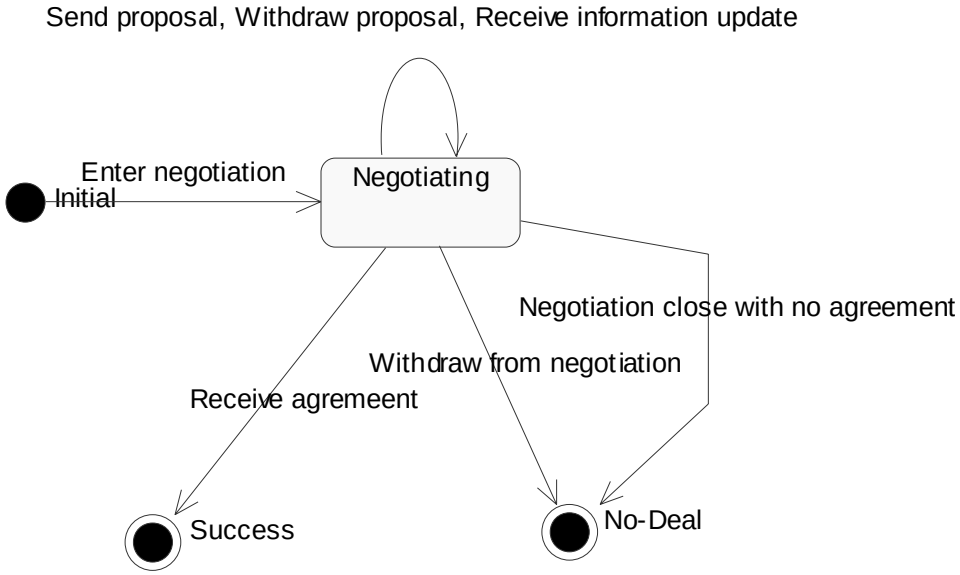
Locale Closing

The negotiation locale closes in circumstances determined by the negotiation rules. (eg when all participants have agreed or withdrawn, at a given time, after a period of quiescence, etc.) At this point, the host notifies all participants with a negotiation-close message, and any further proposals sent are ignored.

```
<element name = "negotiation-close" >
 <complexType content="elementOnly" model="open">
    <element type = "negotiationID">
  </complexType>
</element>
```

3.4.2.4 Protocol Conversation - Participant

In this section, we specify the conversation that a participant engages in when negotiating. In the appendix, we will then use CDL for defining the conversation.



Interactions

Interaction	Is initial step	Transitions
Initial	true	Enter Negotiation

Initial step of the conversation

Interaction	Is initial step	Transitions

Negotiating	false	Send Proposal; Receive Proposal; Receive Information Update; Receive Agreement; Withdraw From Negotiation; Negotiation close with no agreement

Negotiation is open

Interaction	Is initial step	Transitions
Success	false	

Negotiation terminated with success for the participant

Interaction	Is initial step	Transitions
No Deal	false	

Negotiation terminated with no deal for the participant

Transitions

Transition	SourceInteraction	DestinationInteraction	Triggering Document
Enter Negotiation	Initial	Negotiating	Admission request

The participant requests admission to negotiation and receives a response

Transition	From State	To State	Triggering Document
Send proposal	Negotiating	Negotiating	Negotiation Proposal

The participant sends a negotiation proposal

Transition	From State	To State	Triggering Document
Withdraw proposal	Negotiating	Negotiating	

The participant withdraws a negotiation proposal

Transition	From State	To State	Triggering Document
Receive information update	Negotiating	Negotiating	Information Update

The participant receives an update on the current state of the negotiation. The update might either be a negotiation proposal or a digest of the current state

Transition	From State	To State	Triggering Document
Negotiation close with no agreement	Negotiating	No-deal	Negotiation Close

The participant is informed that negotiation is closed (no agreement formed)

Transition	From State	To State	Triggering Document
Withdraw from negotiation	Negotiating	No-deal	Withdraw from Negotiation

The participant requests to withdraw from the current negotiation altogether and receives a response

Documents

Document	Schema	Root Element Name
Request Admission	Admission Schema	Admission-Request

Document	Schema	Root Element Name
Reply to Request Admission	Admission Reply Schema	Admission-Request-Reply

Document	Schema	Root Element Name
Negotiation Proposal	Agreement Template	Proposal

Document	Schema	Root Element Name
Negotiation Proposal Withdrawal	Withdrawal Schema	Withdraw

Document	Schema	Root Element Name
Agreement Notification	Agreement Template	Agreement

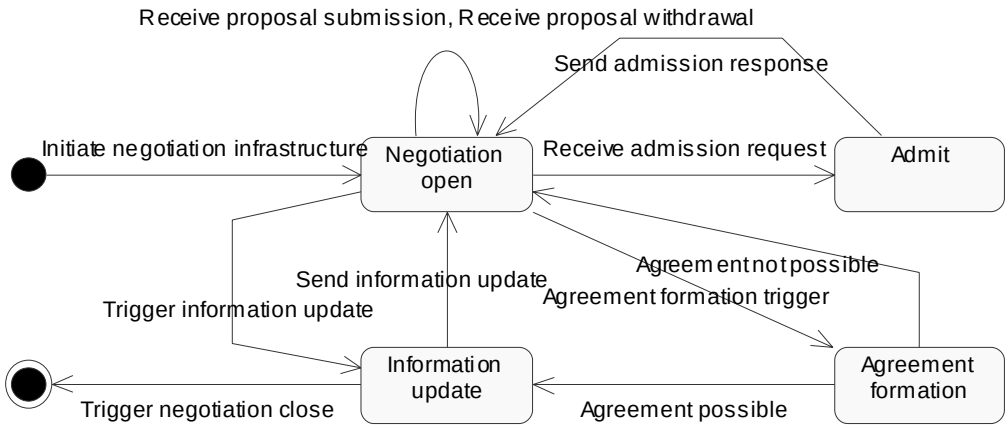
Document	Schema	Root Element Name

Negotiation Close	Negotiation Close Schema	Negotiation-close

Document	Schema	Root Element Name
Negotiation Withdraw Request	Negotiation Withdraw Schema	Negotiation-with-draw-request

Document	Schema	Root Element Name
Negotiation Withdraw Response	Negotiation Withdraw Schema	Negotiation-with-draw-response

3.4.2.5 Protocol Conversations – Negotiation Host



9.4 Examples

We now consider three simple examples of negotiation and provide a high-level description of how the various parts of the frmaework defined above work to enable the negotiation. The three examples are: simple shop front, simple english auction, and a continuous double auction.

Simple Shop Front

Actors:
shopkeeper: Participant, Referee
aBuyer: Participant

shopFront: Negotiation Locale

Use Cases:

1. Define admission policy
   shopkeeper decides policy – usually this will be the null policy: anyone is admitted.
2. Define agreement template
   shopkeeper decides on templates of goods it is willing to sell. These will be fully defined, specifying all details exactly, including price.
3. Define negotiation rules
   shopkeeper adopts standard ‘shopfront take it or leave it’ negotiation rules. These state that;
   a. A buyer may post a proposal at any time, irrespective of posted proposals by other buyers. A seller may post or withdraw proposals at any time.
   b. A buyer’s proposal must be an exact copy of the seller’s proposal (Except it is ‘buy’ rather than ‘sell’)
   c. Termination occurs when there are no seller proposals posted in the shopFront
4. Define agreement formation rules
   shopkeeper adopts standard shopfront agreement rule:
   a. Agreements are formed whenever a buyer posts a proposal identical to the seller’s proposal.
5. Negotiate:
6. Initiate Negotiation – as standard.
7. Submit Proposal
   shopkeeper submits proposals for all goods it sells. (If it expects high demand, it can place several identical proposals on the table for the same good.)
   If all proposals for a given good are accepted, and the shopkeeper still has more in stock, it resubmits identical proposals.
   A buyer submits a proposal, an identical copy of the shopkeeper’s proposal, when it wishes to purchase a given good.
8. Agreement Formation
   shopkeeper (in referee role) identifies valid buyer proposals, and sends agreements to

the buyers.

9. Finalize negotiation – as standard.

Single Item English Auction

Actors:
aSeller: Participant
aBuyer: Participant
auctioneer: Referee

auctionHouse: Negotiation Locale

Use Cases:

1. Define admission policy
   auctioneer and seller decide policy – this could be the null policy: anyone is admitted, or a list of invitees.
2. Define agreement template
   seller decides on template of good it is selling. This will be fully defined, specifying all details exactly, except for the price attribute, which will be open.
3. Define negotiation rules
   auctioneer adopts standard ‘english auction’ negotiation rules. These state that;
   a. A buyer may post a proposal at any time. The price field of the buyer’s proposal must be a certain increment above the value of all previously posted buyer proposals. The seller posts a single proposal at the start of the auction only.
   b. A buyer’s proposal must be an exact copy of the seller’s proposal, with price instantiated with its bid.
   c. Termination occurs at a fixed time.
4. Define agreement formation rules
   auctioneer adopts standard ‘english auction’ agreement rule:
   a. After termination, an agreement between the highest bidding buyer and the seller is formed.
5. Negotiate:
6. Initiate Negotiation – as standard.
7. Submit Proposal
   Initially, seller submits proposal for the good it wishes to auction. It may constrain the price to be above a certain reservation value.
   A buyer submits a proposal instantiated to it’s bid value.
8. Agreement Formation
   auctioneer identifies highest bidding buyer, and forms agreement between it and the seller. It notifies both parties.
9. Finalize negotiation – as standard.

Multiple Item Continuous Double Auction (aka Exchange)

Actors:
aSeller: Participant
aBuyer: Participant
marketMaker: Referee

exchangeFloor: Negotiation Locale

Use Cases:

1. Define admission policy
   marketMaker decides policy – this could be the null policy: anyone is admitted, or a list of invitees.
2. Define agreement template
   marketMaker decides on template of good it is selling. This will be fully defined, specifying all details exactly, except for the price attribute and quantity attribute, which will be open.
3. Define negotiation rules
   marketMaker adopts standard 'CDA (NYSE improvement)' negotiation rules. These state that;
   a. Buyers and sellers may post proposals at any time. The price field of a buyer's proposal must be above the value of all currently posted buyer proposals. The price field of a seller's proposal must be below the value of all currently posted seller proposals.
   b. Proposals must be a copy of the proposal template, with price and quantity instantiated to specific values.
   c. Termination occurs only when the auction ceases to be used.
4. Define agreement formation rules
   marketMaker adopts standard 'CDA' agreement rule:
   a. Agreement is formed between all overlapping buyers and sellers. The price is the midpoint of the overlap. Highest buyers and lowest sellers are satisfied first.
   b. When traders have different quantities, this may result in a single party having trades with several others (multiple agreements).
   c. In case of ties, earlier proposals have priority.
5. Negotiate:
6. Initiate Negotiation – as standard.
7. Submit Proposal
   Buyers and sellers submit proposals at any time, instantiating the price and quantity appropriately.

8. Agreement Formation

    Agreement formation occurs whenever there is an overlap between buyers and sellers, according to the rules above. Participants are notified of any agreements made.

9. Finalize negotiation – as standard.

Simplified templates and proposals

Agreement template

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE paperProposal [
      <!ELEMENT proposal (paper, price, owner, purchase-order>
      <!ATTLIST proposal type (buy | sell)>
      <!ELEMENT paper (catalog-identifier, supplier-name, supplier-part-num-
      ber, product-type, product-desc, paper-size, commodity-code-name, com-
      modity-code)>
      <!ELEMENT price (numerical-constraint, currency)>
      <!ELEMENT owner esurl>
      <!ELEMENT purchase-order>
      <!ELEMENT …>
]>
```

For an agreement to be formed, the two candidate proposals will have to specify all fields, and the information will have to be consistent.

Proposals

These, taken from the service framework specification would be valid proposals.
This is an RFQ for paper at price less than 22.95

```
<!--Usual E-Speak wrappers, -->
      <proposal type=buy>
            <paper>
                  <catalog-identifier>
                        02010001
                  </catalog-identifier>
                  <!—- other fields to specify paper -->
            </paper>
            <price>
                  <constraint xmlns:pv=”http://vocab-auth.com/paper-vocab.xsd>
                        <condition test=”price &lt; 22.95” />
                  </constraint>
            </price>
            <owner>
                  <esurl>es://server.enterpriseA.com:8880</esurl>
            </owner>
      </proposal>
```

This would a valid proposal when working towards a purchase order formation

```
<!--Usual E-Speak wrappers, -->
      <proposal type=buy>
            <! -- paper, price and owner as above>
            <purchase-order>
                  <deliver-to> etc… </deliver-to>
                  <! -- other purchase order fields -->
                  <! -- notice that price is out of this bracket! -->
            </purchase-order>
      </proposal>
```

10 Contract Specification

The central idea of the conceptual model for contracts is that the business relationship that motivates interactions that follow is captured explicitly in an electronic contract. An electronic contract is a document formed between the parties that enter into economic interactions. In addition to describing the promises that can be viewed as rights and obligations by each party, the e-contract will describe their supposed respective behavior in enough detail, so that the contract monitoring, arbitration and therefore enforcement is possible.

The terms and conditions appearing in the e-contract can be negotiated among the contracting parties prior to service execution. In this way businesses with no pre-existing relationships can bridge the trust gap, be able to strike deals and take them to completion.
Negotiation happens by incrementally agreeing on the terms and conditions that govern business interactions. Contract templates support this process by providing constraints that limit these interactions.

10.1 Lifecycle of the B2B interaction

Conceptually the lifecycle of the B2B interaction can be split into following three stages:

Pre-contractual phase - includes composition and exchange of the metadata related to the contract template also mechanisms for matchmaking of participants. In this phase the party types and the types of trade procedures that they will fulfil are identified (e.g.: shipper delivers the goods, buyer pays seller). The template typically has a number of free variables that are agreed upon in the next phase.

Contractual phase - during which concrete trading partners assume contract party roles and negotiate the details of their responsibilities. The negotiable variables of the contract (deadlines, order of actions) become fixed and the trade procedures bound to specific business protocols. The relationships between contract parties are created and are embodied in an agreement that after signing becomes fully-fledged e-contract.

Post-Contractual phase – during which actual delivery of contract consideration happens. Typically this constitutes of service or goods delivery, bill calculation, presentment and payment. The interactions between the parties may be monitored for their conformance to the terms of the contract.

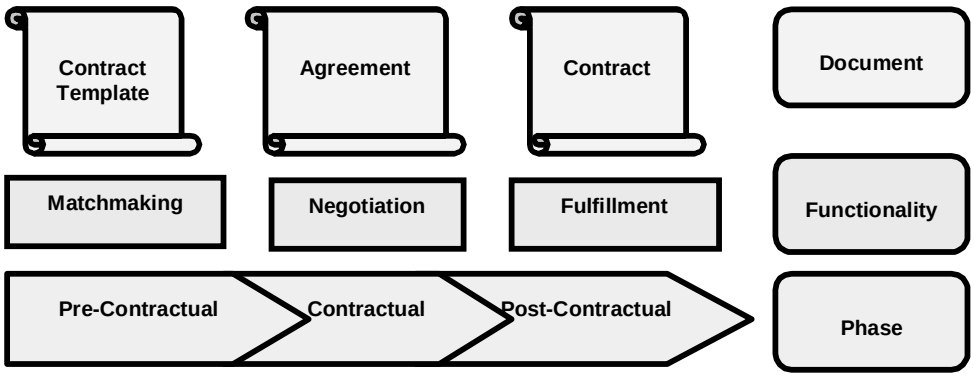


Figure 1. Contract lifecycle views.

The three phases can be described both from the functional point of view (what functionality is realized) and from the informational point of view (what document are generated and exchanged), as depicted in Figure 1. The functional view is as follows:

Matchmaking
Matchmaking is the process of putting service providers and service consumers in contact with each other. Matchmaking presupposes that services that want to be dynamically discovered will have to publish themselves, and services that want to find other services will have to send their request for matches. Some of the services advertising themselves for matchmaking will be simple end-providers, while others may be brokers, auction houses and marketplaces which offer a locale for negotiating with and selecting among many potential providers.

Negotiation
Negotiation is the incremental process by which participants get to agree on the terms and condition that will constitute the e-contract
The participants are seeking to strike the best deal for their stakeholders, who are the individuals or organizations that will then have to execute according to the rights and obligations agreed among participants. The rights and obligations will be expressed in an e-contract that will define terms and condition through which sale/purchase of goods and/or provision/consumption of services take place. Participants behave according to a set of rules, which determine what kind of trading takes place. This set of rules is called the market mechanism.
The market mechanism is embodied by the negotiation protocol that participants must comply with during negotiation.

Contract fulfilment
Once terms and conditions are defined, the fulfillment phase can begin. The fulfillment phase requires a communication infrastructure to be in place, which can enable execution and monitoring of the contractual service interactions. The communication between parties might be either

mediated or non-mediated, according to nature of the relationship.
The parties might identify a trusted third party who would arbitrate possible disputes arising from misbehavior or miscommunication during the execution of the service.

10.1.1 Realization of Conceptual Model

In order to realize any B2B interaction that we have outlined in the previous section an enabling infrastructure must exist with functionalities. Essentially, we partition the infrastructure into matchmaking service, market place services and market governance services and describe their responsibilities.

We stress that the roles in the conceptual model do not imply a specific architecture. Many scenarios are possible where the responsibilities associated with the Market Maker and Market Governor are fulfilled by the participants themselves (e.g. two parties that know each other may enter a 1-to-1 negotiation without the need for a Market Maker, but may want to ask for the Market Governor to monitor the transaction; alternatively a buyer may require services of a Market Maker to set up an auction but that of the Market Governor to automate the fulfillment which can happen offline).
In a similar way, the degree of automation and facilitation possible will highly depend on the specific business scenario.

Matchmaking Service
The matchmaking service realizes the matchmaking functionality as described above.
A matchmaking service receives a lookup request containing a service description, and returns information on how to reach compatible partners that have previously advertised their service offer (or request).
The service description specifies the nature of the service the initiator wishes to trade.
The returned information might also consist in a set of contract templates, along with information of where to go for negotiating the terms and conditions sketched in the templates and who will enforce the execution of contracts derived from them.

Marketplace
A marketplace is a virtual place where one or more buyers and one or more sellers (market participants) meet to trade on a (near -) homogeneous collection of goods or services.
The marketplace is run and regulated by a Market Maker. The Market Maker defines the entry policy to vet participants into the market place, based on participant’s identity and/or credentials. The Market Maker is also responsible for putting communication infrastructure in place to ensure that relevant messages are delivered to whom - and only to whom - they are addressed to, based on participant’s identity and/or credentials.
The participants are seeking to strike the best deal for their stakeholders, who are the individuals or organizations that will then have to execute according to the rights and obligations agreed among participants in the marketplace. The rights and obligations will be expressed in an e-con-

tract that will define terms and condition through which sale/purchase of goods and/or provision/ consumption of services take place. Participants behave according to a set of rules, which determine what kind of trading takes place. This set of rules is called the market mechanism.
The incremental process by which buyers and sellers get to agree on the terms and condition that will constitute the e-contract is called negotiation. The market mechanism of the marketplace is embodied by the negotiation protocol that participants must comply with during negotiation. The Market Maker is responsible for defining the market mechanism of the marketplace, for enforcing its negotiation protocol, for defining and enforcing admission rules to the marketplace and for providing the communication infrastructure for the marketplace. The role of the Market Maker can be further refined into a set of roles as follows:
Market Infrastructure Provider - Provider of the underlying communications infrastructure of the Marketplace

Gatekeeper - Responsible for enforcement of policy governing admission to a negotiation

Protocol Enforcer - Responsible for ensuring that during the negotiation process, participants follow the established negotiation rules

Further refinement of roles and their relationship will be exposed in the chapter on Negotiation.

Market Governance
The Market Governance is the trust holding infrastructure that enables e-contract fulfillment. The Market Governor plays the role of hosting and managing the Market Governance infrastructure. The Market Governor provides a collection of components, which provide the monitoring and communication functionality required for the contractual service interactions.

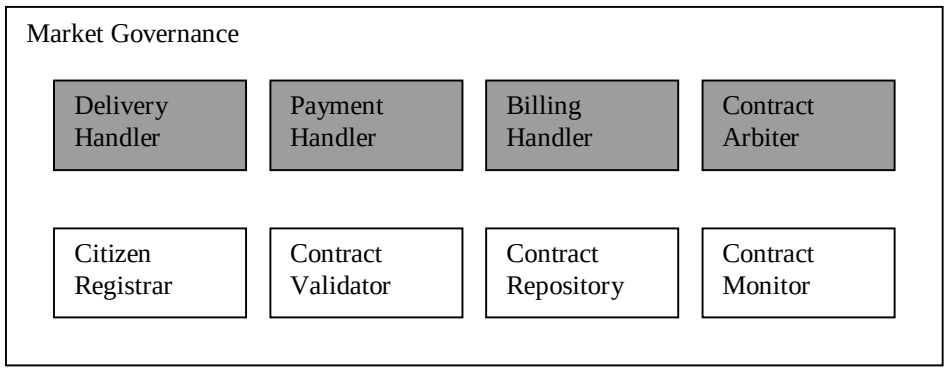


Figure 2. Market Governance. The components marked in white constitute a minimal set necessary for contract monitoring.

The Market Governor decides on the Market Governance services beyond the minimal infrastructure that it will provide. It then provides information to a Contract Template Drafter so that the metadata about the additional services can be a part of the template. A Market Maker responsible for the creation of a Market Place may enter into a conversation with the Market Governor. The outcome of this conversation is a contract template that is based both on the metadata related to the Market Governance services as well as Market Place and service specific metadata. The negotiation between the participants is based on that e-contract template that will eventually become e-contract.
Anyone who would like to be a party in an e-contract will have to be registered as a Citizen with the Market Governance. A participant contacts the Market Governor to obtain a citizenship. The Market Governor requests a minimal set of credentials from the member that will be sufficient to gain a citizenship within the Market Governance it represents. Once a participant fulfills these requirements it obtains a citizen ID that he will require in order to be a party to the e-contract. As citizens interact within the Market Governance the Market Governor may record their behavior in order to create new services like credit history. The citizen will obey the Market Governance arbitrage in the case of disputes.

Market Governance can be viewed as a collection of components shown in Figure 2. The set of minimal components allows for registration of citizens, validation and storage of contracts and monitoring of activities with respect to the contract:

Market Governor – generic role responsible for coordinating between the roles defined below. In the model where the business communication is mediated, Market Governor acts as a Market Mediator.

Contract Drafter - drafts the contract template. Registers the template with the Market Governance.

Contract Arbiter – invokes alternative procedure when the when a violation of the promise by the contract party is detected by the Contract Monitor.

Citizen Registrar – registers participants, checks validity of the identity credential, communication address and financial account information and stores this information as a citizen profile.

Contract Validator – checks if the contract document submitted is valid. This includes checking the signatures of contract parties with the Citizen Registrar and validating the consistency with the corresponding contract template.

Contract Repository –stores the contract and provides means to retrieve it. Publishes the contract template available for refinement and binding into. Performs validation of the contract instance against a corresponding contract template.

Contract Monitor - monitors the contract fulfilment with respect to a given participant providing information on the valid contract progression. Detects contract violation and passes the handling of violation to Contract Arbiter.

As the contract includes logical sections grouping the promises of the same kind i.e., delivery, billing, payment there is a need for specialised Handlers that will be able to format business documents conforming to the agreed protocol so that promises can be fulfilled.

10.2 Animation of Roles in the Lifecycle

Having introduced the main roles we animate them below taking the roles focusing on the functionality of the components fulfilling the roles.

Market Making

This phase takes place on the boundary of market place and Market Governance and aims at arriving at a contract template that would be a basis for the e-contract formation. The primary roles involved here are Market Maker and Market Governor.

1. When the Market Governance is created the Market Governor is responsible for assembling Market Governance services and financial risk analysis. Contract Drafter interacts with the Market Governance and drafts the contract template. The template is made public in the Market Governance for registered citizens.
2. When the Market Place is created the Market Maker contacts a number of Market Governors searching for a suitable contract template that would match the attributes present in the request for market place creation.
3. Market Maker applies a utility function to decide the most suitable contract template and advices the successful Market Governor. Market Governor may at that stage request Market Maker to become a citizen.

Negotiation

This phase takes place inside of the marketplace and exploits type and constraint information contained in the e-contract template. The aim of this phase is to refine contract template into a contract that can be signed by citizens.

1. The participant goes to a well known match-making service (for instance the E-Service Village), to find out which Market Makers are out there who host marketplaces where the services are being traded that he's interested in trading.
2. In order to do this, the participant specifies a set of constraints on the market that he's interested to take part in. The markets differentiate their offers – besides by the market mechanisms that they implement - through the variety of trust services and Market Governance services that they support. Not only does the participant’s request specify the attributes and constraints related to the kind of service that the participant is interested in trading, but also those related to contract fulfillment.

3. In response to the participant's request, The Matchmaking Service[1] presents the participant with a collection of contract templates that cover the criteria expressed by them in the previous step, together with information about the marketplace where they are being negotiated over. The selected contract templates also carry information about the Market Governance that will enforce the contract derived from them.
4. If there's an active marketplace where the selected contract templates are being negotiated over, the participant can require admission into it to negotiate over the selected contract templates.[2]

5. On verification of his credentials, the participant is vetted into the selected marketplaces and can start to negotiate in those. The negotiation consists of attempting to agree the specific instantiations of attributes in the contract templates, the price being one among them. The negotiation is regulated by the market mechanism that is enforced in the marketplace where the negotiation is taking place. The market mechanism defines the negotiation protocol that participants have to comply with and rules for matching bids and offers (see chapter on Negotiation).
6. When participants arrive at an agreement each participant contacts their corresponding citizen on behalf of whom they were negotiating. The participant presents the contract for verification and request signing. If the citizen is satisfied with the negotiated contract it signs it and passes it to the Market Governor who is responsible for the Market Governance where the contract templates can be enforced that where utilized in this contract's formation. The citizen may query the Market Governor about the state of the contract - has it received all required signatures, has it been validated and lodged, etc.

Fulfillment
This phase takes place inside the Market Governance and its aim is to fulfill the contract. The primary roles are Market Governor and service provider/consumer, who are citizens in the Market Governance. The Market Governor role is refined into Contract Verifier, Contract Monitor, and Contract Arbiter.
3.1 The Market Governor receives signed contracts that were formed in the market place and passes them to the Contract Verifier who is responsible for contract validation. If all signatures are present the Contract Verifier validates the contract against the contract template that was agreed upon with the Market Maker. He then verifies the signature verification of the citizens and the Market Maker placed on the contract. Finally, contract Verifier passes the contract to the Contract Repository.
3.2 The responsibility of the Contract Repository is to serialize the contract and instantiate appropriate object model for the Contract Monitor that determines the starting conditions for contract fulfillment.
3.3 The Contract Monitor performs monitoring of the execution of the e-contract based on the

1. This is not the only way of functioning for the Matchmaker.
2. If there's no active marketplace, following the request, a Market Maker can decide to create one and notify all the participants that had previously subscribed expressing interest for the type of services in question. Negotiation follows. Alternatively, a Market Maker might get back to the participant with an offer for subscription to be notified when the marketplace will be created. In this case negotiation doesn't follow right away.

activities of the contract parties. The abstraction level at which the contract arbiter operates is determined by the formal descriptions embedded in the contract. The primary responsibility of the Contract Monitor is to advise on the valid contract progression and detect violation of contractual promises. It delegates dispute resolution to the Contract Arbiter.
3.4 The Contract Arbiter notifies the parties indicated in the violated promise that a breach has occurred. It then invokes the alternative procedure associated with the violation.

10.3 Contract Instantiation Model

Contracts contain the expected course of actions that is to be followed in given situations as well as procedures that will be invoked when agreed actions are explicitly cancelled or not fulfilled. As it is not possible to specify exhaustively every possible violation that might occur the parties appoint a trusted third party that will invoke alternative procedures in situations not covered by the contract. There is a parallel between programming and construction of the contract in that the expected course of actions corresponds to the main program flow, the contract procedures for violation of non-fulfilment of actions corresponds to time-out procedure or handled exception and the contract violation and dispute resolution can be equated with an exception that needs to be handled outside of the main program flow.

It is important to understand that although contract contains the specification of procedures it does not define them. The expected flow of these procedures is specified by the service provider (in the metadata for service description). Contract on the other hand constrains and regulates these procedures.

This is achieved by specifying contractual situations that imply deontic states (such as permissions, prohibitions and obligations). Contract policies specify how the states can be discharged and so lead to further situations. In order to comply with the policy a business interaction is invoked by the party. External events such as expiration of a deadline can also bring about change in the situation. In this setting the situation and policy correspond to the label transition system where situations describe states and policies describe possible transitions.

Contract template

The information that is not subject to change across many types of contract instances is gathered in the contract template. On the basic level the contract contains the following information:

- Descriptions of parties to contract (roles, names, addresses)
- Period of validity of contract
- Nature of consideration being exchanged (services rendered, goods)
- Obligations of parties (time, quality, action sequence)
- Procedures invoked in case of violations (late payment penalty)
- Domain of the contract that guarantees the correctness and enforcement

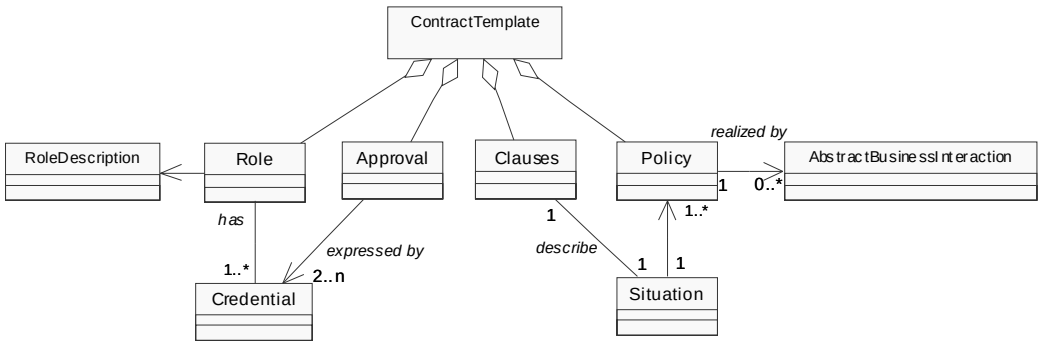


Figure 1. Model of the contract template.

This information is captured in the model shown in Figure 1.

Contract role. In the template the concrete instance of organization, institution, or person (in our model a Citizen) is abstracted into a contract role. As the negotiation starts the participants take specific roles that allows them to understand which commitments they are entering into. This binding occurs through association of the citizen ID with the role.

Role Description. Contains the description required for the identification of the contractual role. We require that the citizen ID is one of the attributes. Depending on the role there can be further attributes like address, VAT number, company registration number, name of registering body, etc.

Credential. Represents a document that is linked with the Role is a way that meets the security requirements described in the section 4. It is used to demonstrate and express the approval of the contractual commitments. All credentials are dated from a trusted time source.

Approval. Is a container for credentials of roles. When a credential instance is placed in the Approval container it signifies that the taker of contract role understands and agrees to the terms and conditions of the contract. The number of Credentials must be equal to the number of Roles for the contract to be valid.

Clauses. It is a structured textual description of contractual obligations of parties. Clauses are designed by an expert Contract Drafter.

Situation. Situation describes a state that implies that an obligation, permission or prohibition exists for specified contract role. Situation has associated with it a number of Policies that when fulfilled will lead to other Situations.

Policy. Describes how the contract party is expected to react to contractual situation. It may contain complex expressions and has associated with it a number of AbstractBusinessInteractions that when realized will fulfill the Policy.

AbstractBusinessInteractions. Is atomic in the context of the contract template. When negotiated upon in will be refined in a PrototypicalBusinessInteraction .

The model shown in the Figure 1 can be implemented as an XML document. In fact XML itself is not expressive enough for content-based validation but we use it here for the demonstration purposes. Below follows the fragment-by-fragment discussion of contract template components:

Contract role.

One of the primary elements created during the drafting phase are contract roles. They have an attribute for binding a specific citizen when contract formation commences

```
<element name="Role" content="elementOnly">
        <group order="sequence">
            <element name="RoleDescription" />
        </group>
        <attribute name="RoleRef" type="string"/>
</element>
```

Tag Name	Required?	Occurs	Description	Semantics
<Role>	Yes	Once or more	Role of the contract participants	Used for binding a specific citizen.

```
<element name="RoleDescription" content="elementOnly">
        <group>
            <element name="Contact" minOccurs='1' maxOccurs='*'/>
        </group>
        <attribute name="CitizenId" type="string" minOccurs="1"/>
</element>
```

Tag Name	Required?	Occurs	Description	Semantics
<RoleDescription>	Yes	Once	Describes the role by providing information as such as contact information and citizenId.	Used for binding a specific citizen.

```
Example:

<Role RoleRef="Buyer">
    <RoleDescription CitizenId="ctz45">
      <Contact>
        <Address>Filton Road BS34 8QZ UK</Address>
        <VAT_Number>VAT5678</VAT_Number>
        <Telephone>+44 344 555</Telephone>
```

```
        <Fax>+44 455 666</Fax>
        <CompanyRegistration>Reg6588</CompanyRegistration>
        <URL>www.A.com</URL>
      </Contact>
    </RoleDescription>
  </Role>
```

Approval and Credential

As soon as the roles are known the Approval section can be created. It links the credential with the role and indirectly with the citizen ID. We assume that all credentials will be time stamped when placed in the approval section.

```
<element name="Approval" content="elementOnly">
        <group order="sequence" minOccurs='1' maxOccurs='*'>
            <element name="Credential" minOccurs='1' maxOccurs='*'/>
        </group>
</element>
```

Tag Name	Required?	Occurs	Description	Semantics
<Approval>	Yes	Once or more	Links the credentials with the role. It is the place holder for signing the contract	The market governor uses it to validate the citizens credentials

```
<element name="Credential" content="elementOnly">
        <group order="sequence">
            <element name="CredentialType" />
            <element name="CredentialRef" />
        </group>
        <attribute name="CitizenId" type="string" minOccurs="1"/>
 </element>
```

Tag Name	Required?	Occurs	Description	Semantics

<Credential>	Yes		Set of citizen credetials.	Used by the market governor tho validate the citizenship of the contact participant.

Example:

```
<Approval>
    <Credential CitizenId="ctz987">
      <CredentialType>Certificate</CredentialType>
      <CredentialRef>www.verisign/ctz987</CredentialRef>
    </Credential>
    <Credential CitizenId="ctz45">
      <CredentialType>Certificate</CredentialType>
      <CredentialRef>www.balltimore/ctz987</CredentialRef>
    </Credential>
  </Approval>
```

Clauses

The contract clauses are drafted by the legal expert. The clause can be pointed to using the clauseID. Clauses are grouped into sections that can also be pointed to.

```
    <element name="Clause" content="elementOnly">
    <group order="sequence">
      <element name="Situation" minOccurs='1' maxOccurs='*'/>
      <element name="ClauseDescription" />
    </group>
    <attribute name="ClauseId" type="string" minOccurs="1"/>
  </element>
```

Tag Name	Required?	Occurs	Description	Semantics
<Clauses>	Yes	Once or more	Structured textual description of contractual obligations of parties	Clauses are designed by the Contract Drafter

Situation

Situation describes a state that implies that an obligation, permission or prohibition exists for specified contract role. Situation has associated with it a number of Policies that when fulfilled will lead to other Situations.

```
<element name="Situation" content="elementOnly">
        <group order="sequence">
            <element name="Policy" minOccurs='1' maxOccurs='*'/>
        </group>
        <attribute name="SituationId" type="string"/>
</element>
```

Tag Name	Required?	Occurs	Description	Semantics
<Situation>	Yes	Once or more	Describes a state that implies that an obligation, permission or prohibition exist for a specific role.	The situation is associated with a number of policies that when fulfilled will lead to other situations. It also used by the contract monitor.

Policy

Describes how the contract party is expected to react to contractual situation. It may contain complex expressions and has associated with it a number of AbstractBusinessInteractions that when realized will fulfill the Policy.

```
<element name="Policy" content="elementOnly">
        <group order="sequence">
            <element name="AbstractBusinessInteraction" />
        </group>
        <attribute name="PolicyId" type="string"/>
</element>
```

Tag Name	Required?	Occurs	Description	Semantics

<Policy>	Yes	One or more	Describes how the contract party is expected to react to contractual situation	The policy is associated to a number of abstract business interactions. The contract monitor also uses it.

AbstractBusinessInteractions

```
<element name="AbstractBusinessInteraction" content="elementOnly">
 <group>
  <element name="BusinessInteractionRef" minOccurs='1' maxOccurs='*'/>
 </group>
   <attribute name="InteractionId" type="string"/>
</element>
```

Tag Name	Required?	Occurs	Description	Semantics
<AbstractBusinessInteraction>	No	Once or more	Describes the abstract business interaction.	Used by the Process manager.

Example:

```
  <Clause ClauseId=1.1>
    <ClauseDescription>
      After Invoice has been received the Buyer
        has to pay the total amount within 7 days.
    </ClauseDescription>
    <Situation SituationId=1.1>
      <Policy PolicyId=1.1>
        <AbstractBusinessInteraction InteractionId="Invoice">
             <BusinessInteractionRef>
                     <href link=PurchaseOrderInteraction#InvoiceReceived/>
             </BusinessInteractionRef>
        </AbstractBusinessInteraction>
        <AbstractBusinessInteraction InteractionId="Payment">
             <BusinessInteractionRef>
                     <href link=PurchaseOrderInteraction#SendPayment/>
             </BusinessInteractionRef>
        </AbstractBusinessInteraction>
      </Policy>
```

```
</Situation>
```

10.4 Contract System Components

In this section we present in more detail the components involved in the contract performance stage and their relationship to other components of the framework like the Communication Manager and Process Manager. The later two components act as application integration interfaces between the Contract System and messaging and enterprise process management systems.

10.4.1 Functional View

We start with the abstract view focusing on the main functional blocks. Distributed Contract Monitor has the responsibility of maintaining consistency of views on the contract from the point of view of participating parties. It also has to determine which Situations are applicable with respect to the contract and what Policies apply to a given Situation. In effect the monitor determines set of valid transitions that will result in contract progression. He then passes the Policies for evaluation by the Decision Manger as shown in Figure 1. The responsibility of the Decision Manager is to select the transition based on the contract Policy as well as the enterprise private data (e.g.: state of the various business processes).

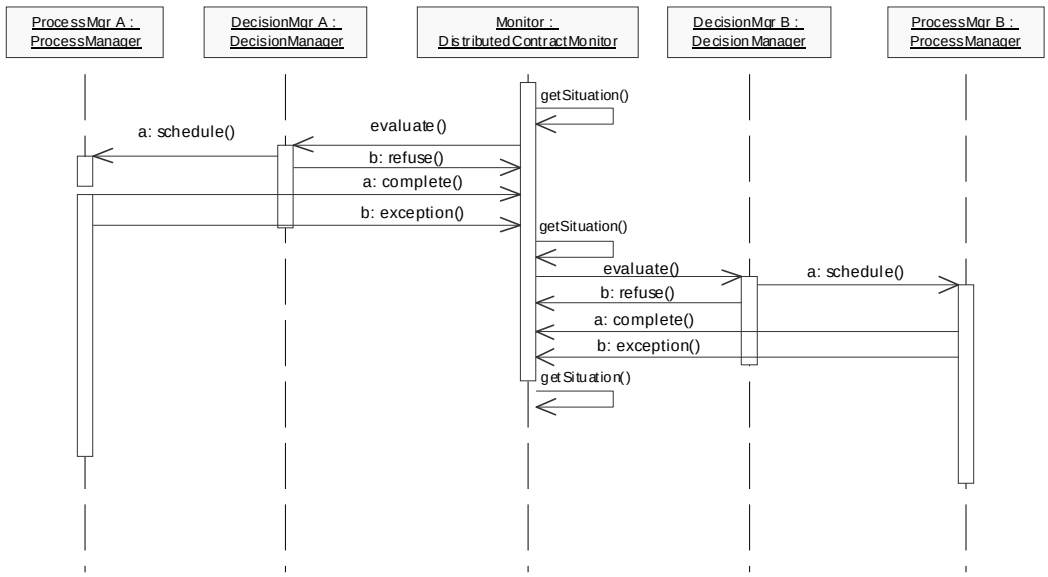


Figure 1. Functional view of contract performance. The expected behaviour is indicated with letter a, interactions marked with letter b indicate need for arbitration. The monitor coordinates execution of contract across enterprise domains.

Note that the Decision Manager may refuse to perform any of the transitions indicated by the Monitor (which would require Arbitration). Under normal circumstances the Decision Manager is able to select one of the transitions and schedules execution of a business process with the Process Manager. The Process Manager is responsible for accomplishment of the transition cho-

sen by the Decision Manager. He notifies the Monitor of completion of the process or an exception if fulfilment of the process is not possible.

The contract Monitor initiates the arbitration in case of an exception or determines the next Situation and applicable Policies if a transition has completed successfully.

We now turn our attention to the arbitration in case of the exception during the business interaction. As mentioned above the Contract Monitor maintains transactional integrity with respect to valid contract progression. It may receive an exception indicating that this progression in not possible. In this case it forwards the exception together with the data relating to the monitored interaction to the Arbiter. The Arbiter evaluates this data (audit log, business interaction stack trace, etc.) in conjunction with the contract metadata that allows him to determine the Situation that caused the exception. He then retrieves the Policy associated with disputed situation and on its basis schedules the arbitration process with the Process Manager. As the arbitration process executes exceptions may arise (exception from the exception) but under normal circumstances the process will complete. The Contract Monitor is notified and resumes the monitoring of further interactions. The Arbiter does not participate in these interactions until an exception is raised again.

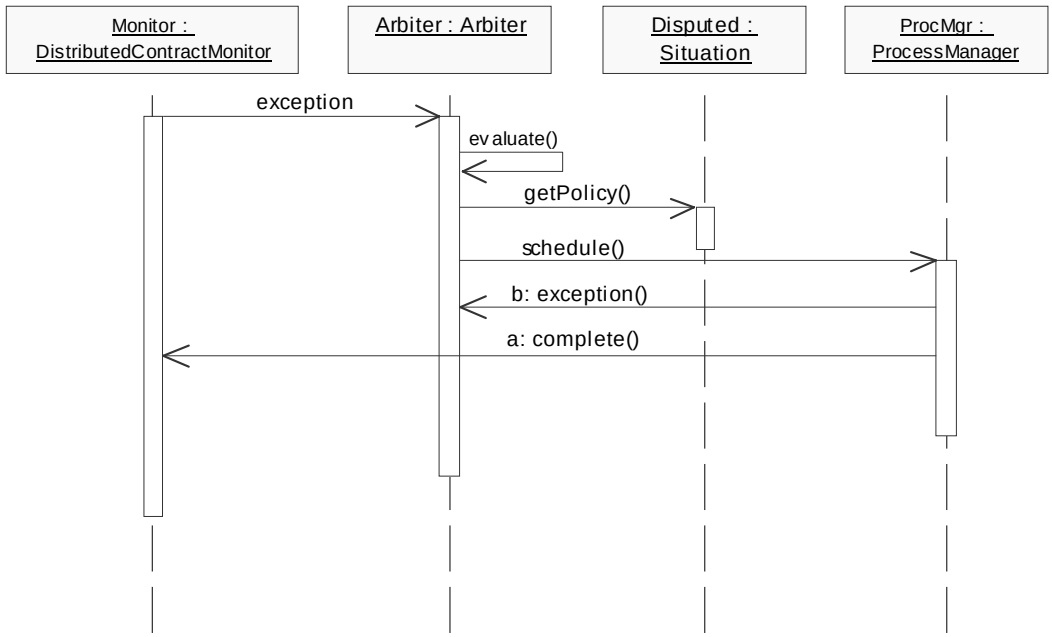


Figure 2. Processing of exceptions by the Arbiter. The exception carries information about the disputed situation that allows the arbiter to determine the arbitration process.

The functional analysis may lead to various designs and implementations. The implementation and a deployment model determine the specifics of the implementation of the Contract Monitor in the distributed environment, communication protocols and further third party roles such as message mediator.

10.4.2 Design: validation of the functional view

In our discussion below we focus on the post-contractual phase i.e., contract performance and the components and their interactions can be readily validated with the use cases given in section 2.6.3. We do not discuss the Use Case 1 on the contract signing as this activity occurs in the enterprise once the participants concluded the negotiation process. We just assume that one of the enterprises will lodge the signed contract with the Governance that can provide support for it.

The components involved in the contract lodging and set-up that are part of the Governance are shown in Figure 2. The Communication Manager receives a contract lodge request from the enterprise that is a citizen of the Governance. He processes the message and forwards the request and the contract to the ContractLifecycleManager. This component is responsible for maintaining the status of the contract and dispatching to further components. The contract is stored in the ContractRepository that and send for validation with the ContractValidator. This component determines the contract template on which the contract instance was based and retrieves it from the ContractRepository. It uses the information in the template to validate the contract instance. This includes checking that for each contract role there is a binding to the citizen and that the signatures placed on the contract are valid. It further validates any roles and constraints that the template may have with the negotiated contract instance. If the validation fails the contract is purged from the ContractRepository and the citizen that lodges it is notified. If the validation is successful the ContractLifecycleManager changes the state of the contract to LODGED and hands off the contract to ContractMonitor that commences the contract set-up. This involves determining the starting conditions for the contract and registering any triggers with appropriate components. Finally the lodging citizen and remaining parties to contract are notified that the contract has been accepted and when performance should begin.

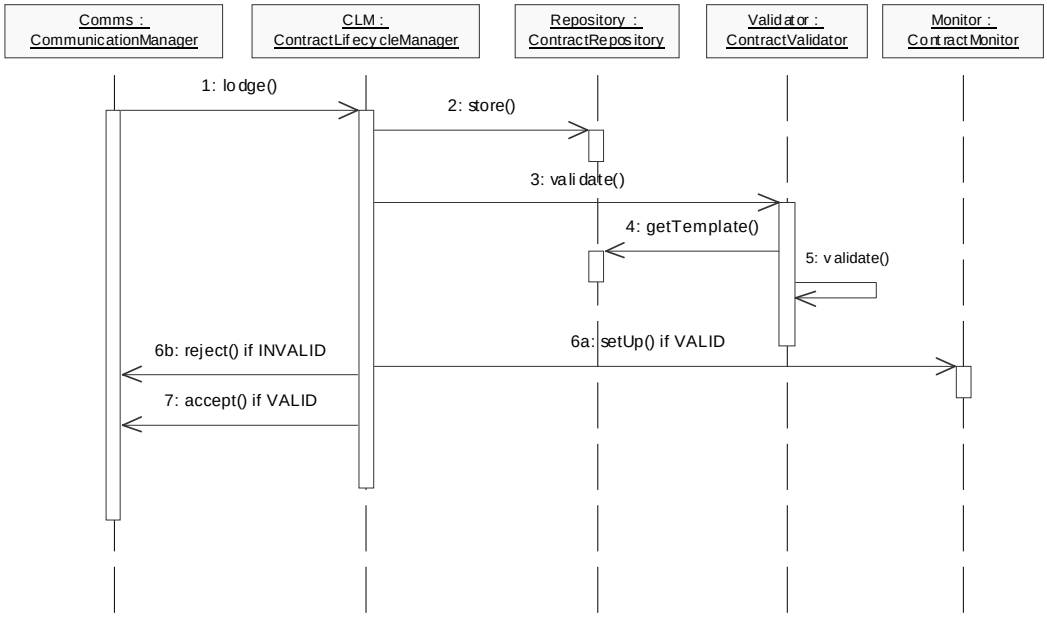


Figure 2. Market Governance: contract lodging and set up. Contract is processed and registered within the Governance. If it is valid all parties receive notification of its acceptance and starting conditions for performance.

In our model we assume that the Governance acting as a trusted third party is executing the validation and set-up that results in the parties ready to perform. The enterprises that are party to contract have identical components that allow for contract storing and processing as well as receiving of messages but the validation is outsourced to the Governance.

The main components that are involved in the contract performance stage in the enterprise are shown in Figure 3. We note that depending on the mediation model (i.e., outsourcing other processes than validation) the Mediator components may follow exactly the same interaction. This is discussed in more detail in section 2.1 and for the time being we turn our attention to the enterprises.

Once the contract has been set-up the goal of the enterprises is to interact with each other according to it. Valid contract progression is accomplished by responding to performance requests that arrive at the Communication Manager who forwards it to the ContractLifecycleManager. He determines if the status of the contract and if the contract is ready for execution passes the request to ContractMonitor that is be able to determine how to respond to it. ContractMonitor retrieves the Situation to which the request refers and obtains a policy that applies to it. It evaluates the policy taking into account enterprise private data and determines if the request should be accepted or rejected. In the case of acceptance it schedules a process that will fulfil the policy and deliver whatever output data necessary. Upon completion of the process the requesting party is notified of the acceptance of performance request and the process output data if necessary. The enterprise may also refuse the request. This may happen either because the enterprise chooses to breach the obligation taking into account other contractual commitments and enterprise private data or because the current Situation indicated by its Monitor does not warrant the Situation indicated in the request.

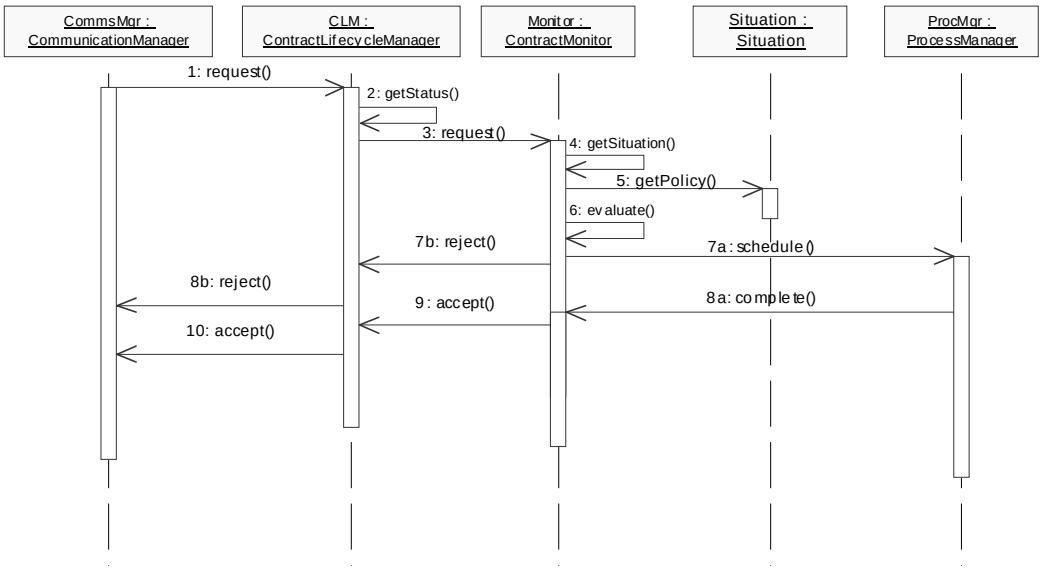


Figure 3. Enterprise: external performance requests. External party to contract requests performance of contractual obligation implied by the Situation. The Monitor determines contract Policies applicable to the Situation and evaluates them taking into account other enterprise private information (e.g.: inventory levels). The evaluation results either in acceptance (7a, 8a, 9, 10) or rejection (7b, 8b) of the contractual obligation implied by the Situation.

Apart from the external requests for the performance contract progression can occur as a result of events such as expiration of a deadline. This is shown in Figure 4 where the EventManager notifies ContractMonitor of the event. Typically ContractMonitor registers Situations with the EventManager so that events carry the information about the relevant Situation. As before the ContractMonitor retrieves the policy related to the Situation and performs the evaluation. If the performance of external party is required it sends a request to the Communication Manager that routs the message appropriately (where it is processed according to Figure 3). When the acceptance of the request is received the Monitor updates its information accordingly. If a rejection is received an exception would be thrown by the enterprise and an Arbiter run by a trusted third party will be involved.

The evaluation of the policy related to the event may also indicate need for internal performance in which case the ProcessManager is requested to schedule relevant process. When it completes the Contract Manager updates its information that may bring about new Situations.

We observe that the interactions described in Figure 3 and 4 imply a protocol (request, accept, reject, exception) that the enterprises follow when performing the contract. The protocol together with the Contract Monitor that each enterprise operates allows for coordinating contractual activities across the Internet. We observe that each enterprise has complete autonomy in how to manage its processes and weather or not perform its obligations.

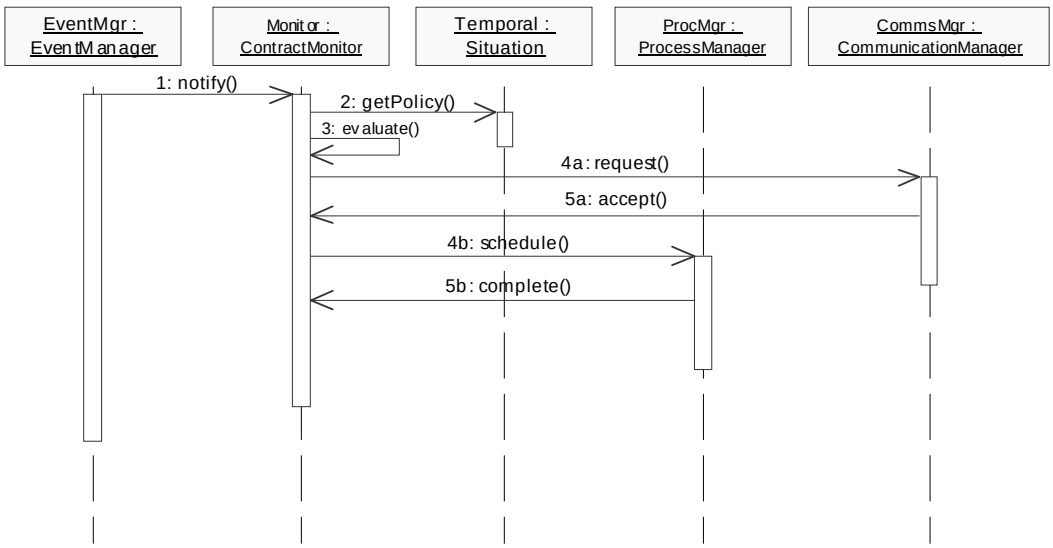


Figure 4. Enterprise: internal performance request. Contractual situations can arise when temporal events like expiration of a deadline happen. The valid contract progression may require performance of other parties (4a, 5a) or enterprise process to the executed (4b, 5b).

It is also worth pointing out that the request of performance shown in Figure 3 may be made by as a part of the dispute resolution when the Arbiter handling the exception requests performance of the dispute resolution step.

10.4.3 Mediation Model

We return now to the issue of the trusted third party and motivation for business interaction mediation. In our model Market Governance acts as a trusted third party and at a minimal functionality level provides identification of the participants and arbitration, as these are the fundamental capabilities required regardless of the business specialization of participants.

Arbitration

We noticed that the contract participant may choose to refuse performance when requested by other party. When this situation arises the party sends an exception to the Governance.

As shown in Figure 5 the exception is received by the Communication Manager in the Governance and if it relates to a contract instance that was lodged before is send to ContractMonitor for processing.

The Monitor retrieves the Situation indicated in the exception and sends it to the Arbiter component. He retrieves the contract template associated with the contract instance and retrieves the policy for Situation. It evaluates the situation, policy and information contained in the contract template to determine if arbitration is justified. If so the Arbiter sends the accept message to the parties to indicate that he accepts to arbitrate and schedules the arbitration process with the Proc-

essManager.

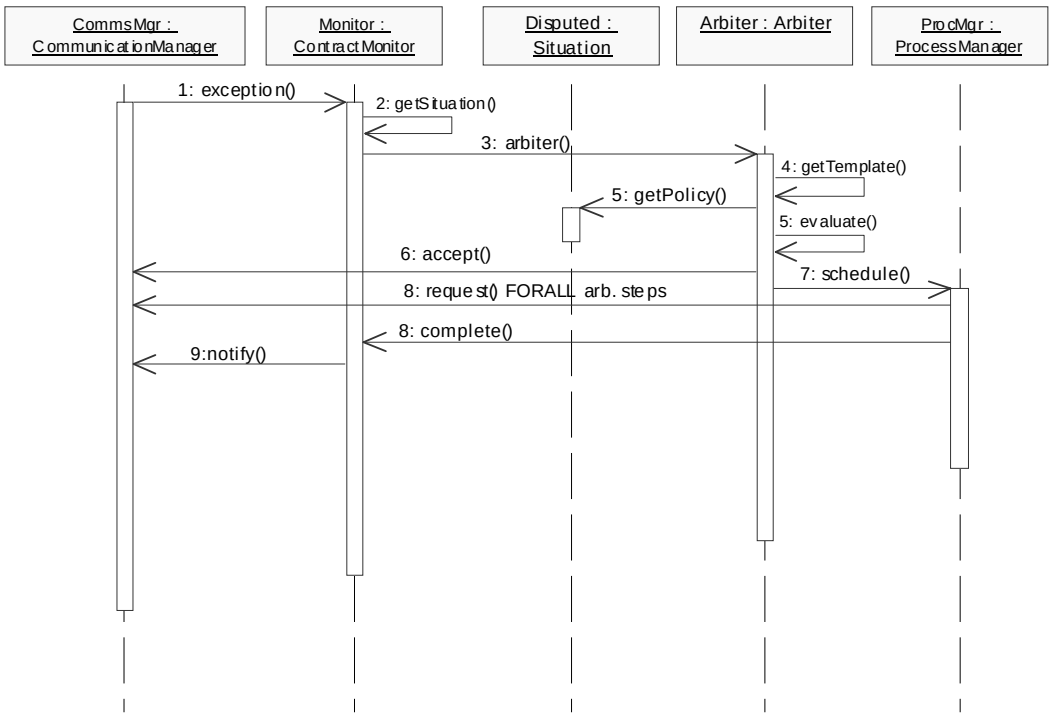


Figure 5. Market Governance: Arbitration. Contract participants send exceptions to Governance for arbitration. Based on the contract template the Arbiter chooses a resolution procedure and requests performance from participants.

As the process executes requests for performance are send to the parties who are expected to respond with the accept message (if one of the parties sends a refusal dispute settlement would have to occur outside the scope of the system) to the Governance. Once the arbitration process is completed the parties are notified and continue performing the contract without the involvement of the Governance.

10.5 Legal Status of Electronic Contracts

The global pervasiveness of the Internet and the ability to use it as a medium for economic interactions enables international trading for partners who otherwise would never have met before. This revolutionary development leads to business scenarios and relationships that have not been previously foreseen.

The work that is described in the previous sections postulates that in the near future contracts will be drafted, formed and enforced on-line. This requires that established law and governments bodies evolve towards the vision of electronic society and pass appropriate legislation. The world-wide legislation effort is limited to that of United Nations Commission on International Trade Law (UNCITRAL) aimed at establishing a global legal framework for EDI but most notable efforts so far can be observed in the European Union where the EU Directive on Electronic

Commerce has recently been passed. The directive lists a set of common rules for the conduct of electronic trade. It states that if the services provided by the service provider are lawful in a member state than the provision of these services in other member state is also lawful in other member state. This “country of origin” principle aims to protect the service providers and applies to B2B interactions. The rules laid out in the directive reinforce existing laws of Rome and Brussels Conventions that regulate which legislation is applicable (typically the law of the country most closely connected with the contract prevails). As many deployment models exist, determining the governing legislation domain can be complex but this issue can be resolved within the European law framework.

The consumer contracts are an exception where the “country of destination” principle prevails. It states that customers will be able to sue under local consumer protection laws if dissatisfied with the service provided in a different country. This issue is further laid out in the directive on the protection of consumers in the respect of distance contracts and specifies the rights for cancellation of the contract, deadlines and procedures for refunds in case of non-performance, arrangements for payment and delivery etc.

The Directive on Electronic Commerce with regards to electronic contracts also places certain information requirements that must clearly be given:

- An outline of different technical steps to follow to conclude contract;
- If the contract is to be filed or will be made accessible;
- Technical means for identifying and correcting errors prior to placing of the order;
- The languages in which contract can be concluded;
- Codes of conduct to which service provider subscribes.

A separate directive exists that stipulates validity of electronic signatures in electronic contracts (although exceptions exist) and their admissibility in legal proceedings. Consequently the electronic contract cannot be dismissed purely on the basis that it is not paper based.

The implication of the on going legislative process is that there are some requirements and guidelines that electronic contract system designers have to take into account. They mostly impact the design of the data structures and protocols involved in the contract formation and give high-level guidelines as to the contract content. We believe that the conceptual model that we have introduced takes into account the legal requirements indicated above.

10.6 Security Requirements

Trust and Security are critical concerns for interactions between business partners over the Internet. The contract framework should support the following security requirements in order to provide the maximum level of trust.

- **Authorization**: The contract should be protected against improper, unauthorized access. Only citizen who are signatories of the contract should be allowed to do so.
- **Integrity**: The contract should be protected against modifications as soon as at least one signature has been placed on the contract.

- **Non-repudiation**: The contract participants (citizens) should not be able to deny their contractual commitments.
- **Confidentiality and Privacy**: The contract content should only be revealed to the signatories and/or trusted third parties that they have nominated.
- **Authentication**: the system must be able to identity contract participant.
- **Valid signatures**: The contract has to have valid signatures of the contract parties so as to have and expression of commitment to contractual obligations.
- **Transport security:** the communication channel between the contract participant should be secure.
- **Secure Messaging**: the business messages between the contract parties should be done in a secure manner.
- **Secure storage**: the contract document should be stored in a storage that will guarantee that it can be retrieved and processed even after a long time has elapsed.

References

[1] SOAP: Simple Object Access Protocol: http://www.w3.org/TR/SOAP/

[2] WSDL: Web services Definition Language: http://msdn.microsoft.com/xml/general/wsdl.asp

Appendix A: Schemas and Example Documents

This section contains various schemas that are part of the SFS specification but that have not been put in the main text in order not to explode the main text. It also contains additional examples.

XML Schema of the CDL language

```
<?xml version="1.0" ?>

<schema name="ConversationDefinition"
        xmlns="http://www.w3.org/1999/XMLSchema"
        xmlns:dt="http://www.w3.org/1999/XMLSchema-datatypes"
        xmlns:conv="http://www.e-speak.net/schema/conversation"
    targetNamespace="http://www.e-speak.net/schema/conversation"
        elementFormDefault="unqualified"
       attributeFormDefault="unqualified" >

<element name="InboundXMLDocument"  dt:type="conv:XMLDocumentType"/>

<element name="OutboundXMLDocument" dt:type="conv:XMLDocumentType"/>

<element name="InboundXMLDocuments" >
   <complexType>
      <element ref="conv:InboundXMLDocument" minOccurs="1" maxOccurs="unbounded" />
   </complexType>
</element>

<element name="OutboundXMLDocuments" >
   <complexType>
      <element ref="conv:OutboundXMLDocument" minOccurs="1" maxOccurs="unbounded"/>
   </complexType>
</element>

<group name="ReceiveSendDocumentGroup">
   <sequence>
      <element ref="conv:InboundXMLDocuments" minOccurs="1" maxOccurs="1"/>
      <element ref="conv:OutboundXMLDocuments" minOccurs="1" maxOccurs="1"/>
   </sequence>
</group>

<group name="SendReceiveDocumentGroup">
   <sequence>
      <element ref="conv:OutboundXMLDocuments" minOccurs="1" maxOccurs="1"/>
      <element ref="conv:InboundXMLDocuments" minOccurs="1" maxOccurs="1"/>
   </sequence>
</group>

<group name="ReceiveDocumentGroup">
   <sequence>
      <element ref="conv:OutboundXMLDocuments" minOccurs="1" maxOccurs="1"/>
   </sequence>
</group>

<group name="SendDocumentGroup">
      <sequence>
         <element ref="conv:InboundXMLDocuments" minOccurs="1" maxOccurs="1"/>
      </sequence>
   </group>
```

```
<element name="Interaction" >
   <complexType>
      <attribute name = "id" type="dt:ID" required="yes" use="required" />
      <attribute name = "initialStep" type="dt:boolean" required="yes"/>
      <attribute ref="conv:interactionType" required="yes" use="required" />
      <choice>
         <group ref="conv:ReceiveSendDocumentGroup" />
         <group ref="conv:SendReceiveDocumentGroup" />
         <group ref="conv:ReceiveDocumentGroup" />
         <group ref="conv:SendDocumentGroup" />
     </choice>
   </complexType>
</element>

<element name="Transition" >
   <complexType name="TransitionType">
      <attribute name = "transitionType" use=”default” value=”Basic”>
        <simpleType base=”dt:string”>
          <dt:enumeration value=”Basic”/>
          <dt:enumeration value=”Default”/>
          <dt:enumeration value=”Exception”/>
        </simpleType>
      </attribute>
      <element ref="conv:SourceInteraction" minOccurs="1" maxOccurs="1"/>
      <element ref="conv:DestinationInteraction" minOccurs="1" maxOccurs="1"/>
      <element ref="conv:TriggeringDocument" minOccurs="0" maxOccurs="1"/>
   </complexType>
</element>

<element name="Conversation" >
   <complexType>
      <attribute name = "name" dt:type="dt:string" required="yes" use="required" />
      <attribute name = "initialInteraction" dt:type="dt:IDREF" required="yes"
            use="required" />
      <element ref="conv:ConversationInteractions" minOccurs="1" maxOccurs="1"/>
      <element ref="conv:ConversationTransitions" minOccurs="1" maxOccurs="1"
                required="yes"/>
   </complexType>
</element>

<element name="ConversationInteractions" >
   <complexType>
      <element ref="conv:Interaction" minOccurs="1" maxOccurs="unbounded"/>
   </complexType>
</element>

<element name="ConversationTransitions" >
   <complexType>
      <element ref="conv:Transition" minOccurs="1" maxOccurs="unbounded"/>
   </complexType>
</element>

<element name="SourceInteraction" >
   <complexType dt:content="empty" >
      <attribute name = "href" dt:type="dt:IDREF" use="required" />
   </complexType>
</element>
```

```
<element name="DestinationInteraction"  >
    <complexType dt:content="empty" >
        <attribute name = "href" dt:type="dt:IDREF" use="required" />
    </complexType>
</element>

<element name="TriggeringDocument"  >
    <complexType dt:content="empty" >
        <attribute name = "href" dt:type="dt:IDREF" use="required" />
    </complexType>
</element>

<attribute name="interactionType" required="yes" use="required" >
    <simpleType base="dt:string">
        <enumeration value="SendReceive"/>
        <enumeration value="ReceiveSend"/>
        <enumeration value="Receive"/>
        <enumeration value="Send"/>
    </simpleType>
</attribute>

<complexType name="XMLDocumentType">
    <attribute name = "id" dt:type="dt:ID" required="yes" use="required" />
    <attribute name = "hrefSchema" dt:type="dt:uriReference" required="no"
              use="optional" />
</complexType>

</schema>
```

Schema of the ServiceDescriptor document

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2000/10/XMLSchema" elementFormDefault="quali-
fied">
<element name="AccessPoint">
<complexType>
<attribute name="urlType" type="string" use="required"/>
</complexType>
</element>
<element name="AuthInfo">
<complexType/>
</element>
<element name="BindingTemplate">
<complexType>
<sequence>
<element ref="AccessPoint"/>
<element ref="tModelInstanceDetails"/>
</sequence>
</complexType>
</element>
<element name="BindingTemplates">
<complexType>
<element ref="BindingTemplate"/>
</complexType>
</element>
<element name="BusinessService">
<complexType>
<sequence>
<element ref="Name"/>
<element ref="Description"/>
<element ref="BindingTemplates"/>
<element ref="CategoryBag"/>
</sequence>
<attribute name="serviceKey" type="string" use="required"/>
<attribute name="businessKey" type="string" use="required"/>
</complexType>
</element>
<element name="CategoryBag">
<complexType>
<element ref="KeyedReference"/>
</complexType>
</element>
<element name="ConversationDefinition">
<complexType>
<attribute name="id" type="string" use="required"/>
<attribute name="href" type="string" use="required"/>
</complexType>
</element>
<element name="ConversationDefinitions">
<complexType>
<element ref="ConversationDefinition" maxOccurs="unbounded"/>
</complexType>
</element>
```

```
<element name="ConversationProcessMap">
<complexType>
<element ref="Map"/>
</complexType>
</element>
<element name="Description">
<complexType/>
</element>
<element name="KeyedReference">
<complexType>
<attribute name="tModelKey" type="string" use="required"/>
<attribute name="keyName" type="string" use="required"/>
<attribute name="keyValue" type="string" use="required"/>
</complexType>
</element>
<element name="Map">
<complexType>
<attribute name="conversation" type="string" use="required"/>
<attribute name="process" type="string" use="required"/>
</complexType>
</element>
<element name="Name">
<complexType/>
</element>
<element name="ProcessDefinition">
<complexType>
<attribute name="id" type="string" use="required"/>
<attribute name="href" type="uriReference" use="required"/>
</complexType>
</element>
<element name="ProcessDefinitions">
<complexType>
<element ref="ProcessDefinition" maxOccurs="unbounded"/>
</complexType>
</element>
<element name="ServiceDescriptor">
<complexType>
<sequence>
<element ref="ServiceProperty"/>
<element ref="ServiceVariable"/>
<element ref="ConversationDefinitions"/>
<element ref="ProcessDefinitions"/>
<element ref="ConversationProcessMap"/>
</sequence>
</complexType>
</element>
<element name="ServiceProperty">
<complexType>
<sequence>
<element ref="AuthInfo"/>
<element ref="BusinessService"/>
</sequence>
</complexType>
</element>
```

```
<element name="ServiceURI">
<complexType/>
</element>
<element name="ServiceURL">
<complexType/>
</element>
<element name="ServiceVariable">
<complexType>
<sequence>
<element ref="ServiceURL"/>
<element ref="ServiceURI"/>
</sequence>
</complexType>
</element>
<element name="tModelInstanceDetails">
<complexType>
<element ref="tModelInstanceInfo"/>
</complexType>
</element>
<element name="tModelInstanceInfo">
<complexType>
<sequence>
<element ref="Description"/>
<element ref="ConversationDefinition"/>
</sequence>
<attribute name="tModelKey" type="string" use="required"/>
<attribute name="lang" type="string" use="required"/>
</complexType>
</element>
</schema>
```

Example of a ServicePropertySheet document

```
<ServiceDescriptor xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="C:\sriram\sdlcdlpdl\SDLSchema.xsd">
- <ServiceProperty>
<AuthInfo />
- <BusinessService serviceKey="serviceKey" businessKey="businessKey">
<Name />
<Description />
- <BindingTemplates>
- <BindingTemplate>
<AccessPoint urlType="http" />
- <tModelInstanceDetails>
- <tModelInstanceInfo tModelKey="tModelKey" lang="cdl">
<Description />
<ConversationDefinition id="ConvDef1" href="#ConvDef1" />
</tModelInstanceInfo>
</tModelInstanceDetails>
</BindingTemplate>
</BindingTemplates>
- <CategoryBag>
<KeyedReference tModelKey="" keyName="" keyValue="" />
</CategoryBag>
</BusinessService>
</ServiceProperty>
- <ServiceVariable>
<ServiceURL />
<ServiceURI />
</ServiceVariable>
- <ConversationDefinitions>
<ConversationDefinition id="ConvDef1" href="http://www.convserver.com/CDL1.xml" />
<ConversationDefinition id="ConvDef2" href="http://www.convserver.com/CDL2.xml" />
</ConversationDefinitions>
- <ProcessDefinitions>
<ProcessDefinition id="ProcDef1" href="http://www.convserver.com/PDL1.xml" />
<ProcessDefinition id="ProcDef2" href="http://www.convserver.com/PDL2.xml" />
</ProcessDefinitions>
- <ConversationProcessMap>
<Map conversation="ConvDef1" process="ProcDef1" />
</ConversationProcessMap>
</ServiceDescriptor>
```

Example of the XML body of a ConversationDefinitions message

The ConversationDefinition is a document returned by the service in the ServiceConversationIntrospection conversation. The following example shows the business payload of a ConversationDefinitions message from a weather service that realizes the two mandatory introspection conversations plus one weather inquiry conversation :

```
<SOAP-ENV:Body>
   <ConversationDefinitions xmlns="http://www.e-speak.net/schema/conversation">
      <Conversations>
         <ConversationDefinition>
           <Conversation name="ServicePropertyIntrospection"
                      xmlns="http://www.e-speak.net/schema/conversation"
                      initialInteraction="#ServicePropertyIntrospection" >
             <!— We omit here the rest of this conversation definition, the CDL
            has been presented earlier on this document and would come here --!>
           </Conversation>
         </ConversationDefinition>
         <ConversationDefinition>
           <Conversation name="ServiceConversationIntrospection"
                      xmlns="http://www.e-speak.net/schema/conversation"
                      initialInteraction="#ServiceConversationIntrospection" >
            <!— We omit here the rest of this conversation definition, the CDL has
            been presented earlier on this document and would come here --!>
           </Conversation>
         </ConversationDefinition>
         <ConversationDefinition>
           <Conversation name="SimpleWeatherInquiry"
                      xmlns="http://www.e-speak.net/schema/conversation"
                      initialInteraction="#DefaultInquiry" >
            <ConversationInteractions>
               <Interaction id="DefaultInquiry" interactionType="ReceiveSend" >
                 <InboundXMLDocuments>
                   <InboundXMLDocument id="GetDefaultWeather"
                             hrefSchema="GetDefaultWeather.xsd" />
                 </InboundXMLDocuments>
                 <OutboundXMLDocuments>
                   <OutboundXMLDocument id="WeatherDefaultStatement"
                             hrefSchema="DefaultWeather.xsd" />
                 </OutboundXMLDocuments>
               </Interaction>
            </ConversationInteractions>
           </Conversation>
         </ConversationDefinition>
      </Conversations>
      <Documents>
         <Schema name="GetServiceProperty">
            <!— We omit here the rest of this document schema, it has been pre-
            sented earlier on this document and would come here --!>
         </Schema>
         <Schema name="ServicePropertySheet">
            <!— We omit here the rest of this document schema, it has been pre-
            sented earlier on this document and would come here --!>
         </Schema>
         <Schema name="GetConversationDefinitions">
            <!— We omit here the rest of this document schema, it has been pre-
            sented earlier on this document and would come here --!>
         </Schema>
```

```
            <Schema name="ConversationDefinitions">
               <!— We omit here the rest of this document schema, it has been pre-
               sented earlier on this document and would come here --!>
            </Schema>
            <Schema name="GetDefaultWeather">
               <element name="DefaultWeatherRequestParameters"
                              type="sfsc:DefaultWeatherInputParametersType"/>
               <complexType name=" DefaultWeatherInputParameters" >
                  <element name="ZIPCode" type="string"/>
                  <element name="CountryCode" type="string" />
               </complexType>
            </Schema>
            <Schema name="WeatherDefaultStatement">
               <element name="DefaultWeatherStatementContent"
                              type="sfsc:DefaultWeatherStatementContentType"/>
               <complexType name=" DefaultWeatherStatementContentType" >
                  <element name="ZIPCode" type="string" minOccurs="1" maxOccurs="1"/>
                  <element name="CountryCode" type="string" minOccurs="1" maxOccurs="1"/>
                  <element name="DateTime" type="string" minOccurs="1" maxOccurs="1"/>
                  <element name="TemperatureC" type="string" minOccurs="0" maxOccurs="1"/>
                  <element name="WindKnots" type="string" minOccurs="0" maxOccurs="1"/>
                  <element name="RainChance" type="string" minOccurs="0" maxOccurs="1"/>
                  <element name="Humidity" type="string" minOccurs="0" maxOccurs="1"/>
               </complexType>
            </Schema>
         </Documents>
      </ConversationDefinitions>
   </SOAP-ENV:Body>
```

Schemas and DTD of the Vocabulary Definition Language

DTD for vocabulary definitions

This DTD is the DTD for schema definitions available at the W3S site with the following changes:
- References to "export" have been eliminated - the assumption is that all parts of a schema are "exportable" and when a schema is imported, all parts of the schema are imported.
- Related to the above, many of the attributes of the import element were deleted, as was the component element (all related to export control).
- It is assumed that element refinement will not be initially supported; the "refines" element has been removed from the DTD.
- The content model for archetype has been simplified by the removal of "mixed" and "any" from content; model is either "open" or "closed" with "closed" as the default.
- DatatypeQual has been eliminated: for clarity, it is preferred that restrictions on datatype values be accomplished via a named datatype.
- A modelGroup is a subset of archetype and has been eliminated. Perhaps modelGroupRef should be retained but as "archRef." For now it is eliminated.
- A constraint element has been added to indicate the constraints that any offer description can conform to.

It should be noted that Schemas that describe Vocabularies may be representable as a limited subset of general purpose Schemas. Suppose that there are no aggregate datatypes, then an Attribute Property is simply described as an XML element whose content is "textOnly" (no elements). This assumption of no aggregate datatypes leads to a very simple Schema for Vocabularies, described by the following simple DTD. Because of the simplifying assumption regarding the absence of aggregate datatypes, the essential part of the schema definition is that an element contains only data, and not other elements. The simplified DTD is as follows:

```
<!ELEMENT schema ((import*, include*,  datatype*,
            (element)*, constraint* ))>
<!ATTLIST schema
            targetNS    CDATA       #IMPLIED
            version     CDATA       #IMPLIED
    xmlns       CDATA
'http://www.w3.org/XML/Group/1999/09/23-xmlschema/'
            model       (open|closed) 'closed' >

<!ELEMENT import EMPTY>
<!ATTLIST import
            schemaAbbrev     NMTOKEN      #REQUIRED
            schemaName       CDATA        #REQUIRED>

<!ELEMENT include EMPTY>
<!ATTLIST include
```

```
            schemaName      CDATA       #REQUIRED>


<!-- The datatype element is defined in XML Schema: Part 2: Datatypes -->
<!-- for publication:
   http://www.w3.org/1999/05/06-xmlschema/datatypes.dtd -->
<!ENTITY % xs-datatypes PUBLIC 'datatypes'
              'datatypes.dtd' >
%xs-datatypes;

<!ENTITY % xs-datatypes PUBLIC 'datatypes'
              'datatypes.dtd' >
%xs-datatypes;

<!ELEMENT element (type, constraint*)>
<!ATTLIST element
        required NMTOKEN
        datatype NMTOKEN
         ismultivalued NMTOKEN
         isreference NMTOKEN
        name          NMTOKEN
        defvalue  NMTOKEN
        schemaAbbrev    NMTOKEN     #REQUIRED
        schemaName      CDATA       #REQUIRED>
```

Schema for vocabulary definition

An alternative method of defining the vocabulary is using schemas. The following schema defines the schema for vocabularies.

```
<?xml version='1.0'?>
<!-- XML Schema for E-speak Vocabulary properties -->
<!-- location of this schema is at:
      http://www.e-speak.net/Schema/core/E-speak.vocab.xsd
  -->
<schema xmlns='http://www.w3.org/1999/XMLSchema'
        xmlns:ES-CORE='http://www.e-speak.net/Schema/vocab/'
        targetNamespace='http://www.e-speak.net/Schema/vocab/'>

  <element name=”schema” minOccurs='0' maxOccurs='1' >
    <type>
          <element  name=”element”  minOccurs='1'  maxOccurs='unbounded'   con-
tent='elementOnly'>
        <type>
          <attribute name='name' type='string' use='required'/>
          <attribute name='datatype' type='string' use='required'/>
                    <attribute  name='required'  type='boolean'  use='default'
value='false' />
```

```
                <attribute  name='ismultivalued'  type='boolean'  use='default'
value='false' />
                  <attribute  name='isreference'  type='boolean'  use='default'
value='false' />
            <attribute name='default' type='string' use='optional' />
   <element ref=”constraint” minOccurs=’0’ maxOccurs=’unBounded’ content=’ele-
mentOnly’/>
        </type>
       </element>
   <element ref=”constraint” minOccurs=’0’ maxOccurs=’unBounded’ content=’ele-
mentOnly’/>
    </type>
  </element>
</schema>
```

Contract XML Schema

```
<?xml version="1.0"?>
<schema>
    <element name="Contract" content="elementOnly">
        <group order="sequence">
            <element name="Role" minOccurs='1' maxOccurs='*'/>
            <element name="Approval" minOccurs='1' maxOccurs='*'/>
            <element name="Clause" minOccurs='1' maxOccurs='*'/>
            <element name="Duration" />
        </group>
        <attribute name="ContractId" type="string"/>
        <attribute name="Type" type="string"/>
    </element>

    <element name="Role" content="elementOnly">
        <group order="sequence">
            <element name="RoleDescription" />
        </group>
        <attribute name="RoleRef" type="string"/>
    </element>

    <element name="Approval" content="elementOnly">
        <group order="sequence" minOccurs='1' maxOccurs='*'>
            <element name="Credential" minOccurs='1' maxOccurs='*'/>
        </group>
    </element>

   <element name="Clause" content="elementOnly">
        <group order="sequence">
            <element name="Situation" minOccurs='1' maxOccurs='*'/>
            <element name="ClauseDescription" />
        </group>
        <attribute name="ClauseId" type="string" minOccurs="1"/>
```

```
</element>

<element name="Duration" content="elementOnly">
    <group order="sequence">
        <element name="Start" />
        <element name="End" />
    </group>
</element>

<element name="Start" content="textOnly">
</element>

<element name="End" content="textOnly">
</element>

<element name="Credential" content="elementOnly">
    <group order="sequence">
        <element name="CredentialType" />
        <element name="CredentialRef" />
    </group>
    <attribute name="CitizenId" type="string" minOccurs="1"/>
</element>

<element name="RoleDescription" content="elementOnly">
    <group>
        <element name="Contact" minOccurs='1' maxOccurs='*'/>
    </group>
    <attribute name="CitizenId" type="string" minOccurs="1"/>
</element>

<element name="Contact" content="elementOnly">
    <group order="sequence">
        <element name="Address" />
        <element name="VAT_Number" />
        <element name="Telephone" minOccurs='1' maxOccurs='*'/>
        <element name="Fax" minOccurs='1' maxOccurs='*'/>
        <element name="CompanyRegistration" />
        <element name="URL" />
    </group>
</element>

<element name="Address" content="textOnly">
</element>

<element name="VAT_Number" content="textOnly">
</element>

<element name="Telephone" content="textOnly">
```

```
    </element>

    <element name="Fax" content="textOnly">
    </element>

    <element name="CompanyRegistration" content="textOnly">
    </element>

    <element name="URL" content="textOnly">
    </element>

    <element name="CredentialType" content="textOnly">
    </element>

    <element name="CredentialRef" content="textOnly">
    </element>

    <element name="Situation" content="elementOnly">
        <group order="sequence">
            <element name="Policy" minOccurs='1' maxOccurs='*'/>
        </group>
        <attribute name="SituationId" type="string"/>
    </element>

    <element name="Policy" content="elementOnly">
        <group order="sequence">
            <element name="AbstractBusinessInteraction" />
        </group>
        <attribute name="PolicyId" type="string"/>
    </element>

    <element name="AbstractBusinessInteraction" content="elementOnly">
        <group>
           <element name="BusinessInteractionRef" minOccurs='1' maxOccurs='*'/
>
        </group>
        <attribute name="InteractionId" type="string"/>
    </element>

    <element name="BusinessInteractionRef" content="textOnly">
    </element>

  <element name="ClauseDescription" content="textOnly">
    </element>

</schema>
```

Example of an Offer

For example, consider the following offer to sell paper put in by acme paper company. The contents of the offer are similar to the suggested supplier catalog entries as specified by commerceOne. Essentially, the following offer represents an offer to sell letter sized paper whose list price is $ 22.95. It also outlines some of the other attributes of the paper being sold. The offer also indicates that it is visible to users who have a credit rating of at least a 'B', or HP employees. Furthermore, the creator of the offer has asked the matchmaker to send it an even in the case when a user orders paper worth $100,000 or more.

```
<offer type="sell">
      <!-- Description of this offer --!>
   <offer-description>

<!-- This namespace points to the schema that the paper element in the document con-
             forms to --!>
         <paper xmlns:pv="urn:schemas-paperXchange-com:paperOffer">

<!--  The catalog identifier is used for the description. Such an identifier is a
             suggested field in the catalog entry for publishers in the commerceOne
             marketsite --!>
       <catalog-identifier>02010001</catalog-identifier>

       <!-- Name of supplier --!>
       <supplier-name>acme office products</supplier-name>

          <!-- Part number --!>
       <supplier-part-number>347005</supplier-part-number>

          <!-- If product is new, used, etc --!>
       <product-type>new</product-type>

       <!-- Plain text description of product --!>
       <paper-description>
       <product-desc>acme copy plus paper 8 ½" x 11" </product-desc>

       <!-- Paper attributes- width, length, etc. --!>

       <paper-size>letter</paper-size>
       <width>8.5</width>
       <length>11</length>
              </paper-description>
             <paper-price currency="usd">
       <list-price>22.95</list-price>
       <list-uom>cs</list-uom>
       </paper-price>
       <classificationCode classType="UN/SPSC">
<!—- The standard encoding scheme that is used to identify the product. UN/SPSC
             refers to the UN, Dunn and BradStreet encoding scheme for all commodi-
             ties--!>
   <!—- The 10 digit commodity code according to UN/SPSC encoding for paper. Note
             that this code is an example. The actual code for paper may be differ-
             ent. --!>
       1234567689
```

```
            </ClassificationCode>
          </paper>
    </offer-description>

    <!-- More information about the product --!>
    <offer-info>

    <!-- Location of product overview document --!>
        <overview>http://www.acme-paper.com/347005/ov.pdf</overview>
    <!-- Location of technical specs about the product --!>
        <tech-spec>http://www.acme-paper.com/347005/ts.pdf</tech-spec>
        <drawing>http://www.acme-paper.com/347005/drawing.gif</drawing>
    </offer-info>
    <private-info>
        <upper-limit>100000</upper-limit>
    </private-info>
       <!-- ID number identifying the matchmaker-contract --!>
    <matchmaker-contract>
        <contractID>324fdgfd65765</contractID>
    </matchmaker-contract>

    <!-- Owner of this offer --!>
    <owner>
        <esurl>es://server.acme-paper.com:80/paperseller</esurl>
    </owner>
    <owner-rules>
       <security>
          <condition test="$user/profile/credit_rating &gt; 'B'" />
          <condition test1="$user/profile/certs contains HP-emp-cert"/>
</security>

       <!-- Min and Max order information --!>
        <min-order-quantity>10</min-order-quantity>
        <max-order-quantity>1000</max-order-quantity>
        <lot-size>10</lot-size>
        <event-rules>
            <arule>
                <pre-condition>
        greater($query/quantity * $offer/offer-description/paper/list-price,
                                $offer/private-info/upperlimit)
                </pre-condition>
                <action>
                    $self.notify($query)
                </action>
            </arule>
        </event-rules>
    <owner-rules>
    <owner-interfaces>

    </owner-interfaces>

     <!-- How long this offer is good for --!>
    <availability>
        <availabilityTuple>
            <offerRef>offer-description/paper</offerRef>
            <time-line>6 months</time-line>
            <quantity>1000</quantity>
        </availabilityTuple>
<availabilityTuple>
```

```
            <offerRef>offer-description/paper</offerRef>
            <time-line>1 month</time-line>
            <quantity>100</quantity>
         </availabilityTuple>
      </availability>
   <!-- ID number for this offer --!>
      <offerID>
         3484390548
      </offerID>
   </offer>
```

The description sent in by the supplier describes the product that is being put for sale.

Relationship to UDDI

We now turn our attention to describing how the structures that we have outlined in the Match Making and Offer chapters can be used to encode the kind of matchmaker that the UDDI specification outlines. UDDI defines the notion of a registry for four things:
1. BusinessEntity
2. BusinessService
3. BindingTemplate
4. TModels

Essentially, each entity is registered with a site Provider, and the site provider also provides a key for any entity registered with it. This key is similar to the offerID that the matchmaker returns as the result of registering any offer. We now outline how the UDDI functionality can be captured using the mechanisms outlined above.

1. save_*** APIs:
Essentially, the save_*** APIs in the UDDI programmers specification can be viewed as registration requests where the offer that is registered corresponds to the appropriate schema. That is, the saveBusinessEntity API call can be translated to an offer registration request where the contents of the offer description conform to the BusinessEntity vocabulary. Since any schema can form the basis of a vocabulary, we can define the vocabularies for the entities.

For example, the business entity vocabulary looks as follows, (the schema for business entity in UDDI):

```
<element name = "businessEntity">
    <type content = "elementOnly">
    <group order = "seq">
       <element ref = "discoveryURLs" minOccurs = "0" maxOccurs = "1"/>
      <element ref = "name"/>
      <element ref = "description" minOccurs = "0" maxOccurs = "*"/>
      <element ref = "contacts" minOccurs = "0" maxOccurs = "1"/>
     <element ref = "businessServices" minOccurs = "0" maxOccurs = "1"/>
     <element ref = "identifierBag" minOccurs = "0" maxOccurs = "1"/>
    <element ref = "categoryBag" minOccurs = "0" maxOccurs = "1"/>
   </group>
   <attribute name = "businessKey" minOccurs = "1" type = "string"/>
   <attribute name = "operator" type = "string"/>
   <attribute name = "authorizedName" type = "string"/>
   </type>
</element>
```

The vocabulary definition looks as follows:

```
<?xml version='1.0'?>
<schema targetNS="urn:uddi-org:businessEntity.xsd"
       xmlns="http://www.e-speak.net/Schema/vocab">
<element name = "businessEntity">
```

```
     <type content = "elementOnly">
     <group order = "seq">
        <element ref = "discoveryURLs" required=”no” datatype=”URL” multi-val-
ued=”yes”/>
        <element ref = "name" required=”yes” datatype=”string”/>
        <element ref = "description" required=”no” datatype=”string”/>
        <element ref = "contacts" />
       <element ref = "businessServices" datatype=”businessService:serviceKey”
isreference=”yes” ismultivalued=”yes”/>
       <element ref = "identifierBag" datatype=”tModelIdentifier:reference”
isreference=”yes” ismultivalued=”yes”/>
      <element ref = "categoryBag" datatype=”category” ismultivalued=”yes”/>
    </group>
    <attribute name = "businessKey" required=”yes” type=”string” />
    <attribute name = "operator" type = "string"/>
    <attribute name = "authorizedName" type = "string"/>
    </type>
</element>
</schema>
```

Any business entity can register itself with a matchmaker. In such a situation, the matchmaker contract field can represent the contract between the business entity and the entity that runs the matchmaker.

Note that the notion of vocabularies in e-speak also supports the notion of multi-valued attributes. Essentially, one can define a named element and indicate that it takes on multiple values. This allows the e-speak attributes to mimic the Bag construct in UDDI. Therefore, the identifierBag in the schema above is can be modeled as an attribute named identifierBag that can take on multiple values. In other words, it is a multi-valued attribute.

A similar technique allows e-speak matchmakers to accept other UDDI entity descriptions such as Business Services, binding templates, and T-models. We briefly sketch the vocabulary definitions for the other concepts in UDDI.

```
<?xml version='1.0'?>
<schema targetNS="urn:uddi-org:businessService.xsd"
   xmlns=”http://www.e-speak.net/Schema/vocab”>
<import schemaAbbrev="be" schemaName="urn:uddi-org:businesEntity.xsd"/>

<element name = "businessService">
<type content = "elementOnly">
<group order = "seq">
        <element ref = "name" required=”yes” datatype=”string”/>
       <element ref = "description" required=”no” datatype=”string”/>
       <element ref = "bindingTemplates" isreference=”yes” datatype=”bind-
ingTemplate:templateKey” />
       <element ref = "categoryBag" datatype=”category” ismultivalued=”yes”/>
</group>
<attribute name = "serviceKey" minOccurs = "1" type="string"/>
<attribute name = "businessKey" type = "be:reference"/>
```

```
</type>
</element>
<constraint>
   <goal>
      <batom>
         <wdConstraint>
             <atom>
                 <predicate>
                     isValid
                  </predicate>
                  <args>
                     <term>
                     <variable>
                         $businessService/businessKey
                     </variable>
                    </term>
                    <term>
                       $businessService/bindingTemplates
                    </term>
                   </args>
              </atom>
         </wdConstraint>
   </batom>
   </goal>
</constraint>
</schema>


<?xml version='1.0'?>
<schema targetNS="urn:uddi-org:tModel.xsd"
   xmlns=”http://www.e-speak.net/Schema/vocab”>
   <element name = "tModel">
      <type content = "elementOnly">
          <group order = "seq">
             <element ref = "name" datatype=”string” required=”yes”/>
             <element ref = "description" datatype=”string” required=”no”/>
             <element ref = "overviewDoc" datatype=”URL” required=”yes”/>
            <element ref = "identifierBag" datatype=”tmodel:identifier:refer-
ence” isreference=”yes” ismultivalued=”yes”/>
            <element ref = "categoryBag" datatype=”category” ismultival-
ued=”yes” required=”no”/>
         </group>
         <attribute name = "tModelKey" minOccurs = "1" type = "string"/>
         <attribute name = "operator" type = "string"/>
         <attribute name = "authorizedName" type = "string"/>
      </type>
   </element>
<constraint>
   <goal>
      <batom>
         <wdConstraint>
             <atom>
                 <predicate>
```

```
                    isValid
                  </predicate>
                  <args>
                     <term>
                     <variable>
                         $tModel/identifierBag
                     </variable>
                    </term>
                 </args>
              </atom>
         </wdConstraint>
   </batom>
   </goal>
</constraint>
</schema>

<?xml version='1.0'?>
<schema targetNS="urn:uddi-org:bindingTemplate.xsd"
  xmlns:vocab="http://www.e-speak.net/Schema/vocab">
<import schemaAbbrev="tm" schemaName="urn:uddi-org:tModel.xsd"/>
<import schemaAbbrev="bs" schemaName="urn:uddi-org:businessService.xsd"/>

   <element name = "bindingTemplate">
      <type content = "elementOnly">
         <group order = "seq">
            <element ref = "description" datatype="string" required="no"/>
            <group order = "choice">
            <element ref = "accessPoint" datatype="URL" isreference="yes"/>
            <element ref = "hostingRedirector" datatype="URL" isrefer-
ence="yes"/>
            <element ref = "tModelInstanceDetails" datatype="tm:tModelKey"
isreference="yes"/>
         </group>
         <attribute name = "bindingKey" minOccurs = "1" type = "string"/>
         <attribute name = "serviceKey" type = "bs:serviceKey"/>
      </type>
   </element>
<constraint>
   <goal>
      <batom>
         <wdConstraint>
             <atom>
                 <predicate>
                     isValid
                  </predicate>
                  <args>
                     <term>
                     <variable>
                         $bindingTemplate/serviceKey
                     </variable>
                    </term>
                 </args>
              </atom>
         </wdConstraint>
```

```
   </batom>
   </goal>
</constraint>

</schema>
```

The key thing to note is that the addition of the constraint element to the schema allows us to capture the requirement that the references in the bindingTemplates, businessServices, tModels, and businessEntitys should be well formed. For the purposes of this exercise, we have assumed that the check for validity of these references is a well known operation as defined by the semantics of UDDI.

2. Owner of any UDDI entity:
Furthermore, in UDDI, any BusinessService is owned by a BusinessEntity, and any BindingTemplate is owned by a BusinessService that has to already exist. The offers also have a notion of offerOwner. In addition, if any part of a description has a reference to another entity that is to be registered, the implementation can stipulate that the references that are created have to exist. Therefore, the description of the business services has references to the business entity that owns it.

3. find_*** APIs:
The query language supported by e-speak is much richer than the query language supported by UDDI. The query language in UDDI is basically based on matching the names, identifiers, and catoegories and tModel Ids. Furthermore, the matching is a logical OR when applied to identifiers, a logical AND when applied to tModel id matches, a logical OR when applied to url like references, and a logical AND when applied to categories.

4. delete_*** APIs:
The matchmaker supports a retract-offer API that essentially captures the functionality provided by the delete_*** APIs in UDDI.

Appendix B: Software Support for SFS

Developing and deploying e-services

To be filled in...

Generic Software Stack for sending and receiving SFS messages

The following figure shows how the various protocol layers correspond to the abstract software stack of any servers and clients in the eco-system that send and receive SFS messages. We do not describe here what software components realize these abstract software layers, as this may vary from server type to server type. The appendix describes how e-speak products support SFS and SFS messaging.

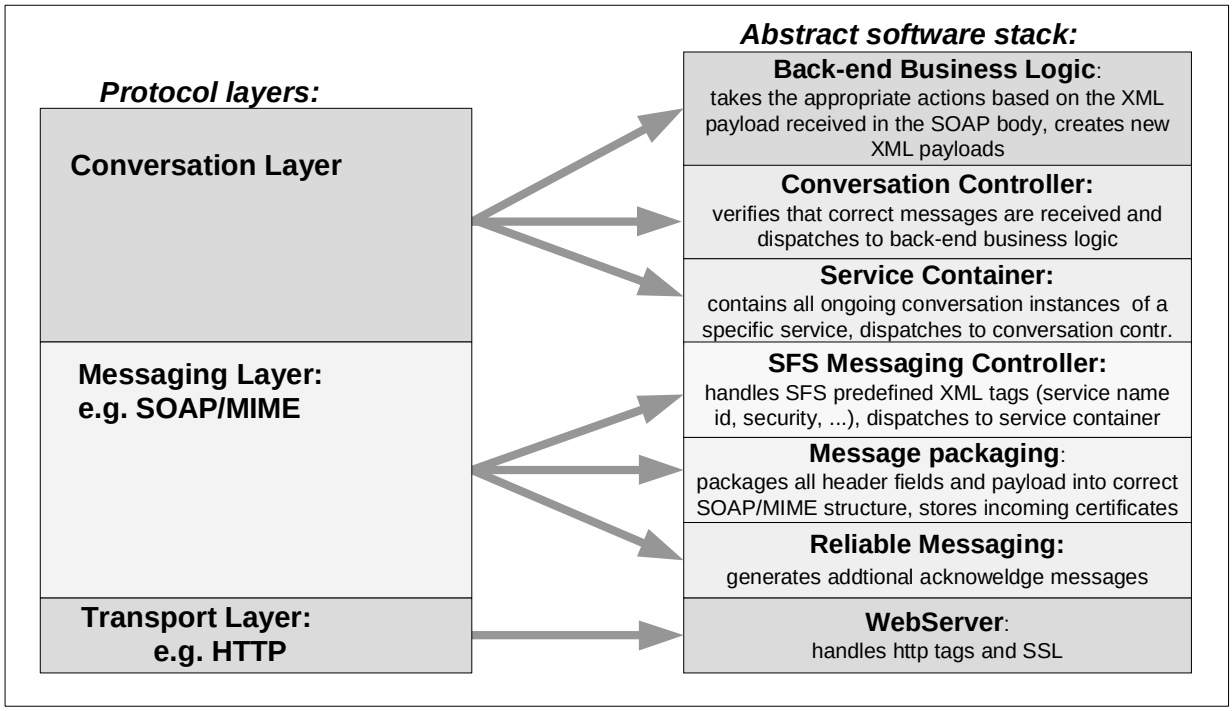


Figure 1: Figure: Abstract software stack of servers and clients

E-speak: E-Services Village (Collaborative Portal Framework)

....

E-speak: Conversation Server

...