



uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES
ET POSTDOCTORALES



FACULTY OF GRADUATE AND
POSTDOCTORAL STUDIES

Benjamin Ezze

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.Sc. (E-Business Technologies)

GRADE / DEGREE

E-Business Technologies

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

A Policy-Based Message Broker for Event-Driven Services in B2B Networks

TITRE DE LA THÈSE / TITLE OF THESIS

Dr. Liam Peyton

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Dr. Morad Benyoucef

Dr. Craig Kuziemy

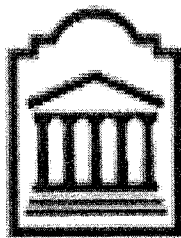
Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

A Policy-based Message Broker for Event-driven Services in B2B Networks

Benjamin Eze

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the M. Sc. degree in E-business Technologies



University of Ottawa
Ottawa, Ontario, Canada
January 2009

© Benjamin Eze, Ottawa, Canada, 2009



Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-51835-9
Our file Notre référence
ISBN: 978-0-494-51835-9

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.


Canada

Abstract

Today's global economy requires businesses to expand their processes beyond organizational domains to integrate business partners and collaborators. SOA through Web services is emerging as a framework for bridging heterogeneous business systems over the Internet. Unfortunately it is still limited by many of the constraints of the legacy applications: procedural interaction, data polling and strong coupling of applications. Event-driven systems like publish/subscribe on the other hand provide a flexible interaction pattern that does not require rigid transport or communication protocol. Robust B2B process integration requires a framework that supports enterprise and internet scale data sharing, point-to-point, push interaction as well as publish/subscribe broadcast based interaction. In our thesis, we accomplish this through a framework that views a B2B network as a streaming database and uses declarative policies to describe data sharing based on an SOA publish/subscribe infrastructure. The key contribution of our thesis is to define a flexible policy approach for describing streaming data. A Palliative Severe Pain Management scenario was implemented to evaluate our framework against BPI frameworks.

Acknowledgment

I particularly want to thank my supervisor, Dr. Liam Peyton for his guidance and encouragement throughout the course of my research and during the writing of this thesis. I am profoundly grateful for those insights that kept moving this work in the right direction. I have richly gained from our interactions and meetings.

I also appreciate Grant Middleton for his involvement in various areas of this research. I would like to thank Allison Parsonage and Brandi Donnelly for proof reading various sections of this thesis. I also want to thank the entire Palliative care research team for their helpful contributions.

This work was supported, in part, by research grants from the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Ontario Research Fund (ORF) created by the Ontario Ministry for Research and Innovation (MRI).

Finally, I thank my family for their unfailing emotional and financial support since the inception of this program.

Table of Contents

ABSTRACT	2
ACKNOWLEDGMENT	3
TABLE OF CONTENTS	4
LIST OF FIGURES	8
LIST OF TABLES	10
LIST OF ACRONYMS	11
CHAPTER 1. INTRODUCTION	14
1.1 THESIS MOTIVATION AND OBJECTIVES.....	14
1.2 CONTRIBUTIONS	15
1.3 THESIS METHODOLOGY	15
CHAPTER 2. BACKGROUND	17
2.1 BUSINESS PROCESS INTEGRATION.....	17
2.2 EVENT-DRIVEN ARCHITECTURES	18
2.3 PUBLISH/SUBSCRIBE SYSTEMS	19
2.3.1 Message Queue Telemetry Transport (MQTT)	21
2.3.2 Java Message Service	21
2.3.3 Really Simple Syndication.....	22
2.4 SERVICE ORIENTED ARCHITECTURE (SOA)	22
2.5 WEB SERVICE STANDARDS	24
2.5.1 SOAP	24
2.5.2 WSDL	27
2.5.3 UDDI	27
2.5.4 Web Service Addressing	29
2.5.5 Web Service Notification and Topics	29
2.5.6 Web Service Eventing	31
2.5.7 WS-BPEL	32
2.6 POLICY FRAMEWORKS.....	35
2.6.1 Web Services Policy Framework (WS-Policy)	35
2.6.2 Event Condition Action (ECA).....	36
2.6.3 XACML	37

2.7	RELATED WORKS	40
CHAPTER 3.	POLICY-BASED SOA PUB/SUB FRAMEWORK.....	42
3.1	REQUIREMENTS FOR DATA-DRIVEN BUSINESS PROCESS INTEGRATION	42
3.1.1	Information Discovery and Categorization.....	43
3.1.2	Platform Independence	43
3.1.3	Configurable Communication Patterns	44
3.1.4	Service Quality.....	45
3.1.5	Surveillance Infrastructure	45
3.1.6	Dynamic Business Configuration	45
3.1.7	Security.....	46
3.2	SOA PUBLISH/SUBSCRIBE FRAMEWORK.....	46
3.2.1	Simple SOA Message Broker.....	48
3.2.2	Data Sharing Model	50
3.2.3	The Partner Service Component.....	50
3.2.4	Partner Registration	52
3.2.5	Partner Service Communication Authentication	53
3.2.6	Surveillance Infrastructure	54
3.2.7	Topic Registrar.....	54
3.3	POLICY-BASED SOA PUBLISH/SUBSCRIBE	55
3.3.1	Policy-based SOA Message Broker	56
3.3.2	Policy Model	60
3.3.3	Policy Language Model	62
3.3.4	Types of Policies	64
3.3.5	Subscription Management	65
3.3.6	Notification Message	67
3.3.7	Managing Pullpoints	70
3.3.8	Batched Message Retrieval.....	71
3.3.9	Data Queries.....	72
3.4	SUMMARY	73
CHAPTER 4.	CASE STUDY.....	74
4.1	PALLIATIVE CARE.....	74
4.2	PALLIATIVE SEVERE PAIN MANAGEMENT SCENARIO	75
4.3	WEB APPLICATION PORTAL.....	76
4.3.1	Web Application Portal Architecture.....	76
4.3.2	Analysis.....	77
4.4	INTEGRATED BPEL CoT	78
4.4.1	Integrated BPEL CoT Architecture	78
4.4.2	PainScore BPEL	80
4.4.3	PainReport BPEL	81
4.4.4	PainAlert BPEL	82
4.4.5	PrescriptionChange BPEL Process.....	83
4.4.6	Analysis.....	85

4.5	SIMPLE SOA PUBLISH/SUBSCRIBE	85
4.5.1	Simple SOA Publish/Subscribe Architecture	85
4.5.2	Registration	87
4.5.3	Defining Data	88
4.5.4	Subscription	88
4.5.5	Receiving and routing Messages	89
4.5.6	Analysis	90
4.6	POLICY-BASED SOA PUBLISH/SUBSCRIBE	91
4.6.1	Policy-based SOA Publish/Subscribe Architecture	91
4.6.2	Policy-based Subscriptions	93
4.6.3	Retrieving Batched Messages	96
4.6.4	Requesting Stateful data through a DataQuery	96
4.6.5	Support for Multiple Groups and Roles	97
4.6.6	Analysis	99
CHAPTER 5.	EVALUATION	100
5.1	CANDIDATE BPI FRAMEWORKS	100
5.1.1	Web Application Portal	100
5.1.2	Integrated BPEL Liberty Alliance CoT	101
5.1.3	Simple SOA Publish/Subscribe	101
5.1.4	Policy-based SOA Publish/Subscribe	101
5.2	INFORMATION TOPIC	102
5.2.1	Information Topic Definition	103
5.2.2	Information Discovery	104
5.2.3	Information Selection	104
5.2.4	Information Aggregation	104
5.2.5	Information Transformation	105
5.3	COMMUNICATION PATTERN	105
5.3.1	Coupling	106
5.3.2	Interaction Type	107
5.3.3	Interaction Pattern	107
5.4	MESSAGE DELIVERY SERVICE QUALITY	108
5.4.1	Delivery Mode	108
5.4.2	Delivery Service Quality	109
5.4.3	Batched/Store and Forward	109
5.5	INFORMATION MANAGEMENT	109
5.5.1	Common Data Model	110
5.5.2	Active Information Subscription Management	110
5.5.3	Surveillance Infrastructure Support	111
5.6	EVALUATION OF ENGINEERING EFFORT	111
5.7	RELATIONSHIP WITH RELATED WEB SERVICE SPECIFICATIONS	113
5.7.1	Topic/Content Filtering	115
5.7.2	Message Transformation	115
5.7.3	Message Delivery and Service Quality	115
5.7.4	Use of an Intermediary	116
5.7.5	Identification of Subscribers and Publishers	116

5.7.6	Security	117
5.7.7	Dependency on other specifications.....	117
5.7.8	Policy-based approach	117
5.7.9	Stateful data and Surveillance infrastructure.....	117
5.7.10	Support for Legacy systems.....	118
5.7.11	Summary	118
5.8	RELATIONSHIP OF POLICY MODEL TO XACML	118
CHAPTER 6. CONCLUSIONS		121
6.1	SUMMARY OF CONTRIBUTIONS	121
6.2	FUTURE WORK	122
6.2.1	Distributed Policy-based SOA Publish/Subscribe	122
6.2.2	Identity Management Infrastructure	123
6.2.3	Automatic Definition of Subscription Policies.....	123
6.2.4	Partner Service Component Support for BPEL	123
REFERENCES		124
APPENDIX A – POLICY SOA PUB/SUB DATA MODEL		130
SCHEMA		130
POLICY-BASED MESSAGE BROKER’S WSDL		134
PARTNER SERVICE COMPONENT WSDL		135
APPENDIX B –POLICY DATABASE.....		137
POLICY DATABASE RELATIONSHIP DIAGRAM.....		137
SQL SCRIPTS		138

List of Figures

FIGURE 1.	PUBLISH-SUBSCRIBE ILLUSTRATED	19
FIGURE 2.	SOA ILLUSTRATED	23
FIGURE 3.	SOAP MESSAGE	25
FIGURE 4.	RPC TYPE INTERACTION	25
FIGURE 5.	DOCUMENT STYLE WEB SERVICE INTERACTION	26
FIGURE 6.	UDDI REGISTRY DATA MODEL	28
FIGURE 7.	WS-BASENOTIFICATION AND WS-BROKEREDNOTIFICATION ROLES AND RELATIONSHIPS	31
FIGURE 8.	WS-EVENTING COMPONENTS AND INTERACTIONS	32
FIGURE 9.	WS-BEL AND THE WEB SERVICE STACK	33
FIGURE 10.	XACML DATA-FLOW DIAGRAM.....	38
FIGURE 11.	XACML POLICY LANGUAGE MODEL	39
FIGURE 12.	SIMPLE SOA PUBLISH/SUBSCRIBE MODEL ILLUSTRATED	47
FIGURE 13.	SIMPLE WEB SERVICE MESSAGE BROKER	48
FIGURE 14.	SOA PUBLISH/SUBSCRIBE DATA SHARING MODEL.....	50
FIGURE 15.	THE PARTNER SERVICE COMPONENT.....	51
FIGURE 16.	PARTNER SERVICE REGISTRATION WITH THE POLICY MESSAGE BROKER.....	52
FIGURE 17.	SECURITY TOKEN USE ILLUSTRATED.....	53
FIGURE 18.	TOPIC REGISTRATION PROCESS.....	55
FIGURE 19.	POLICY-BASED SOA PUBLISH/SUBSCRIBE FRAMEWORK	57
FIGURE 20.	THE POLICY-BASED MESSAGE BROKER MODEL.....	58
FIGURE 21.	DATA-CENTRIC NOTIFICATION PATTERN	59
FIGURE 22.	XACML ROLE-BASED VIEW	60
FIGURE 23.	POLICY FRAMEWORK.....	61
FIGURE 24.	POLICY LANGUAGE REPRESENTATION	62
FIGURE 25.	POLICY TYPES.....	64
FIGURE 26.	SUBSCRIBE OPERATION	66
FIGURE 27.	RENEW/PAUSE/UNSUBSCRIBE OPERATION	67
FIGURE 28.	WRAPPED MESSAGE STRUCTURE.....	67

FIGURE 29.	NOTIFICATION MESSAGE PROCESSING	68
FIGURE 30.	CREATE PULLPOINT OPERATION	71
FIGURE 31.	BATCHED MESSAGE RETRIEVAL ILLUSTRATED	71
FIGURE 32.	THE USE OF DATA QUERIES.....	72
FIGURE 33.	PALLIATIVE SPM SCENARIO.....	76
FIGURE 34.	WEB APPLICATION PORTAL ARCHITECTURE	77
FIGURE 35.	INTEGRATED BPEL COT ARCHITECTURE	79
FIGURE 36.	<i>PAINSCORE</i> BPEL.....	81
FIGURE 37.	<i>PAINREPORT</i> BPEL.....	82
FIGURE 38.	<i>PAINALERT</i> BPEL	83
FIGURE 39.	<i>PRESCRIPTIONCHANGE</i> BPEL.....	84
FIGURE 40.	SOA PUBLISH/SUBSCRIBE ARCHITECTURE	86
FIGURE 41.	SOA MESSAGE EXCHANGE FOR PARTNER REGISTRATION	88
FIGURE 42.	SUBSCRIPTION TO <i>PAINREPORT</i>	89
FIGURE 43.	MESSAGE STRUCTURE: <i>PAINREPORT</i>	90
FIGURE 44.	POLICY-BASED SOA PUBLISH/SUBSCRIBE ARCHITECTURE.....	92
FIGURE 45.	POLICY-BASED SUBSCRIPTION.....	95
FIGURE 46.	DATA QUERY SOAP MESSAGE.....	97
FIGURE 47.	SUPPORTING MULTIPLE GROUPS AND ROLES IN A PARTNER SERVICE COMPONENT	98

List of Tables

TABLE 1.	SUMMARY OF THE MESSAGE BROKER INTERFACES	49
TABLE 2.	NOTIFICATION MESSAGE TYPES.....	59
TABLE 3.	FILTER COMBINING ALGORITHMS	63
TABLE 4.	PALLIATIVE SPM MANAGEMENT IN A PUBLISH/SUBSCRIBE TOPOLOGY.....	87
TABLE 5.	CORRELATION LOOKUP TABLES	98
TABLE 6.	COMPARISON OF BPI FRAMEWORKS ON INFORMATION TOPIC	102
TABLE 7.	COMPARISON BY COMMUNICATION PATTERNS	106
TABLE 8.	COMPARISON BY MESSAGE DELIVERY TECHNIQUES AND SERVICE QUALITIES	108
TABLE 9.	COMPARISON BY INFORMATION MANAGEMENT APPROACHES.....	110
TABLE 10.	EVALUATION OF EFFORTS AND COMPLEXITY TO CREATING DATA-DRIVEN PROCESSES.....	112
TABLE 11.	RELATIONSHIP WITH WS-NOTIFICATION AND WS-EVENTING	114
TABLE 12.	SIMILARITIES OF FUNCTIONALITY BETWEEN XACML RBAC AND THE POLICY-BASED SOA PUBLISH/SUBSCRIBE	119

List of Acronyms

Acronym	Definition
API	Application Programming Interface
B2B	Business-to-Business
BP4WS	Business Process Execution Language for Web Services
BPI	Business Process Integration
CORBA	Component Object Request Broker Architecture
CoT	Circle of Trust
CRM	Customer Relationship Management
DCOM	Distributed Component Object Model
DNS	Domain Name System
DPWS	Devices Profile for Web Service
ECA	Event Condition Action
EDA	Event Driven Architecture
EDI	Electronic Data Interchange
EPR	Endpoint Reference
ERP	Enterprise Resource Planning
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol

IDP	Identity Provider
IIOp	Internet Inter-Orb Protocol
ITPF	Integrated Trust Process Framework
J2EE	Java 2 Platform Enterprise Edition
JMS	Java Message Service
MQTT	MQ Telemetry Transport
OASIS	Organization for the Advancement of Structured Information Standards
OC4J	Oracle Containers for J2EE
ODE	Orchestration Director Engine
P2P	Peer-to-Peer
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
Service Quality	Service Quality
RBAC	Role-based Access Control
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RSS	Really Simple Syndication
SAML	Security Assertion Markup Language
SAP	Systems, Applications and Products
SCM	Supply Chain Management
SMTP	Simple Mail Transport Protocol
SOA	Service Oriented Architecture

SOAP	Simple Object Access Protocol
SQ	Service Quality
TTP	Trusted Process Provider
UDDI	Universal Description Discovery and Integration
URI	Universal Resource Identifier
W3C	World Wide Web Consortium
WS	Web Service
WS-BPEL	Web Service Business Process Execution Language
WSDL	Web Service Description Language
WSRF	Web Service Resource Framework
XACML	eXtensible Access Control Markup Language
XML	Extensible Markup Language
XPath	XML Path Language
XSD	XML Schema Document

Chapter 1. Introduction

1.1 Thesis Motivation and Objectives

The dynamic nature of Business to Business (B2B) networks is causing enterprises to continuously seek better ways of integrating applications and business processes (Themistocleous & Irani, 2002). The absence of integration is a substantial restricting factor to the automation of business processes, resulting in the proliferation of islands of data that do not share a common integration framework (Liu & Plale, 2003).

Service Oriented Architecture provides an efficient framework for combining disparate applications through the Internet. However, the process-centric nature of SOA limits its application for dynamic data sharing. The objective of this thesis framework is to combine SOA and event-driven publish/subscribe in one data sharing architecture through declarative policies. A declarative policy framework using an SOA message broker provides the needed flexibility for heterogeneous data sharing required in dynamic, event driven B2B process integration.

The key goal of our framework is to use declarative policies to:

- combine topic, content-based and type-based subscriptions in one middleware;
- achieve flexible communications patterns like one-to-one, one-to-many, and many-to-one message exchanges through the same middleware;
- account for subscriptions, and reliable message delivery through a Service Quality aware publish/subscribe;
- achieve message filtering and automatic message format translation.

1.2 Contributions

This thesis proposes and evaluates a new approach to achieving flexible data sharing and BPI in B2B networks. In doing so, the following key contributions were made:

- Identified the essential requirements/criteria for flexible data sharing and BPI in a B2B that any framework needs to support.
- Proposed a policy-based SOA publish/subscribe message broker architecture for event-driven BPI in a B2B network;
- Proposed a common, flexible, streaming, relational, B2B data model for event-driven BPI based on streaming events organized into topics (and their attributes).
- Proposed a Policy Model based on Extensible Accessible Control Markup Language (XACML) syntax for defining broadcast and targeted messages, as well as flexible delivery formats, Service Quality, and both system-wide and topic-specific constraints.
- Proposed a Partner Service component that acts as both a publisher and subscriber to messages for flexible integration with devices, applications and business processes.

1.3 Thesis Methodology

The methodology followed an evolutionary process of analysis, construction and evaluation. In particular we took the following steps:

1. Identify and analyze problem (Data-driven Business Process Integration)
2. Review literature and industry approaches to problem
3. Identify strengths and shortcomings of these approaches
4. Create a proof of concept SOA publish/subscribe message broker
5. Case study (Palliative Care Application) to evaluate and compare this messaging model to traditional portal-based approaches

6. Identify the shortcomings of previous work and review objectives for thesis
7. Extend the SOA publish/subscribe architecture with a policy-based approach to address these shortcoming and meet the new set of objectives.
8. Case study to evaluate and compare the policy-based approach to portal and BPEL-based process integration
9. Draw conclusions and identity future work.

Chapter 2. Background

2.1 Business Process Integration

The flow of information between parties in B2B networks is crucial for effective and efficient collaboration. As a result, businesses are continuously seeking new, fast, inexpensive and secure means of sharing information on goods and services (Premkumar, Ramamurthy, & Sree, 1994). This need is not new, as electronic data interchange (EDI) has been around for over three decades now (Stefansson, 2002). The rapid growth of the Internet has provided a universal platform for businesses to migrate their applications and services for better integration with partners and customers (Bailey, Poulouvasilis, & Wood, 2002).

The Internet and its associated technologies have brought to light new opportunities and challenges to business process integration. Businesses are relying more on the Internet as a platform for broadening their organizational domains beyond the enterprise boundaries. As enterprises rely more and more on the web for spreading their tentacles around the globe, the definition of organizational domain has started changing. Global competition, changing global economics and rapid evolution in information technologies are increasingly pushing companies to outsource various logistical arms of their operations to external parties (Ira. Lewis & Alexander, 2000). In addition, the need to streamline operations in today's global economy is forcing businesses to depend more on other companies to be globally competitive. We are seeing an ever increasing need for electronic data sharing.

These pressures are fostering new requirements for distributed applications which include continued decentralization, and distribution of software, hardware as well as human resources. Applications, databases, processes and increasingly devices share resources, and need to exchange data on a continuous basis. (Foster, Kesselman, Nick, & Tuecke, 2002). Important considerations include communication, methods of interaction, security and privacy semantics, distributed resource management, performance, and Service Quality.

Efficient B2B data sharing requires a more flexible, manageable and maintainable enterprise infrastructure that can support changing business and technical requirements. B2B data sharing requires support for both procedural (API) and data-centric communication, enterprise and Internet-scale data sharing as well as point-to-point (pull type) interaction and publish/subscribe broadcast (push) interaction.

The goal of BPI is to reduce integration time and maintenance cost. Enterprises need not follow a monolithic approach but combine process-centric and data-centric integration technologies. In addition, there is the need to utilize both traditional integration technologies like database-oriented middleware, distributed objects, interface-based technologies as well as recent application integration technologies like Web services, service oriented and event-driven architectures (Themistocleous & Irani, 2001).

2.2 Event-driven Architectures

Event-driven architecture (EDA) is characterized by its ability to decouple service providers and consumers through messages (Niblett & Graham, 2005). That is, producers and consumers of data are contextually coupled by the data exchanges and not procedural calls. Applications produce events that get consumed by other applications. Central to this architecture is the event notification service, an application independent infrastructure whereby generators of events publish notifications to this infrastructure and event consumers subscribe to these events (Carzaniga, Rosenblum, & Wolf, 2001) and have messages pushed to them as they become available.

Point-to-point BPI integration used by most traditional enterprise systems including Service Oriented Architecture (SOA) requires the client to continuously poll the server for data using a strongly coupled messaging model. In EDA, situational changes inside or outside a business domain trigger messages that get disseminated to all interested parties for actions (Michelson, 2006) that may trigger other events. EDA provides support for various types of message exchanges: one-to-one, one-to-many, many-to-one, and many-to-many interactions. In addition, it fosters flexible loose coupling (Niblett & Graham, 2005), asynchronous communication, and traceability.

2.3 Publish/Subscribe Systems

Publish/subscribe is an event-driven model that allows many clients to subscribe to the same service, and have the information sent to them as messages as they become available. The basic principles of publish/subscribe were identified by (Eugster, Felber, Guerraoui, & Kermarrec, 2003). It is characterized by one or more publishers or event producers, subscribers to events or event consumers and an event service, the message broker (Figure 1). The event service decouples the producers and consumers of information in time (messages need not be sent and received at the same time), space (service provider and consumers need not be connected to each other) and synchronization (service providers and consumers need not synchronize interactions).

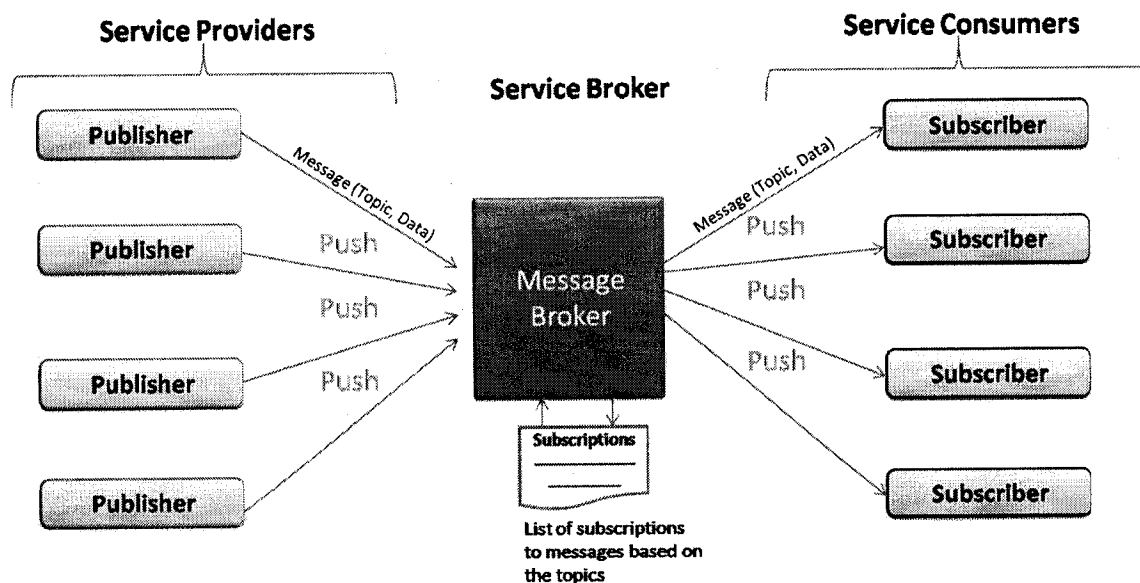


Figure 1. Publish-subscribe illustrated

The message broker acts as a central application hub for routing messages to subscribers. It supports asynchronous "push" communication and could perform message translation and formatting for heterogeneous subscribers (Themistocleous & Irani, 2002). Publish/subscribe systems are usually characterized by the types of subscriptions they handle. As identified by (Eugster, Felber, Guerraoui, & Kermarrec, 2003), the three most popular types are:

- **Topic-based subscriptions:** This is based on the notion of topic/subjects as an indication of interest to events. Topics are then applied to a distinct communication channel. It usually presents a static and primitive type of implementation but is usually very efficient.
- **Content-based subscriptions:** This improves on topic-based subscription by basing the subscription on the actual content of an event, event data structures, event meta-data, message filters and content correlation. It is highly expressive, but requires sophisticated protocols and high runtime overhead.
- **Type-based subscriptions:** This type of subscription extends the event system through classification attributes that are often natural descriptors of the event or message. For example request and response.

Integrating business processes using publish/subscribe increases scalability by removing explicit dependencies between the interacting participants (Eugster, Felber, Guerraoui, & Kermarrec, 2003) and is therefore very adaptable to distributed environments like the Internet. This paradigm presents many advantages: there is total de-coupling of all collaborating parties; interactions are based purely on messages organized into topics and subscriptions to the messages of a given topic. This push nature of message delivery can be a major performance boost because it relieves subscribers of the burden of continuously polling the servers for messages. Communication is usually anonymous, asynchronous, multicast in nature and is flexible enough for easy modifications in a dynamic environment.

The publish/subscribe paradigm is therefore very suitable for not only application-to-application communications, but collaboration with resource constrained applications running in low power devices especially in mobile and adhoc environments (Yongqiang Huang & Garcia-Molina, 2004). Message brokers interconnect heterogeneous applications like CRM, SAP, SCM, e-Procurement, including legacy systems and databases. According to (Pereira, Fabret, Jacobsen, Llibat, & Shasha, 2001), publish/subscribe when used in large-scale information dissemination systems bridges interoperability gaps between communications platforms (wireless, wired, mobile and small devices). Other popular implementations of distributed content-based publish/subscribe systems are Elvin, Siena and Gryphon (Gupta, Sahin, Agrawal, & El-Abbadi, 2004).

The efficiency of the publish/subscribe topology can also be tied to the topology of the underlying network. Topologies can span from simple to distributed and peer-to-peer systems. (Terpstra, Behnel, Fiege, Zeidler, & Buchmann, 2003) identified the benefits of integrating a content-based publish/subscribe system to a dynamic peer-to-peer (P2P) overlay network. The goal of this combination is to eliminate a single point-of-failure associated with using a single node as the message broker or topology bus.

There are also some disadvantages to this paradigm. Initial design and implementations are often more complex when compared with other approaches. In addition, publish/subscribe architecture may not guarantee the delivery of messages to all intended recipients since messages are usually pushed to subscribers without confirmation (Papazoglou & Ribbers, 2006). Therefore the basic publish/subscribe model may not be well suited for applications that require strong guarantees for message delivery. It may also be predisposed to security issues when malicious publishers and unauthorized subscribers are involved (Mudhakar & Ling, 2005). It does not provide for demand-based messaging and stateful data sharing.

2.3.1 Message Queue Telemetry Transport (MQTT)

MQTT is a TCP/IP based protocol that uses Transport Control Protocol (TCP) ports as channels to support publish/subscribe communication pattern. It is also one of the transport protocols supported by the WebSphere Business Integration Broker (O'Connel & Stanford-Clark, 2005). MQTT requires all collaborating stations and devices to support the MQTT protocol in order to participate in publish/subscribe interaction pattern.

As a publish/subscribe pattern, MQTT is more adaptable for small messaging applications like pipeline monitoring, energy and utilities billing, process control, weather monitoring, environmental monitoring and control, security and factory automation (Stanford-Clark, 2004). Its dependence on TCP limits its application scope to low-level applications and APIs and makes it unsuitable for Internet-scale data sharing.

2.3.2 Java Message Service

The Java Messaging Service (JMS) API is a messaging standard built on top of the Java 2 Platform, and Enterprise Edition (J2EE). JMS enables application components to create, send,

receive and manipulate messages in a loosely coupled, reliable and asynchronous manner (SDN, 2008). The JMS supports both point-to-point and publish/subscribe messaging. When used in publish/subscribe applications, clients address messages to a topic. The publishers and subscribers can dynamically publish or subscribe to those topics.

The JMS provider acts as the message broker with the publishers sending messages to the JMS provider. The subscribers receive messages from the JMS provider based on their subscription to one or more topics. JMS scales well with applications that run in the same network/domain but may not be well adapted for application components that send and receive messages across domains and firewalls (Thomas, 2002).

2.3.3 Really Simple Syndication

Really Simple Syndication (RSS) is based on the publish/subscribe pattern. Clients subscribe to feeds of interest from online sources using an aggregator. Periodically, the aggregator polls the feeds to receive updates for the client. RSS contents are encoded in XML and are displayed by a feed reader or an RSS-integrated web browser.

RSS uses a message format that can be read by any aggregator. Though very widely adopted, (Ma, Xu, Wang, & Huang, 2006) indicated that “RSS feeds face high loads imposed by active polling and clients that don’t detect updates in time”. The characteristics and behaviours of RSS as a publish/subscribe system were analysed by (Hongzhou, Venugopalan, & Emin, 2005). In spite of being the first widely used publish/subscribe pattern on the Internet, the lack of an inherent call-back mechanism in RSS still makes it a controversial publish/subscribe system. RSS is good for information aggregation and distribution from a single message broker to polling subscribers. However, it lacks a definite distinction between the information broker and the publisher. In most implementations, publishers are totally nonexistent or represented by tightly integrated online information sources.

2.4 Service Oriented Architecture (SOA)

SOA is a software architecture for grouping functionalities around business processes that gets packaged as interoperable services. According to (Le Hegaret, 2003; W3C), SOA is “a set of components which can be invoked, and whose interface descriptions can be published and

discovered". SOA approaches application integration through a layered approach. It is characterized by business functions that are modeled as networked services, and a collection of self-describing application interfaces (Eidson, Maron, Pavlik, & Raheja, 2005). It organizes and describes Web services to support dynamic and automated discovery, and binding in heterogeneous computing environments (Papazoglou & Ribbers, 2006) (Figure 2).

SOA is a request/response type interaction pattern and its popularity stems from the fact that message exchanges can span beyond organizational domains and corporate firewalls. SOA through Web services supports platform independence, interoperability and communication between heterogeneous applications.

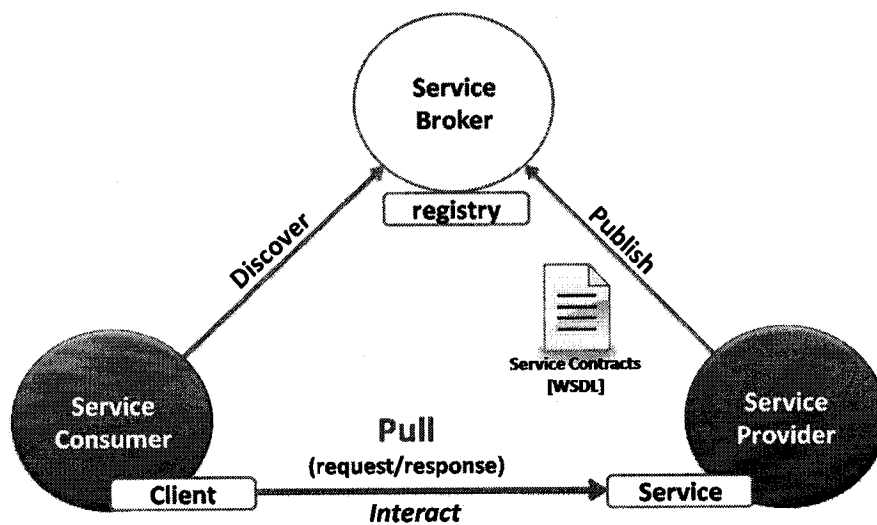


Figure 2. SOA Illustrated

SOA implementations are usually rigid and static because of this underlying point-to-point and synchronous communication pattern. Its use in large scale application implementations results in complex software engineering (Natis, 2003). In addition, SOA provides no inherent support for adhoc and event-driven applications often required in B2B data sharing. These shortcomings spurred research efforts both in the academia and among major software vendors like IBM and Microsoft into collaborating with standards bodies, especially OASIS and W3C, to define various technical specifications targeted at extending Web services and SOA to support event-driven services and data-driven business processes. Oracle and some vendors describe such

event-driven SOA applications as SOA 2.0 compliant (Hahn, 2006). This term is still very controversial but it says a lot about the direction SOA is heading for the future.

2.5 Web Service Standards

This decade has seen Web Services emerge as the standard for systematic and extensible framework for machine-to-machine interaction on the Internet. Web services also provide an interface for existing back-end applications such as Customer Relationship Management (CRM) and Enterprise Resource Planning (ERP) systems (WebLogic, 2008). This ability to encapsulate application logic and package communications using standard protocols like XML and HTTP, exposing them to the Internet is the key to its wide adoption. Built on existing Internet transport protocols like HTTP and FTP, Web services do not suffer the drawbacks of the earlier communication approaches like CORBA, RMI, and DCOM (Curbera et al., 2002). They are based on open standards that shield developers from vendor specific details required in consuming services.

The Web Services Architecture is based on three W3C specifications: Simple Object Access Protocol (SOAP), the communication protocol, Web Service Description Language (WSDL) for service description and Universal Description, Discovery and Integration (UDDI) for Web service discovery.

2.5.1 SOAP

SOAP is a lightweight XML-based protocol used to exchange information in decentralized, distributed environments using Web services. It is an XML-based protocol for XML messaging and for making remote procedural calls between web service end-points (Curbera et al., 2002). Its goal is to diffuse the barriers of heterogeneity that separate distributed computing platforms.

In a web service, SOAP handlers serialize and deserialize XML messages and Remote Procedural Calls (RPC). SOAP messages are usually the data payload of popular Internet transport protocols like HTTP, FTP and SMTP. As shown in the Figure 3, the SOAP Envelope is made of an optional SOAP header, the SOAP body containing the XML payload and an optional SOAP Fault element.

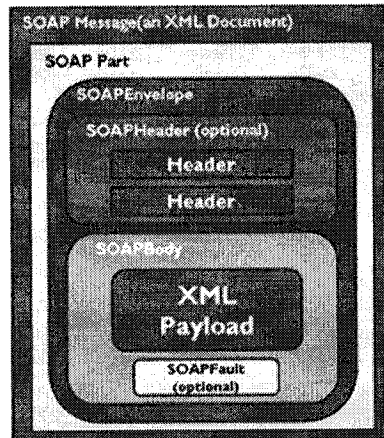


Figure 3. SOAP Message Illustrated

SOAP was initially designed to support only RPC style messaging on the web (Tyagi, 2004). RPC encoding style imitates procedural calls in traditional programming and usually results in monolithic and rigid implementations. The consequence of little modifications to the messaging structure is a redeployment of an entire system.

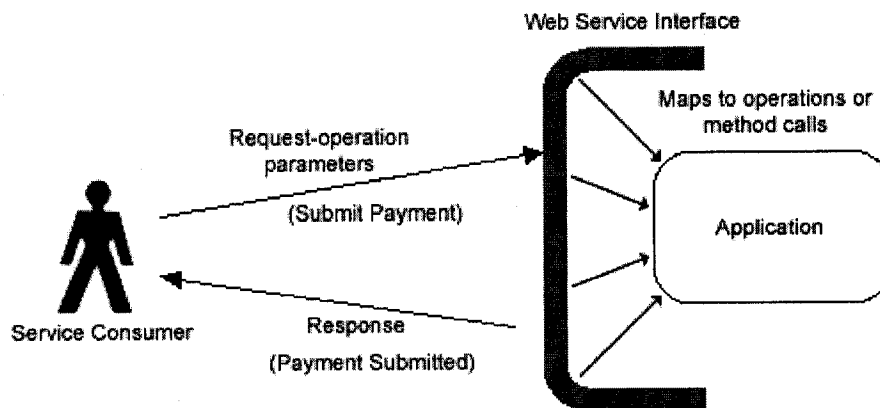


Figure 4. RPC type interaction¹

Document-style messaging was added from SOAP 1.0 to support the transmission of unstructured documents in SOAP messages. This flexibility allows the service providers and consumers to share different types of documents using similar interfaces and constructs. The

¹ Source: (Tyagi, 2004)

validation of the messages or documents is then pushed to the applications that will make use of the data.

One of the major differences between RPC and document-style messaging is the encoding style. Most document style implementations use literal encoding while RPC uses SOAP encoding. SOAP encoding uses a set of rules based on the XML schema data types to encode the data. The best choice of encoding styles is very dependent on implementations. Implementing document style encoding requires the developer to use low-level APIs to marshal the messages.

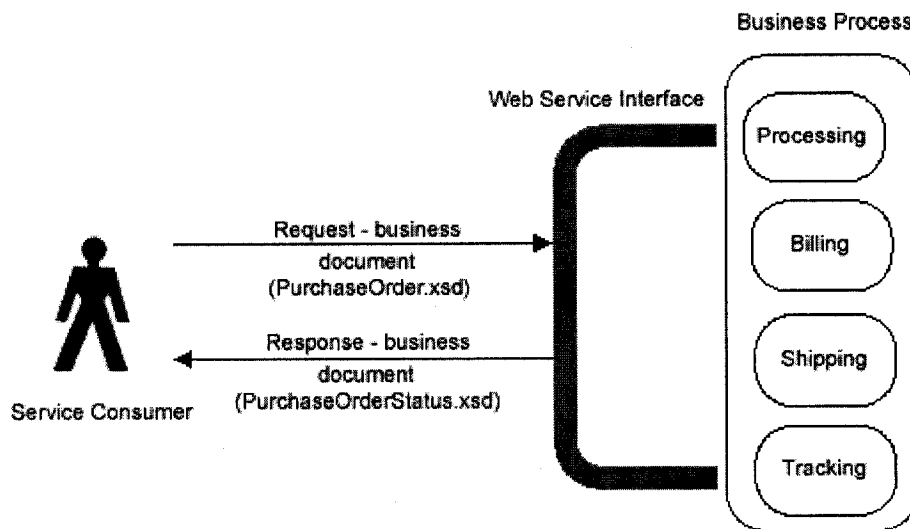


Figure 5. Document style Web service interaction²

Document style-messaging has some advantages over RPC. It supports asynchronous processing, imposes less rigidity in message validation between the data producer and consumer. In RPC, interface changes break the communication between the communicating services. Document-style uses the same interface, but requires changes to be made only in the schema which can be updated easily without breaking the communication. Other benefits of document style messaging include ease of interoperability, better performance and payload independence (McCarthy, 2002).

² Source: (Tyagi, 2004)

2.5.2 WSDL

Web Service Description Language (WSDL) is an XML format for describing Web services as collections of communication end points that can exchange certain messages (Brambilla, Ceri, Passamani, & Riccio, 2004). SOAP is used as a transport protocol but it lacks a mechanism for describing the interfaces and messages exchanges are needed to successfully interact with a service endpoint. WSDL provides a formalized description of client-service interaction. This W3C XML-based specification describes available Web service operations, input and output parameters, and how a client application connects and consumes to the service.

A WSDL document describes a Web services' interface and provides users with a point of contact to the service. It is divided into two sections. The first section, the *Abstract Interface Description* describes the vocabulary, messages and end point interaction of the web service. The second section, the *Concrete Service Endpoints Description* describes the binding protocol and exact end point reference of the service. The abstract elements includes: *Type definitions* that describe message exchanges, *Operations* are categorized in collections of *PortTypes* that represent an abstract communication interface to the endpoint. The actual description of the concrete protocol and messaging formats for these abstract elements are set in the Binding elements. A *port* defines an endpoint as a combination of the binding element and a network address. A collection of these ports define a Web service.

WSDL describes the publicly-exposed interface and the XML in the SOAP messages exchanged between applications. It could be used as the input to a proxy generator that produces client code according to service requirements. WSDL can also be used as input to a dynamic invocation proxy to generate the correct service requests at runtime.

2.5.3 UDDI

UDDI, Universal Description, Discovery and Integration is an OASIS specification (OASIS, 2001) that offers a unified and systematic way of finding service providers through a centralized registry. Synonymous to an automated online "phone directory" for Web services, the UDDI vision is to create interacting registries similar to what Domain Name Service (DNS) registries are for computers on the Internet (Mckendrick, 2000). UDDI depends on other W3C and IETF

Internet standards like XML, HTTP, and DNS protocols. It uses WSDL to describe interfaces to Web services and SOAP for messaging and transport (OASIS, 2001).

UDDI provides a platform for business partners and collaborators to publish Web service endpoint interface descriptors contained in their WSDLs, for other services to query, discover and bind to for information sharing (Liang-Jie, Tian, Henry, & Jen-Yao, 2004). The UDDI registry is divided into three major information structures: *White Pages* contain the name and contact details of the service provider, *Yellow Pages* provide service categorization based on business and service types while *Green Pages* provide technical data about the service (Curbera et al., 2002).

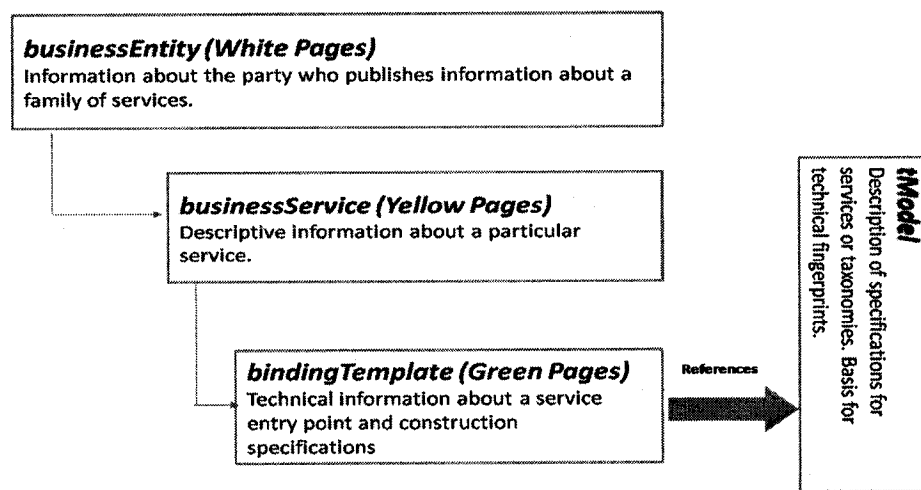


Figure 6. UDDI Registry Data Model

Figure 6, describes the relationship between the UDDI data structures as well as the information they capture on services and the publishing organizations. The `<businessEntity>` element is recorded in the *White Pages*. It provides the details on the organization publishing information about a family of services identified by one or more `<businessService>` elements. The `<businessService>` is a yellow page element providing descriptive details on each of these services. Finally the `<bindingTemplate>` elements provide the technical details on service level binding and messaging syntax. Like the `<businessService>`, the `<bindingTemplate>` references one or more `<tModel>` elements. *tModel* provides technical descriptions that adequately describe a Web service endpoint. UDDI uses unique identifiers to reference these data entities. many

unique key identifiers for each data entities. These keys are universally unique, providing a reference to arbitrary information types.

2.5.4 Web Service Addressing

It is important that Web services interoperate effectively and independent of technology and transport. According to (W3C, 2005), Web Services Addressing (WS-Addressing) "provides transport-neutral mechanisms to address Web services and messages". WS-Addressing defines two constructs for addressing messages. The first construct, a mandatory *endpoint reference* (EPR) provides the needed information for addressing a Web service endpoint. EPR is a Universal Resource Identifier (URI), representing the network address of a Web service. It makes a Web service endpoint a referenceable and addressable resource. The second construct, a set of optional *message addressing properties*, provide end-to-end characteristics of a message as a unit and facilitate interactions. These properties are immutable and are not intended to be modified along a message path.

EPR optional properties includes a Qualified Name (*QName*) identifying the selected interaction interface, a *QName* identifying a description of a set of endpoints at which a particular Web service is deployed and optionally a number of policies describing the behaviour, requirements and capabilities of an endpoint.

EPR makes up for the limitations of WSDL 1.1 description model on supporting dynamic usage patterns required for dynamic generation and customization of service endpoints. Message addressing properties are usually referred to as message information headers because they are embedded in the SOAP headers. These addressing properties identify the message (*wsa:MessageID*), the sender (*wsa:From*), the reply address (*wsa:ReplyTo*), the address that to send faults (*wsa:FaultTo*) and a mandatory action that must be taken on receipt of the message (*wsa:Action*).

2.5.5 Web Service Notification and Topics

Web services provide a platform for business integration and secure, universal access to data and applications (Sabbah, 2004). WS-Notification is a family of related specifications that define a standard Web service approach to notification using topic-based publish/subscribe pattern.

According the (OASIS), WS-Notification is an effort to “define a set of specifications that standardise the way Web services interact using "Notifications" or "Events"”. The goal is to standardize roles, terminologies, concepts and message exchanges needed in a web service notification pattern.

The three normative specifications that make up this family of specification include the WS-BaseNotification, WS-BrokeredNotification and WS-Topics.

WS-BaseNotification describes the standard message exchanges that service providers need to implement in order to participate in a direct notifications pattern. Figure 7 depicts the relationships between the various roles used in the WS-BaseNotification. The *NotificationProducer* subscribe to messages with the *SubscriptionManager* upon receiving a *SubscribeRequest* from a *Subscriber*. The *SubscriptionManager* is an endpoint that manages/administers subscriptions. The *NotificationConsumer* manages subscriptions on behalf of *Subscribers*. It is a WS-Addressing endpoint and can receive messages from the *NotificationProducer* on behalf of the *Subscriber*. *Subscriptions* define the relationship between the *NotificationConsumer*, the *NotificationProducer*, and filtering parameters like *Topics* and the *SubscriptionManager*.

Situations define those internal changes in a domain that may result in same or different types of *NotificationMessages*. *Publishers* publish *messages* containing information about a situation that needs to be communicated with other entities. The *NotificationProducer* then pushes the message to the *NotificationConsumer* with registered subscriptions to the message.

In WS-BrokeredNotification, the roles of *NotificationConsumer*, *NotificationProducer* and *SubscriptionManager* are combined in *NotificationBroker*. The *NotificationBroker* then serves as an intermediary between the publishers and message subscribers.

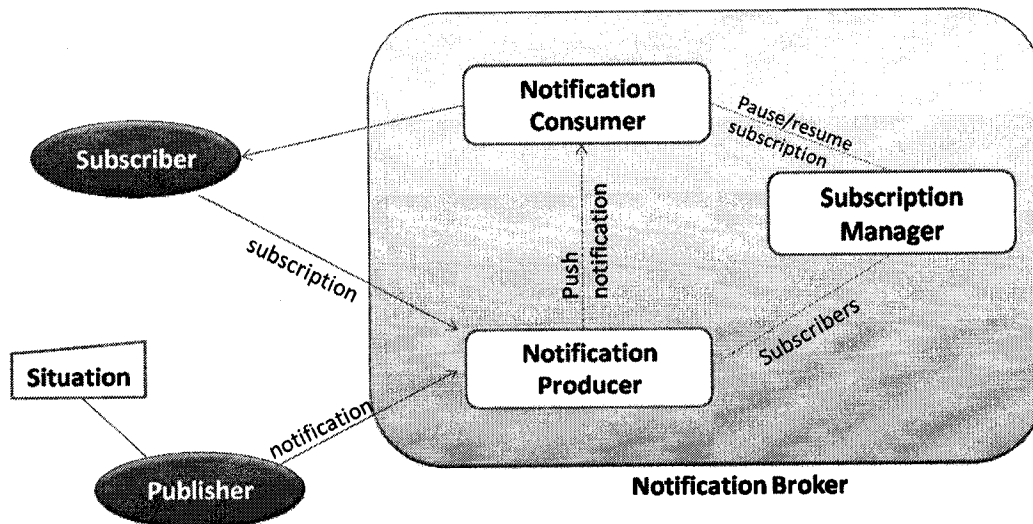


Figure 7. WS-BaseNotification and WS-BrokeredNotification roles and relationships

A *NotificationBroker* publishes messages for entities that are themselves not service providers. It can also interact with other *NotificationProducers* and *NotificationConsumers* based on the WS-BaseNotification specification.

WS-Topics (OASIS, 2006) define a mechanism for organising and categorizing messages for subscriptions. This it achieves through a hierarchical namespace that categorizes topics into a tree structured *Topic Space*. A Topic is then able to have child topics and may belong to a parent topic. Subscriptions to a parent topic, means an automatic subscriptions to all its child topics within the parent Topic Space. A Topic is defined by a name and a set of optional attributes. These includes *messageTypes* attribute, for defining the type of messages that the topic identifies, and *final* attribute, that allows a *NotificationProducers* to dynamically add topics. Other attributes are the *AliasRef* for masking topics on each other and *MessagePattern* for describing the pattern of the messages that are associated with a Topic.

WS-Notification is a complex specification (Pallickara & Fox, 2005), that is comprised of three other specifications, and links to WS-Resource Framework (OASIS-WSRF).

2.5.6 Web Service Eventing

According the (W3C, 2006) submission, WS-Eventing specification defines a protocol for subscribers to register interest through subscriptions to another Web service in receiving

messages. The specification addresses similar problem areas as WS-Notification. However, in this WS-Eventing, *Subscribers* subscribe, on behalf of event sinks, to event messages from event sources (publishers). Event sources also designate a subscription manager that is used by the sink to manage (renew, cancel) subscriptions (Figure 8)

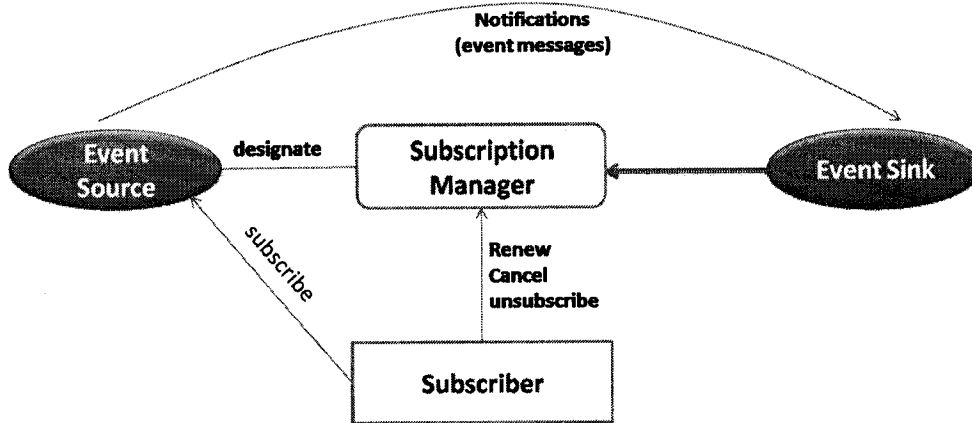


Figure 8. WS-Eventing components and interactions

Unlike the WS-Notification, this specification is self contained and does not depend on other specifications, making it less complex to implement. The simplicity of its structure makes it very attractive for inter-device communication since devices use this infrastructure for dynamic service discovery and subscription. The Devices Profile for Web services (DPWS) uses WS-Eventing for setting up and managing subscriptions to events from the device (Jeyaraman, Modi, Driscoll, Bullen, & Nixon, 2008; Microsoft, 2006). In addition to providing support for asynchronous push message delivery, WS-Eventing also supports pull, and message batching methods for message delivery. It also supports explicit unsubscribe operation, and non standard topic expressions (Pallickara & Fox, 2005).

WS-Eventing like WS-BaseNotification is so direct notification and therefore does not support message brokers like WS-BrokeredNotification. The source of events (publishers) tracks all of its sinks (subscribers) and is also responsible for forwarding notifications to the sinks.

2.5.7 WS-BPEL

Web Services Business Process Execution Language (WS-BPEL) is an OASIS standard. According to (Barreto et al., 2007) “WS-BPEL 2.0 is the revision of the original acronym

BPEL4WS (Business Process Execution Language for Web Services) 1.0 and 1.1". WS-BPEL is a language for orchestrating Web service interactions in a business process.

WS-BPEL is firmly tied to Web services. Business processes are encapsulated with interfaces for interacting with external services that can define a work flow of interactions. The process manifests itself as a Web service with external interfaces described using WSDL (Leymann, Roller, & Thatte, 2007).

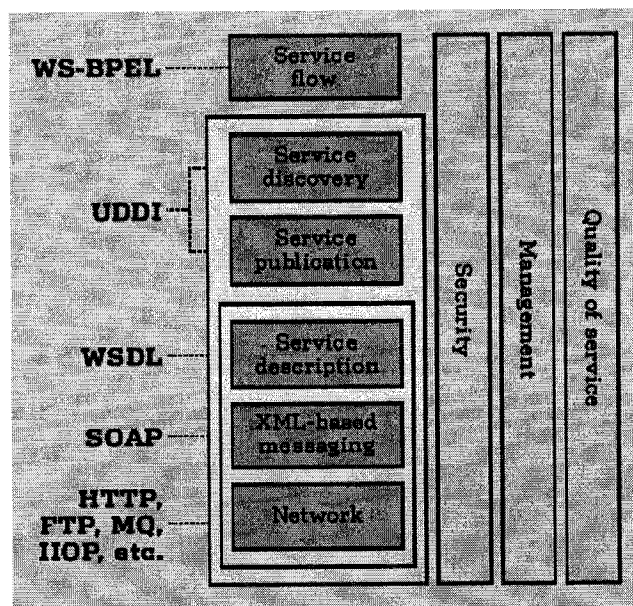


Figure 9. WS-BEL and the Web service stack ³

WS-BPEL defines business processes using an XML based language. It lies above the Web service stack (Figure 9), and uses Web service to define workflow processes. In orchestrating Web service interaction, each service interaction is viewed as an interaction link to a business partner. These links, described as *partner links*, are specified as WSDL *port types*. Each *partner service* can have a set of *partner links* that represent communication channels with the business processes. A `<partnerLink>` element has type and role attributes for identifying if the functionality provided comes from the business process or the partner service. Variables can be defined to capture messages, thus holding the state of the process at runtime. XML schemas

³ Source: (Pieczkowski, 2007)

attached to the process WSDL define complex types that describe these variables. Message exchanges between variables are done using XPath 1.0.

The business process invokes functionalities in the Partner services through the `<invoke>` activity. Conversely, the process receives message from external partners by accepting external calls through the `<receive>` activity. The business process then processes the message and returns data back to the external partner through the `<reply>` activity.

Executing the business process logic in specific order is done through the `<sequence>` activity. The `<if-else>` activity like traditional programming languages constructs allows branching of execution when given a set of choices. Repetitive activities can be encased in `<while>`, `<repeatUtil>` and `<ForEach>` activities. Parallel processing is supported through the `<flow>` activity.

Internal data manipulation is carried out using `<assign>` activity. It contains one or more `<copy>` constructs, specifying the variable from which messages are copied from, using the `<from>` construct as well as the variable to copy the message to, using the `<to>` variable. WS-BPEL also allows declarative XPath 1.0 queries through the `<query>` activity. Exceptions are handled through `<faultHandlers>` with one or more `<catch>` constructs and at most one `<catchall>` activity.

WS-BPEL handles selective events using the `<pick>` activity with one or more `<onMessage>` constructs that point to a Web service operation exposed by the business process and a variable to hold the received message. BPEL engines support various types of interactions as required by WS-BPEL: one-way, request-response, within stateful and long-running asynchronous communications. Support for long-running processes is provided through message correlation. Correlation helps the execution engine to relate messages to running business processes. BPEL implements correlation through *correlation sets*. It also support long-running process through WS-Addressing. Message headers are used to identify messages, the sender and a reference to the previous messages that the current message relates to.

WS-BPEL 2.0 utilizes several XML specifications including WSDL 1.1, XML Schema 1.0, XPath 1.0, and XSLT 1.0 (Scholar, 2007). The popular BPEL designers and engines include the

Apache Orchestration Director Engine (ODE), Netbean BPEL Designer/Glassfish, JDeveloper/OC4J, and ActiveVOS Designer/Active BPEL.

2.6 Policy Frameworks

Policy-based management offers flexibility in the automation and management of functions, processes and resources through defined rules. A policy is a declarative specification of guidelines, rules of conduct, organization, and behaviour of entities in a given environment (Ross-Talbot, Tabet, Chakravarthy, & Brown, 2003). It reduces the cost of resource management, and complications involved in managing controls by defining strictly regulated access to resources.

A policy framework describes the architecture for policy management (USG, 2001). It defines how policies are used, stored, and executed against system resources. For example in ubiquitous computing environments, these resources include a large number of devices, processes and applications participating in various B2B communications. Administrators can use policies to specify what actions that can or cannot be performed by the environment on the resources in response to certain events (Shankar, Ranganathan, & Campbell, 2005).

2.6.1 Web Services Policy Framework (WS-Policy)

Web Services Policy Framework (WS Policy) defines a set of constructs that can be used alongside other Web services specifications to describe a broad range of service requirements and capabilities for entities in a Web services-based system (W3C, 2007).

WS-Policy provides the syntax and semantics for service providers and requestors to use in describing their requirements, preferences and capabilities of their domains in a flexible and concise manner (Nolan, 2004).

In WS-Policy, a policy is a collection of policy alternatives. Each policy alternative is also a collection policy assertions. Policy assertions represent a requirement, a capability or other properties of behaviour. It could describe authentication scheme, encryption and transport protocol selection for message exchanges. It could also describe service selection and usage parameters such as privacy policy, and Service Quality characteristics.

As the basic unit of policy, assertions identify those behavioural requirements of a policy subject. Policy subjects represent an entity such as an endpoint, a message, a resource or an operation, with which a policy can be associated. Ignorable assertions are those assertions that can be ignored when there are policy intersections on a given subject or capability. Assertions can be nested in a parent-child fashion. A parent policy assertion of one security domain may also encapsulate other policy expressions from other domains.

In a Web service system based system, entities use policies to convey the conditions and requirements under which they function, or are able to participate in message exchanges. A policy assertion is supported by an entity in the system only if it satisfies the requirements and capabilities corresponding to the assertion. Conversely, if the entities satisfy the condition in the alternative policy assertions, the alternative assertion is supported.

Syntactically, a policy is made of assertions and nested combinations of the operators `<wsp:All>`, `<wsp:ExactlyOne>`, and the attribute *Optional*. Policies that include the *Optional* attribute or the `<wsp:ExactlyOne>` operator has more than one alternative. On the other hand, a policy built using only assertions and the `<wsp:All>` operator has no alternatives and therefore can be compressed into a single alternative and nested within other policies.(Nolan, 2004).

Entities need not understand all the tags that define available alternatives. They only need to find and use those alternatives they can support. This flexibility allows for incremental deployment of new assertions without affecting existing and operation policy assertions. However it does not describe complex relationships required in describing events, conditions and actions (Erradi, Maheshwari, & Tasic, 2007).

2.6.2 Event Condition Action (ECA)

ECA policy pattern is used by the majority of businesses that deal with complex sets of industry regulations, internal business processes and policies (Boglaev, 2005). Policy-based management systems use different forms of obligatory policies to guide system behaviour. Obligations refer to those actions that an entity must or must not perform on the occurrence of an event (Shankar, Ranganathan, & Campbell, 2005). Obligation policies are usually specified as ECA rules.

ECA rules automatically perform a set of actions in response to events provided that certain stated conditions are met (Bailey, Poulouvassilis, & Wood, 2002). ECA is the most acceptable pattern for building complex regulatory conditions (Boglaev, 2005) needed in setting up both internal and federated business processes. Rules-based policies govern system behaviour by providing reactive functionalities.

An ECA rule takes the form:

```
on    [Event]
if    [Condition]
do    [Actions]
```

[Event] define a situation or behaviour that causes a change in the active system being monitored. [Condition] define comparisons of the event properties or attributes to a certain set constants. [Actions] define those activities that must take place as a result of a successful or unsuccessful test of these conditions.

ECA rules are declarative, and are usually externalized and managed separately from the systems in which they apply to. This makes them very adaptable to a wide range of applications, such as adapting composite services with constantly changing business processes (Weigand, Heuvel, & Hiel, 2008).

2.6.3 XACML

XACML (extensible Access Control Markup Language), is an OASIS standard that describes a policy language for describing access control requirements in XML. According to (OASIS, 2003), XACML presents a common security-policy language that “allows the enterprise to manage the enforcement of all the elements of its security policy in all the components of its information systems”. It provides the flexibility needed in expressing and enforcing access control policies in a variety of environments using one language. XACML supports Role Based Access Control (RBAC) (Ferraiolo & Kuhn, 1992) and is integrated with the OASIS Security Assertion Markup Language (SAML).

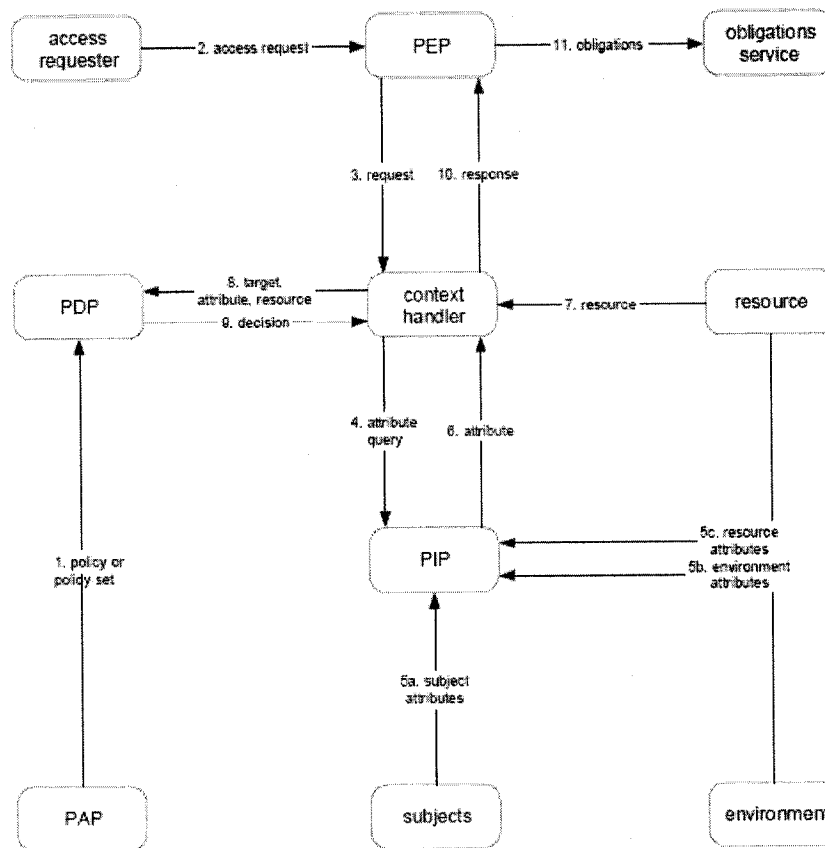


Figure 10. XACML Data-Flow Diagram⁴

XACML domain data flow model (Figure 10) describes how an access requester (*subject*) accesses (*action*) a security constrained *resource*. The requestor sends a request for a resource to a Policy Enforcement Point (PEP). The PEP then uses the *context handler* to retrieve attributes of the *subject*, *resource* and *action*, forms an *authorization request* based on the requestor's attributes, resource being requested and the action to be performed on the requested resource. This request is then forwarded to the *Policy Decision Point* (PDP). The PDP uses the *Policy Administration Point* (PAP), to retrieve applicable policies before executing the policies against the request. During the policy execution, the PDP may consult the *Policy Information Point* (PIP) through the *context handler* to retrieve the attributes of *subject*, *resource*, and *environmental attributes*. The result of the execution is then sent back to the PEP as an *authorization decision*. The PEP, based on the decision, may or may not grant access to the

⁴ Source: (OASIS, 2003)

request. At the same time, it may perform obligation services as a result the *authorization decision*.

The XACML policy language root element is a *Policy* or *Policyset* element (Figure 11). A *Policyset* is a container for other policies or *policysets*, each of which executes to an access control decision. A *policyset* combines these access control decisions through a *policy combining algorithm*. Each *policy* or *policyset* may contain multiple rules. Each of these rules like policies can evaluate to different access control decisions. These decisions are reconciled through the *rule combining algorithms*.

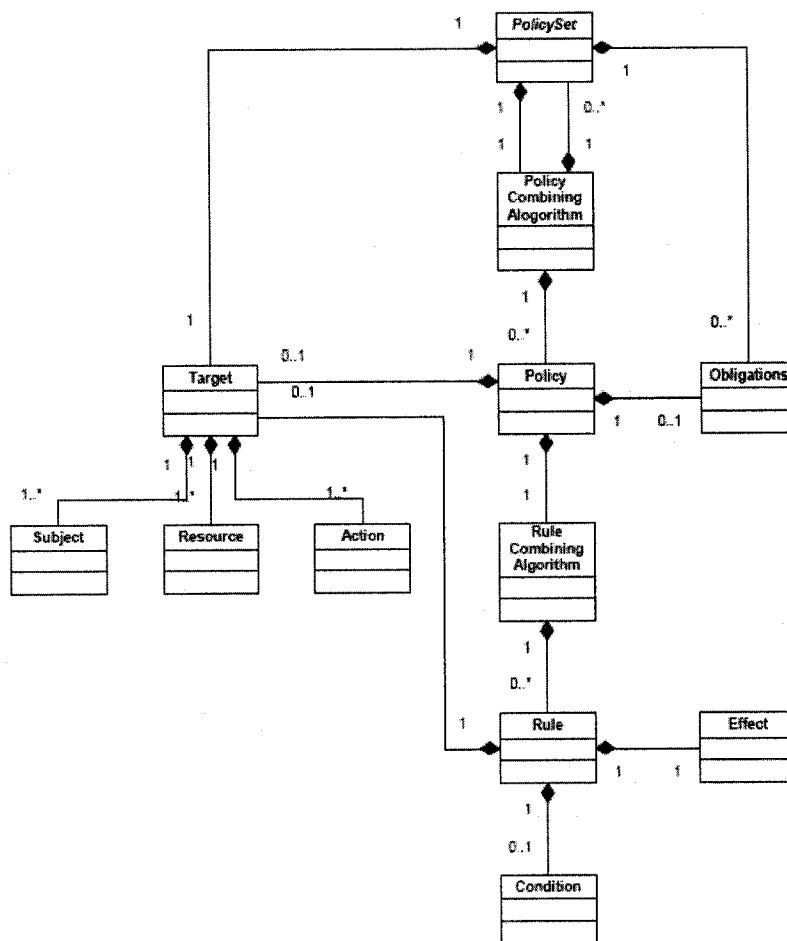


Figure 11. XACML Policy Language Model⁵

⁵ Source: (OASIS, 2003)

According to the specification, once the applicable policies have been found and verified to apply to a request, its rules are evaluated. Each *rule* has a *condition* that evaluates to *true*, *false*, or results in an error. When it evaluates to a *true*, then the rule's *effect* is a *Permit*. When it is false, the *effect* is a *Deny*, or in cases where are errors, it is *Indeterminate*. If none of the conditions apply, then it returns *NotApplicable*.

The PDP processes an access control by first determining the applicable policyset or policy. This is done through the *Target* element. A Target defines the conditions that must be met by the *subject*, *resource* and *action* for the *PolicySet*, *Policy* or *Rule* to apply to the request.

2.7 Related Works

The Liberty Alliance Project (LibertyAlliance, 2008) was formed in 2001 to establish open standards, guidelines and best practices for identity management in B2B networks. Over the years the alliance grew from 30 to over 150 participating organizations and hundreds of Special Interest Groups.

The goal of the project is to provide a framework for secure sharing of consumer identities and data using SOA and Access control technologies. This is achieved through a *Circle of Trust* (COT), in which cooperating organizations share data on their customers using pseudonymous identifiers. CoT allows businesses to share data on their customers while giving the customers the control over the sharing of such data (Shin, Ahn, & Shenoy, 2004).

The Liberty specification (C.Cahill, C. Canales, H. Le Van Gong, P. Madsen, & G.Whitehead, 2008) describes the infrastructure for achieving a COT. An *Identity Provider* (IDP) manages consumer's federated identity and *Single-Sign On* (SSO) across the enterprise. It allows consumers to define sharing permissions through a *Policy Decision Point* (PDP) while a *Discovery Service* (DS) helps service providers to discover and manage these permissions in order to provide the best service to their customers.

Liberty Alliance uses SOA as a building block to its data sharing infrastructure. Cooperating organizations must implement appropriate SOA and security infrastructure as defined by the specification in order to participate in data sharing. The framework manages permissions but

does not describe the routing of the actual data. Therefore inter-organizational data sharing is by direct, point-to-point interaction using customer pseudonymous identifiers coordinated by the DS.

The Integrated BPEL CoT framework (Doshi, 2008) extends the Liberty Alliance Framework by adding business process support to Liberty Alliance CoT. BPEL processes can use the framework to support business processes that span organizational boundaries in a B2B network that span multiple trust domains (Doshi, 2008).

The framework introduces the Trusted Process Provider (ITPF) to the Liberty Alliance CoT. The TPP helps service providers manage business process automation within a CoT. It leverages the CoT IDP for federated identity management. TPP also provides an *Audit Trail* service for logging the shared data for non repudiation purposes and legal compliance, *Event Stat* Service provides key performance indicators on running processes, and a *Task service* coordinates those steps that require human interactions.

The Integrated BPEL CoT framework approach data sharing using BPEL processes to orchestrate services from various service providers in a CoT. WS-BPEL is message-based but requires very strong coupling of the interacting Web services. It approaches B2B process integration in a CoT through a procedural-centric view. Unfortunately, this is not purely data-drive as the BPEL process integrates tightly with the data semantics. It does not provide runtime adaptation to changing data structures and relationships of business partners in B2B networks.

Chapter 3. Policy-based SOA Pub/Sub Framework

This chapter describes our thesis approach to building a framework to support flexible data-driven business process integration in B2B networks. In section 3.1, we describe the requirements for data-driven BPI. Section 3.2 describes the design considerations for the simple SOA publish/subscribe and our proposed data sharing model and extensions to the simple SOA publish/subscribe architecture. Section 3.3 describes the policy-based SOA publish/subscribe framework, the policy model and policy-based subscriptions.

3.1 Requirements for Data-driven Business Process Integration

Flexible data sharing is a major requirement for business process integration in B2B networks. The principle challenge of B2B collaboration is the difficulty in making disparate applications and processes operating in heterogeneous platforms to communicate and share data. The new realities brought about by globalization has shown that B2B networks are often times adhoc and need not conform to a single communication and messaging pattern (Phillipa & Arthur, 2007). Traditional integration solutions approach this multi-domain data sharing problem by implementing homogenous infrastructure and vendor compatible applications (Gisolfi, 2001). Coupled with inter-domain communications constraints, differences in protocols and communication patterns, there is the need for a data sharing model that supports resources in multiple domains(Stefansson, 2002).

Organizational processes characterized as data-focused, event-driven, dynamic and operate on reactive and heterogeneous environments like the healthcare services delivery are not well served by current B2B BPI approaches. Subsections below describe the requirements for a BPI framework to address such applications.

3.1.1 Information Discovery and Categorization

Information discovery is an important aspect of service interaction in B2B networks. Service providers need a mechanism for registering their availability and describing the type of service they provide with an information broker. On the other hand, consumers of these services need a mechanism for querying the information broker for available services, and a mechanism for obtaining and consuming services.

SOA approach this problem through UDDI, which not only serves as contract registry for services producers, but also provides an interface for consumers to discover available services, reference their contracts and retrieve information on how to bind and consume the services they provide (Section 2.5.3). Unfortunately, UDDI registry data can be quite complex as it drills down from business details to service interface descriptions and binding details. UDDI search focuses on basic identifiers like service name, business name, discovery URL, category, etc but fails to provide the ability to discover service interface data descriptors and meta-data. The simplicity of its search is usually not adequate for business integration needs (Brogi, Corfini, & Popescu, 2005; Liang-Jie, Haifei, Henry, & Tian, 2002). Also the gap between discovery and service invocation still makes its use unpopular (Brogi, Corfini, & Popescu, 2005; Mckendrick, 2000) for B2B integration.

Unlike UDDI, publish/subscribe approaches this problem by focusing less on binding and more on categorizing the information being disseminated. Topics are used to categorize messages, and further categorization can be done through content attributes. An important distinction between the procedural information discovery offered by UDDI and the event-driven information discovery used in publish/subscribe is the elimination of transport and information binding details. A simple indication of interest through subscriptions is usually adequate for data sharing.

3.1.2 Platform Independence

Enterprises often align to different integration technologies and platforms for automation, resulting in non-homogenous applications that do not share a common messaging model. Proprietary technologies like Microsoft DCOM, OMG CORBA, and Sun JMS do not interoperate. Web services runs on top of these technologies to provide a platform independent

communication framework, leveraging most Internet transport protocols. It also creates the necessary wrapper for legacy systems to inter-connect with existing applications.

Effective data-driven business process integration needs to ride on a platform independent messaging framework. SOA through Web services uses XML, a language independent representational format, to encapsulate application messages as well as leverage existing transport protocols to integrate disparate applications.

3.1.3 Configurable Communication Patterns

Traditional application interaction is usually strongly coupled and point-to-point. The calling application waits for the called application to respond before continuing its own execution. Strong coupling results in fast execution when all interacting services are online and responsive. However, whenever there is a disruption to communication as a result of internal or external factors, the entire execution process fails.

The communication patterns used in B2B process integration should be flexible and also resilient to disruptions and service binding details. Synchronous point to point communication pattern is very popular with BPI because of its simplicity, but results in tight coupling of applications. Asynchronous broadcast-based communication pattern on the other hand is built around decoupled execution and stateless message exchanges. In this type of interaction, each message is treated independently of the other, while pattern tracking, message numbering and correlation are used to relate message to process execution progress.

Some of the strengths of publish/subscribe are the abilities to support asynchronous message delivery, content-driven interactions, broadcast-based communication and anonymity through decoupling of collaborating business processes and applications. In some applications, these flexibilities can also become a bottleneck. The broadcast nature of message distribution usually results in message flooding; content-filtering is dependent on message sizes and can introduce some latency and delays to the system.

A good data-driven BPI framework should be able to control message flooding through controlled publishing, targeted message delivery, context-aware filtering, and increased message matching accuracy. It should also support load balancing, restrictions to message sizes,

notification frequency, flexible subscriptions to messages, as well as security and privacy of shared data.

3.1.4 Service Quality

QoS is often used to refer to simple quality aspects of a service, but actually has a precise meaning with respect to networks (Wikipedia, 2008), we use the term Service Quality (SQ) to mean a resource reservation control mechanism and allocation measure that applies to different applications, users and processes. In the context of this thesis, Service Quality defines those measures for controlling, allocating, handling and communicating messages especially as it relates to message delivery. Subscriptions without Service Quality definition are based on “Best Efforts” delivery. Other Service Quality attributes include delivery reliability, message batching on failure delivery and delivery acknowledgement. Service Quality focuses on availability, and reliability of software processes in a B2B network.

3.1.5 Surveillance Infrastructure

In a B2B network, there is need for active surveillance and auditing of information flow between collaborating businesses in the network. BPI requires the ability to mine this data for trend analysis, threshold monitoring, and to provide timely reporting. Traditional publish/subscribe systems deals with data on the atomic level, as stateless entities (Dufour, 2007) without the ability to track trends, correlate existing message and take care of compensations when required. A surveillance infrastructure provides stateful data for analysis, reporting and monitoring in an event-driven system. It also offers a means of documenting legal and regulatory compliance through audit trail reports (Peyton, Doshi, & Seguin, 2007).

3.1.6 Dynamic Business Configuration

Organizations need to be increasingly agile in today’s business environment. Data sharing and BPI frameworks need to be flexible enough to accommodate fast changing requirements by allowing the business rules which control sharing of information to be declaratively defined and dynamically configured. A BPI framework in dealing with the flow of information between businesses needs to support a variety of message types, flexible context-based access control

rules, as well as active message filtering and transformation to conform to changing business goals and requirements.

3.1.7 Security

Security and privacy of data is an important aspect of data-driven business process integration. As we strive to find methods for fostering flexibility in data sharing and information dissemination, ensuring that data streams are not compromised as they get routed within a B2B network is an important concern. Message exchanges across the Internet are susceptible to manipulations. The following security requirements are important for message exchanges:

- **Integrity:** Messages should not be tampered with en route transmission to the intended recipient. There is need for a mechanism for appending trust signatures on data before transmission.
- **Authentication and Authorization:** Access control to resources is an important concern in data-driven business process integration. Communicating processes, devices, and applications require the ability to authenticate and authorize shared resources by implementing appropriate access control infrastructure.
- **Confidentiality:** It is not very complex to support data tunnel security (Damiani, Vimercati, Jajodia, Paraboschi, & Samarati, 2003) for routed and shared data especially in centralized applications. However, decentralized systems and adhoc B2B networks present unique privacy concerns because of the fast changing and dynamic rules of association and constraints. Data sharing for such dynamic relationships require considerations for time-dependent access even within trusted organizational domains. Confidentiality of shared data requires a model that combines data security with dynamic access control policies that are congruent with privacy considerations.

3.2 SOA Publish/Subscribe Framework

Service Oriented Architecture (SOA) through Web services provides a framework for business process integration, and universal access to data and applications. The goal of SOA is to provide a platform for heterogeneous service integration through Web services. Conversely,

publish/subscribe as a communication pattern is designed to exploit the functionality of an underlying transport-level protocol. Our proposed SOA Publish/Subscribe framework attempts to integrate the best of both these paradigms in order to provide flexible, dynamic, data-driven business process integration in a B2B network.

Figure 12 provides a simple representation of SOA publish/subscribe architecture. It uses a Web service middleware (SOA Message Broker) to interconnect service providers and consumers using the publish/subscribe pattern. Service providers push data to this Web service, when they become available, and the broker in turn pushes the data to consumers that indicated interest to the published data. The publishers are not necessarily Web services but any application that implements a web service client using the WSDL of the SOA Message Broker. To support this, the SOA Message Broker uses a Web service endpoint that implements interfaces used by the publishers to receive SOAP messages, which is then broadcast to the registered subscribers based on their subscriptions.

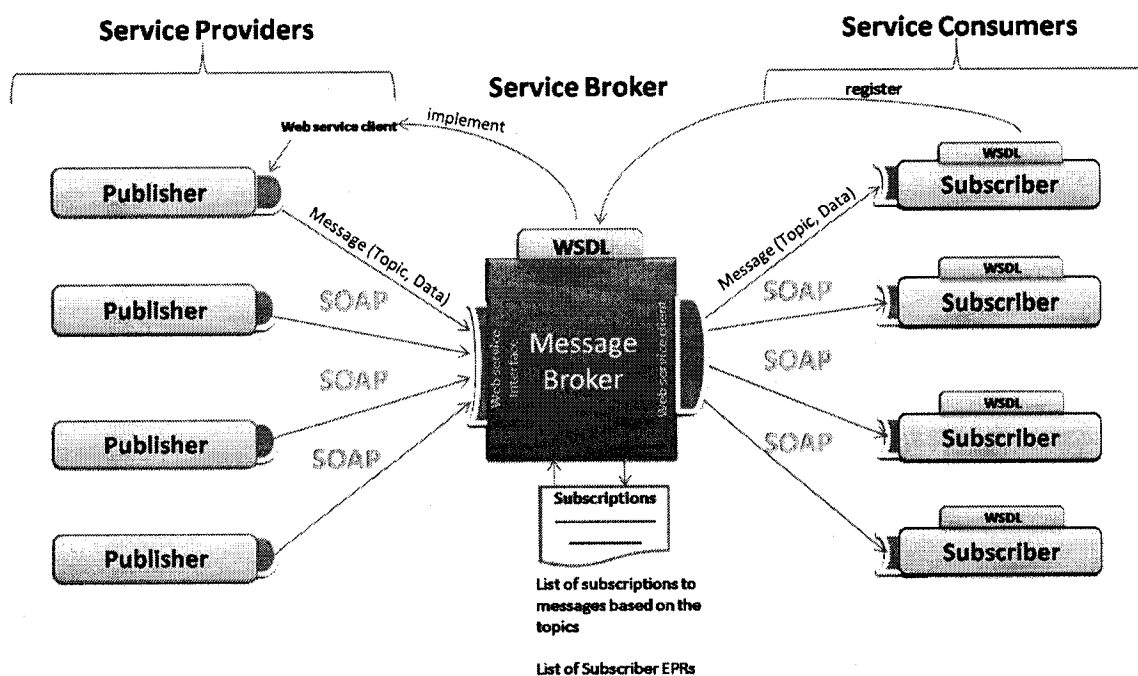


Figure 12. Simple SOA Publish/Subscribe Model illustrated

In addition, the SOA message broker keeps track of subscriber endpoint references and subscriptions to topics representing the pattern of messages from publishers. This simple model of SOA publish/subscribe instantly brings the benefits of publish/subscribe to Web service interactions.

3.2.1 Simple SOA Message Broker

The simple SOA message broker implements two groups of service interfaces to support publishers and subscribers (shown in Figure 13). The message broker's WSDL describes the publisher and subscriber interfaces needed for the implementation of subscriber Web services and the publisher web client components. The BPI design considerations required in implementing this prototype Web service message broker includes:

- A mechanism for defining and categorizing data using topics;
- Implementing automatic binding of Web service endpoints;
- Supporting asynchronous communication from the message broker to subscribers;
- Having a common information representation model that is published to subscribers and publishers through the message broker;
- Supporting a security framework for secure registration and subscription.

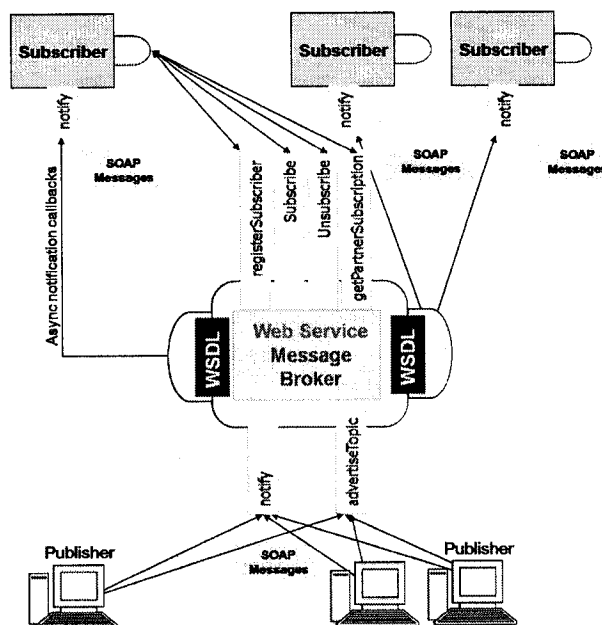


Figure 13. Simple Web Service Message Broker

The publishers interact with the message broker through the *advertiseTopic* and *notify* interfaces while the subscribers use the *registerSubscriber*, *Subscribe*, *UnSubscribe* and *getPartnerSubscription* interfaces to interact with the message broker. Table 1 describes the communication and message types supported by each of these interfaces.

Table 1. Summary of the Message Broker interfaces

Service Interfaces	Description	Input Parameters	Return Partners
registerSubscriber (Synchronous)	Subscribers register to the broker by calling this interface with their contact information, and ServiceEPR and in return receive a Security Token after a successful registration.	Partner Contact, ServiceEPR	Security Token
Subscribe (Synchronous)	Uses SecurityToken, ServiceEPR and topics name as parameters, subscribers can subscribe to topics published on the broker. Each successful subscription is represented by a fresh security token.	Security Token, ServiceEPR, Topic	Security Token
Unsubscribe (Synchronous)	Similar to <i>Subscribe ()</i> but allows a subscriber to remove its interest to a topic.	Security Token, ServiceEPR, Topic	Status (String)
notify (Asynchronous)	Used by publishers to notify the broker of new messages.	Message	None
advertiseTopic (Asynchronous)	Used by Publishers to alert the broker of the availability of new topics.	Topic (String)	None
getSubscriptionTopics (Synchronous)	Provides the available events to all parties.	None	List <Topic>
getPartnerSubscriptions (Synchronous)	Retrieves the subscribed events of a subscriber based on its identifier.	Partner Identifier	Topics

In our framework, we extended this simple SOA message broker architecture to offer more abstraction to its integration with applications as well as its description and handling of messages. Subsection 3.2.2 presents our relational data model for SOA publish/subscribe. Subsections 3.2.3– 3.2.7 describe additional components that were added to the simple SOA publish/subscribe to support the identified requirements to data-driven BPI in B2B networks.

3.2.2 Data Sharing Model

A data sharing model provides the basis for supporting BPI requirements for information discovery, categorization, communication patterns, service quality and dynamic business configuration. Our data sharing model is illustrated in Figure 14; data is being published and subscribed to by applications, devices and in some cases business processes. The topic registrar is a service that defines as well as describes message topics to the message broker.

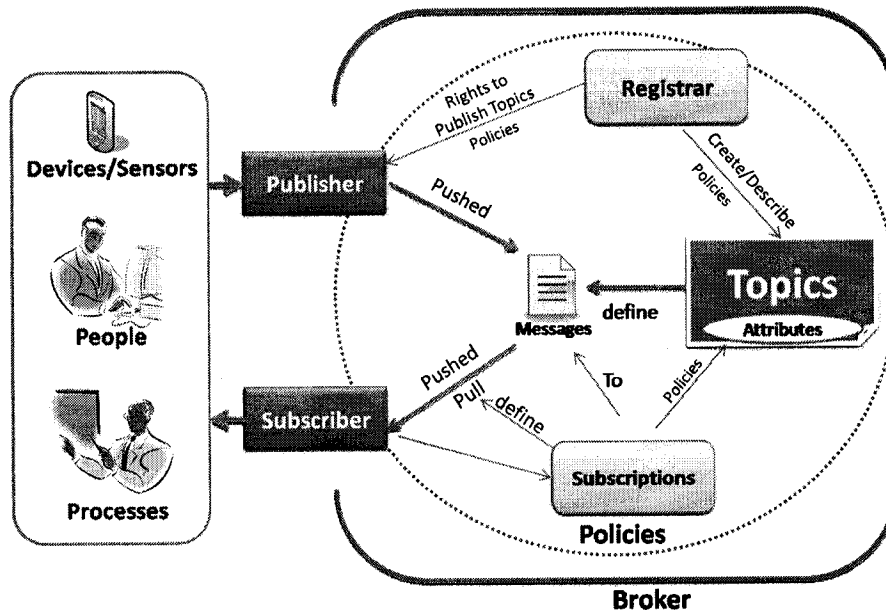


Figure 14. SOA Publish/subscribe data sharing model

In addition, the registrar could optionally define policies that restrict message publishers to defined topics. Subscriptions on the other hand describe interest to messages. Like topics and messages, policies are used to describe this interest flexibly, as well define attributes for message transformation, aggregation, and delivery Service Quality. Messages are categorized by their topics. They are pushed by publishers and then broadcast to subscribers based on their subscriptions.

3.2.3 The Partner Service Component

In a B2B, it is normal for applications to be both publishers and subscribers of data. In addition, applications in a B2B usually do not support a common communication platform, and messaging pattern. In our simple SOA publish/subscribe architecture this requires the subscribing

applications to implement a Web service endpoint while the publishing applications implement the SOA message broker's WSDL as a proxy to its interfaces. Coupled with that, these applications need to implement components and message structures for communicating registration, subscriptions, topics and notification messages with the SOA message broker. Implementing these functions for each collaborating application, could be quite complex to setup, manage and debug.

The Partner Service (PS) Component is an application that assumes both the roles of a publisher and a subscriber by integrating all of these features in a single installable application adapter (Figure 15). It is both a web service client implementing the WSDL of the message broker and a Web service endpoint that can be called by the message broker. In addition to providing the interface, it also implements the message broker's messaging model, shielding participating application from its details. The PS component uses handlers to manage messages to/from the message broker. The *Publisher Handlers* packages these messages, before binding and transmitting them to the message broker. The *Subscriber Handlers* packages these messages, before binding and transmitting them to the message broker.

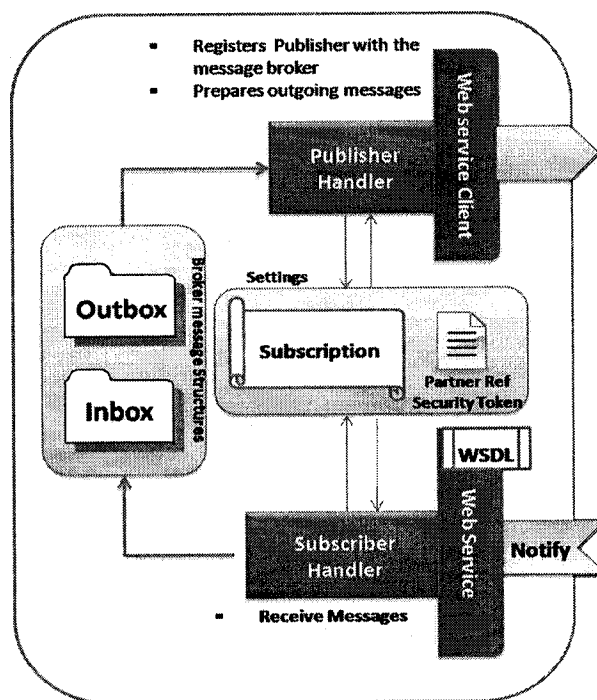


Figure 15. The Partner service component

The PS component also presents an interface that is used by other applications to submit and retrieve their application specific messages from the message broker. As shown in Figure 15, it presents an outbox-inbox model to external applications. Applications drop off messages in an *outbox* and check for in-coming messages through the *inbox*. Depending on the implementation, the PS component could be set to monitor the *outbox* for out-going messages. On the other hand, the *Subscriber Handler* receives messages from the broker and relays them to the interacting applications through the *inbox*.

3.2.4 Partner Registration

Partner Service registration is the first step towards providing applications and processes the framework for participating in event-driven BPI. The Partner Service component uses a declarative configuration file for tracking its *PartnerReference*, and *SecurityToken* which are required for authenticating and encrypting its messaging with the SOA message broker. The file also contains the component endpoint address (*PartnerEPR*), the endpoint address of the message broker (*BrokerEPR*), and a list of subscription references.

Before a Partner Service is registered, it is issued a *RegistrationToken* generated by the policy broker and recorded against the PS *Partner EPR* (Figure 16). To initiate the registration process, the PS component binds dynamically to the referenced policy broker, and sends its *RegistrationToken* and *PartnerEPR*.

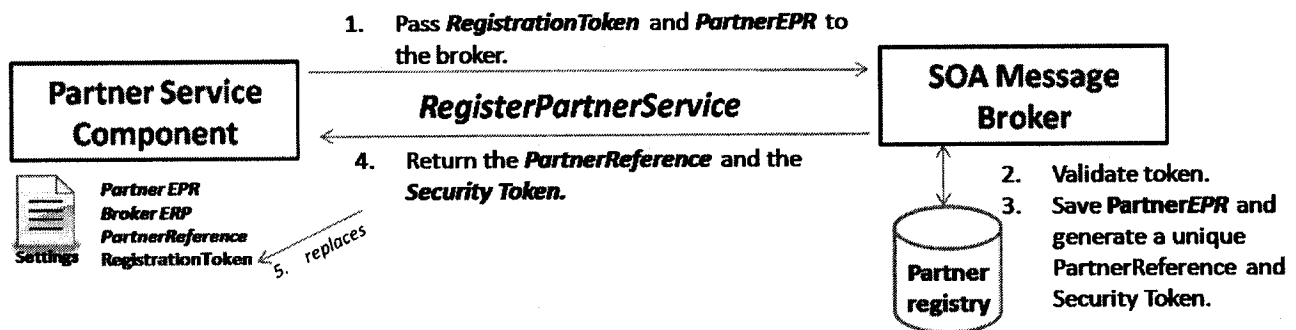


Figure 16. Partner Service Registration with the Policy Message Broker

The broker captures this information and validates the token as a means of authentication. For a valid token, the policy broker generates a unique *PartnerReference* and *SecurityToken*, creates an

account for the PS and then responds with this new dataset. The PS captures this back into the configuration file, replacing the *RegistrationToken* with the *SecurityToken*. The broker will only accept an unencrypted *RegistrationToken* for brand new PS registrations.

Unlike the simple SOA publish/subscribe where subscribers are identified by their WSDL addresses and subscriptions reference these as identifiers to subscribers, subscriptions in our framework identify PS components by their *PartnerReferences*. Therefore, a PS component can refresh its registration whenever it desires without affecting subscriptions referencing it. Reasons for registration renewal include changes in deployment endpoint address, service migrations, domain changes and arbitrary security reasons.

To perform registration renewal, a PS simply repeats this process but with its current security token, this time encrypted since it has a partner reference. Once validated the broker then refreshes the PS component details, renews the security token and returns this new token to the registering component.

3.2.5 Partner Service Communication Authentication

Security is important for message exchanges especially in a publish/subscribe communication pattern. Figure 17 illustrates how the Partner Service component uses security tokens to provide secure communication with the message broker.

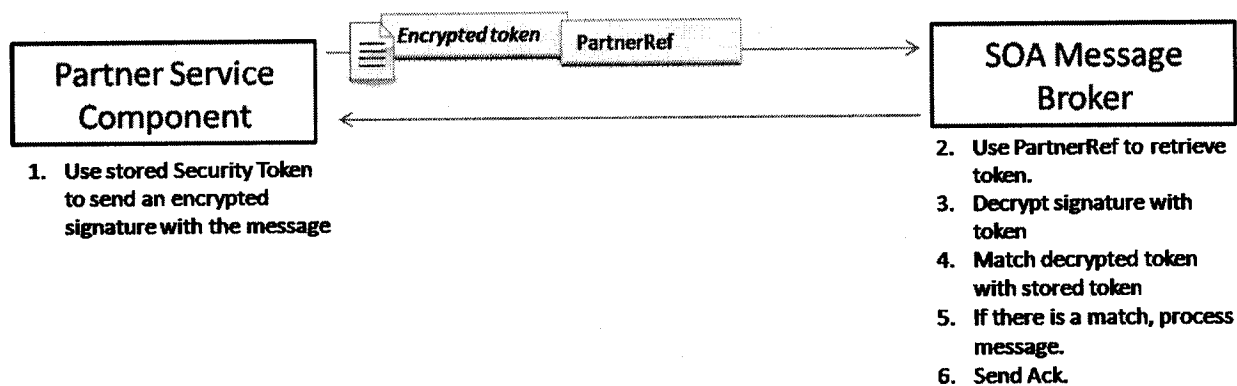


Figure 17. Security Token use illustrated

To achieve secure communication with PS components, the policy message broker needs to uniquely identify and authenticate the source of each request and associated messages. For each

exchange with the PS component, the broker requires the component to encrypt and transmit its security token alongside its unencrypted *PartnerReference*. Before processing an administrative request or notification message, the message broker first identifies the PS component using this unique identifier, it then retrieves its security token and uses this token to decrypt the encrypted token contained in the message. If the decrypted token matches the original token, the request is processed, actions are taken and the appropriate acknowledgement returned.

3.2.6 Surveillance Infrastructure

Our framework views a B2B as a streaming database, publishers producing data while subscribers subscribe to them. The message broker handles the routing of data streams from source to destinations. A surveillance infrastructure is integrated into our framework to analyze, report and audit streaming data.

The surveillance infrastructure is a special Partner Service component that automatically subscribes to streaming data and topic creation events for the purpose of message logging and reporting. When a topic creation request is received from a *TopicRegistrar*, the SOA message broker creates the topic and subsequently sends a message to the surveillance infrastructure which in turn creates a representational table using the topic attributes. The surveillance infrastructure then creates and returns a subscription to the topic back to the message broker which allows it to capture and log all data published on the topic. This data forms the basis for stateful data processing, aggregations, auditing, report generation, pattern triggers and threshold analysis on the data streams. The surveillance infrastructure is also able to process data queries for requesting subscribers.

3.2.7 Topic Registrar

Topics are the basis for information discovery and categorization. In our framework, topics are created using special Partner Service components called the Topic Registrars. Topics are identified by a name, and described by a schema (attribute value pairs) that represents the structure of its messages. The topic registrar could attach an optional policy identifying the PS components that are authorized to publish messages on the topic. Figure 18 illustrates the process of creating a topic in our framework.

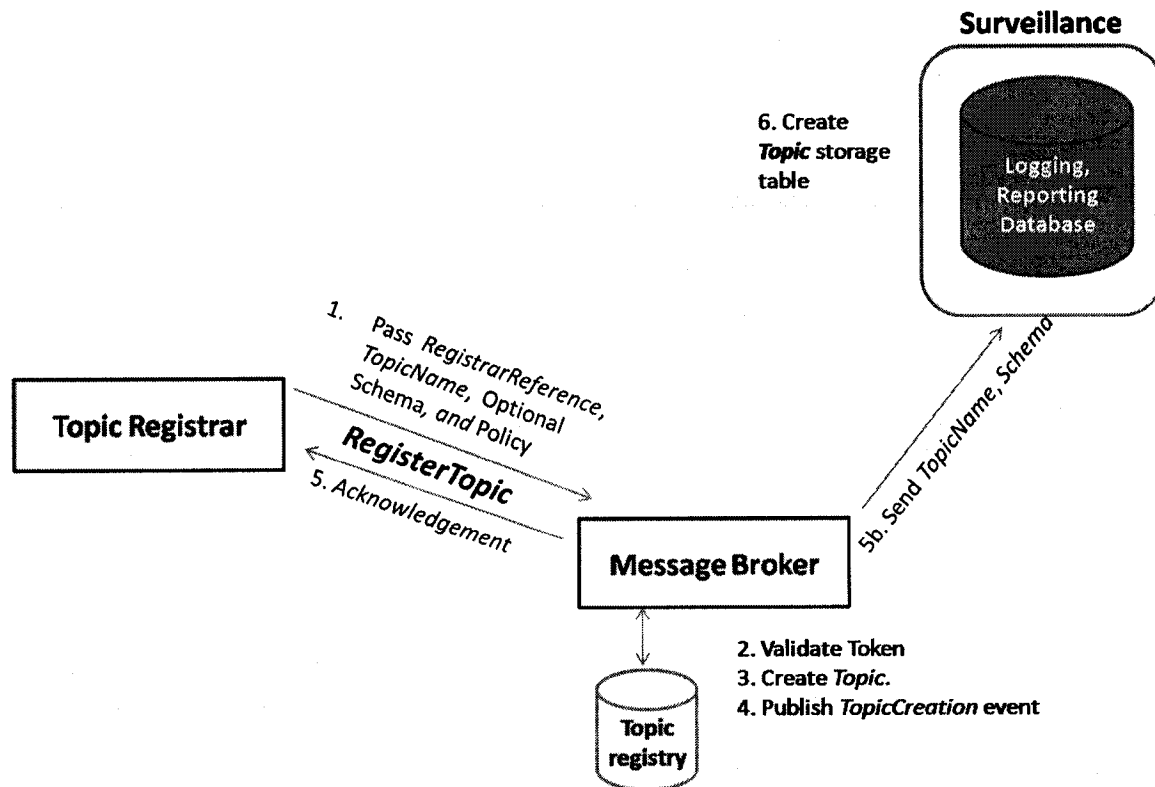


Figure 18. Topic Registration Process

During the topic registration process, the topic registrar sends the topic data set to the message broker. The message broker after validating the sender creates the topic and then publishes a *TopicCreation* message that gets sent to the surveillance infrastructure. It then returns an acknowledgement back to the topic registrar. The surveillance system uses the details of the *TopicCreation* message to dynamically create tables that will be used to capture and log messages on this topic.

3.3 Policy-based SOA Publish/subscribe

In section 3.2 we described the simple SOA publish/subscribe architecture and showed how our framework extended it through a data sharing model that incorporates a Partner Service component, a Surveillance infrastructure and a Topic Registrar. In this section, we describe the flexibilities added to SOA publish/subscribe through a declarative policy model. Combined with the extensions described in section 3.2, our framework uses this policy model to realize the

requirements for data-driven BPI integration discussed in Section 3.1 especially those required for dynamic business configuration (Section 3.1.6) and message delivery service quality (Section 3.1.4) in event-driven BPI.

Our framework views a B2B network as a streaming data model with processes requesting and accessing streaming data in real-time. The goal is to mimic the XACML RBAC architecture but applied to a streaming event infrastructure. Our major goal is achieve a flexible and expressive subscription to data. Expressiveness of subscription is an important component of a flexible publish/subscribe framework. The simplicity of topic based subscriptions, as described in section 3.2, limits the ability of our initial framework to appropriately express interest to data. Conversely, content-based subscriptions result in message processing overheads and can greatly affect the performance and scalability of a publish/subscribe infrastructure (Opyrchal, 2004). Message brokers need to tie content-based filters in the context of statically configured topics (Eugster, Felber, Guerraoui, & Kermarrec, 2003). It implies that we use content filtering only when the message topic matches a statically configured topic. Our approach tries to marry topic-based publish/subscribe with content-based filtering.

In addition, publish/subscribe has little support for the service quality of message delivery. Messages are pushed to subscribers without a means of validating receipt. (Niblett & Graham, 2005) identified the need for message brokers that support Service Quality options like reliable message delivery, logging, and batching of messages. Subscriptions need to support declarative expressions for message transformation, delivery service quality, constraints to publishers and recipient subscribers as well as timing and scheduling. Timed subscriptions reduce down-times, increases predictability, end-to-end timeliness and foster a more flexible form of information dissemination.

3.3.1 Policy-based SOA Message Broker

Our framework uses declarative policies to provide expressiveness to subscriptions. Figure 19 gives a generalized view of the framework. People, devices and sensors, surveillance infrastructure share real-time data in adhoc B2B networks using an SOA publish/subscribe model. Because of differences in technology, capacities and running platforms, the Partner

Service component interacts with these applications through an application message *inbox/outbox* type interface.

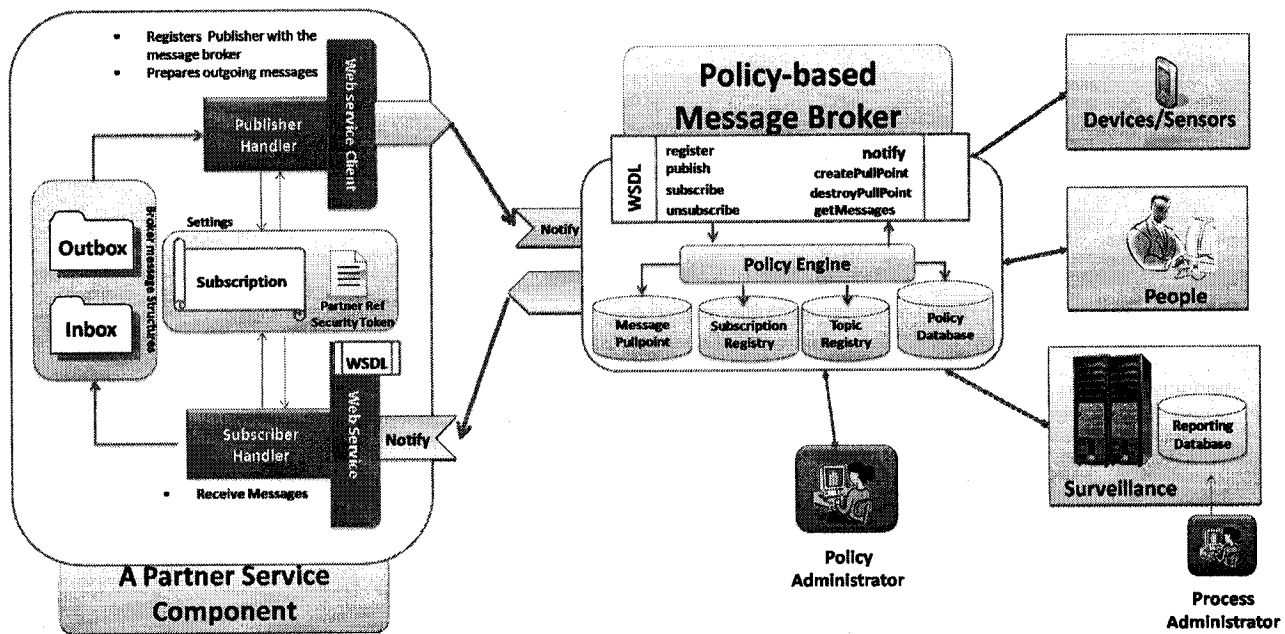


Figure 19. Policy-based SOA Publish/Subscribe Framework

In our policy-based framework, the simple SOA message broker is extended to include a policy database and a policy execution engine. This database stores policy-based subscriptions while the policy engine executes message events against these policies.

The policy administrator represents a service linking the data sharing requirements for collaborating processes in a B2B to organizational policies and procedures. This is accomplished through dynamically changing rules of association for B2B processes represented by flexible subscription policies.

The Policy-based SOA Message Broker is an SOA message broker with a policy engine for parsing and executing policy-based subscriptions. It builds on a messaging ontology that can be used synchronously or asynchronously. To achieve this, it uses a single *PortType* with three messaging interfaces:

- **sNotify** for synchronous messaging,

- *aNotify* for asynchronous messaging and
- *aNotifyResponse* for asynchronous acknowledgment.

In the synchronous mode, received messages are forwarded to a *message context handler* for processing and waits for an acknowledgment which is sent back to the calling application on the same channel. In the asynchronous mode, message are received asynchronously, passed to the same *message context handler* which then processes the message but return an acknowledgment by calling the *aNotifyResponse* interface of calling Partner Service.

Figure 21 shows the policy-based message broker architecture. It follows a totally data-centric model to message handling. We apply context to messages to determine what actions to perform on the messages instead of linking them to static WSDL interfaces. This allows incremental deployment and the addition of new functionality without breaking existing functionalities.

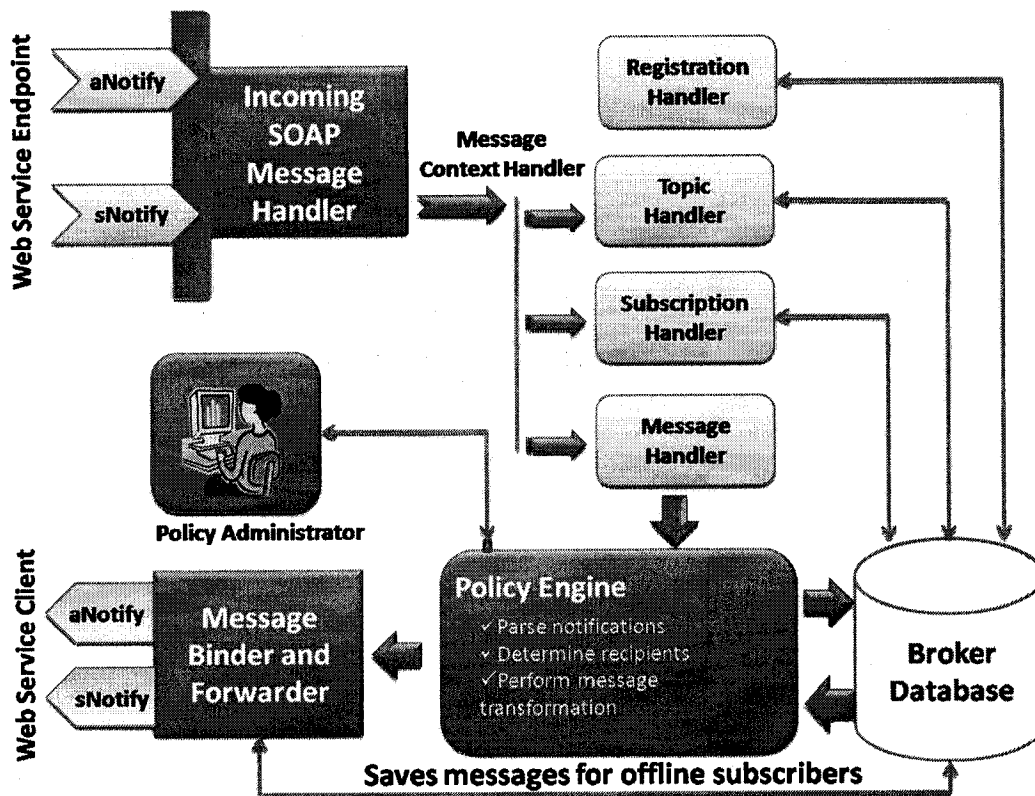


Figure 20. The Policy-based message broker model

We treat data simply as structured XML documents that is not coupled to any specific operation. Publishing endpoints attach their global identifiers to data which is then used by the message broker to return the appropriate responses after processing. Subscriber and the broker identify the operation to be performed on a request or an acknowledgment by using the *message context handler* to check the type of message before using the appropriate low-level handler to process it and subsequently generate a return message which can either be a response or a fault message (Figure 21).

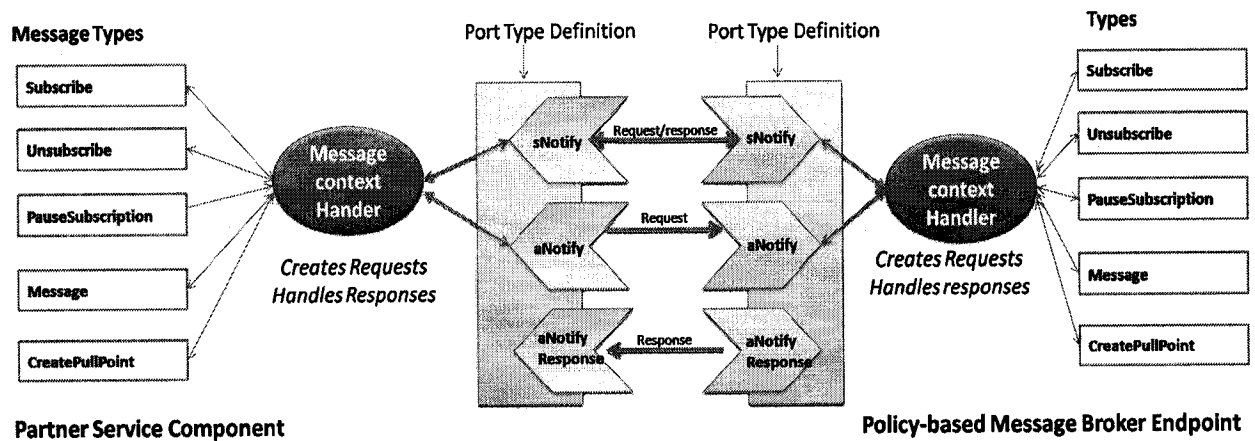


Figure 21. Data-centric Notification Pattern

Table 2 shows the currently supported message types for the requests and responses. The detailed schema is available in Appendix A.

Table 2. Notification Message Types

NotificationRequest		NotificationResponse	
Message is a choice of ...		Message is a choice of...	
Requests	Message	Responses	MessageResponse
	Subscribe		SubscribeResponse
	UnSubscribe		UnSubscribeResponse
	PauseSubscription		PauseSubscriptionResponse
	RenewSubscription		RenewSubscriptionResponse
	GetMessages		GetMessagesResponse
	RegisterTopic		RegisterTopicResponse
	UnRegisterTopic		UnRegisterTopicResponse
	RegisterPartnerService		RegisterPartnerServiceResponse
	UnRegisterPartnerService		UnRegisterPartnerServiceResponse
	DataQuery		DataQueryResponse
	CreatePullpoint		CreatePullpointResponse
	DestroyPullpoint		DestroyPullpointResponse

The benefit of this data-driven messaging model is that Web services using SOAP operate as the transport layer protocol and is used to relay messages from the source to a destination. Having message validation and processing done at a level above the Web services layer enables support for incremental development, deployment and modification of the message broker's operations without affecting existing functionalities.

We now introduce the policy model and language used by our framework to describe policy-based subscriptions.

3.3.2 Policy Model

The policy model used by our framework is based on XACML Role-based Access Control architecture. As described in Section 2.6.3, XACML uses an XML-based interchangeable policy format that is independent of implementation to describe conditions for accessing security constrained resources. Especially important with XACML is its support for context-based access control that relies on the control and management of semantic information from multiple channels (Tzelepi, Koukopoulos, & Pangalos, 2001). In our framework, a modified XACML context is used to define compliance policies that business processes must conform to as it relates to rules of data sharing in a streaming database.

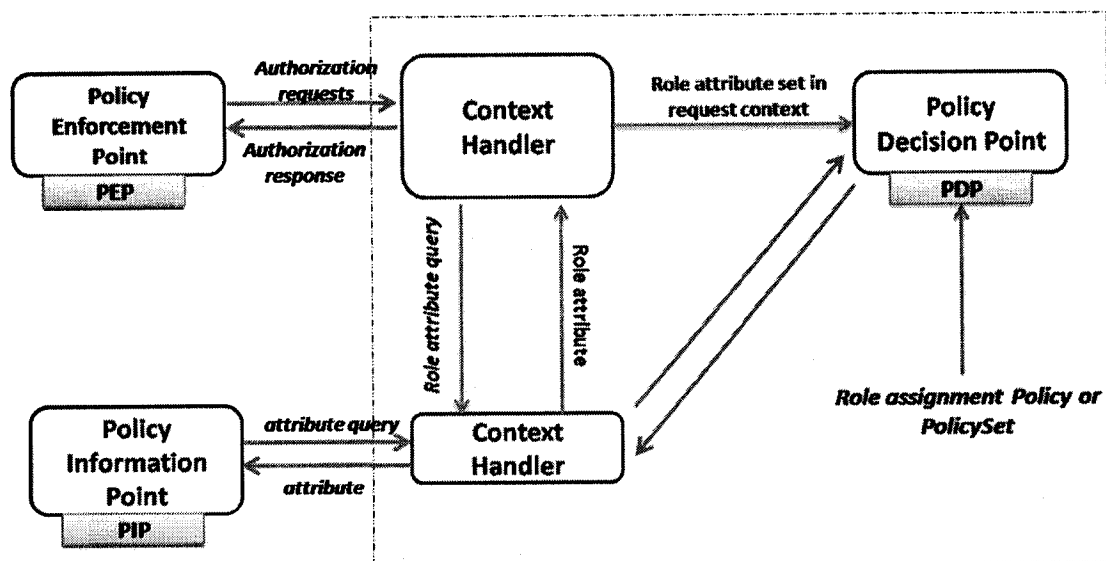


Figure 22. XACML Role-based view

XACML RBAC is very similar to our policy-based data-sharing infrastructure. We illustrate these similarities in Figure 22 and Figure 23. Section 2.6.3 describes in details the components of XACML RBAC infrastructure. When a device (subject) requests access to a security constrained resource, the Policy Enforcement Point (PEP) sends an authorization request to the Policy Decision Point (PDP) through a context handler. The context handler appends the relevant attributes of the subject as well as the resource and forwards the request to the PDP for authorization. The *PDP* then executes XACML policies to determine if access is granted or denied.

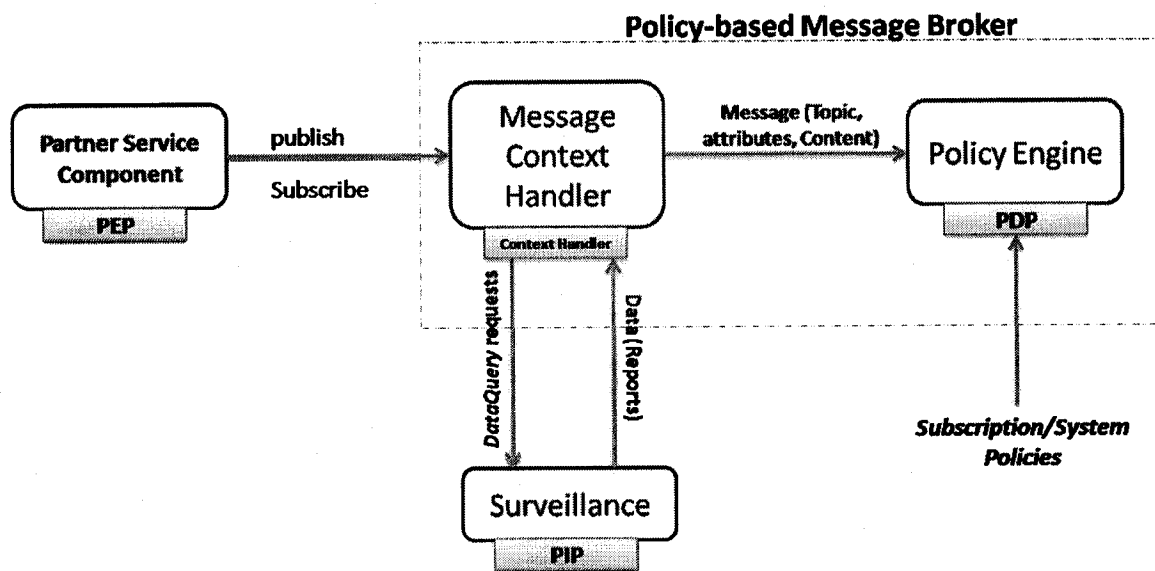


Figure 23. Policy Framework

Like XACML policies, our policy language is based on ECA rules. B2B applications use Partner Service (PS) Components to publish and subscribe to streaming data. Like the PEP, the PS component handles all requests, message exchanges and acknowledgments from the policy-based SOA message broker. In this context, messages are events to be executed against subscription policy rules.

The policy-based SOA message broker uses a message context handler to handle streaming data. Message events are executed against policies, which like XACML are set of rules containing specific conditions which can results in some actions being taken in response. The actions dictate message transformation rules and delivery service quality. Like XACML where PDP uses the

Policy Information Point (PIP) to get context and environmental information on the subject or resource, the Policy-based SOA message broker communicates with a Surveillance infrastructure to provide access to stateful data, reports, alerts and historical information that are not defined in streaming data.

3.3.3 Policy Language Model

The structure of the policy language is illustrated in Figure 24. Each *<Policy>* element uses a type attribute to describe the policy type. In this model, a *<Policy>* defines a set of rules, described by the *<Rule>* element. Each rule uses an optional *<Schedule>* as well as *<ExpirationTime>* to define the rule application schedule and expiration period.

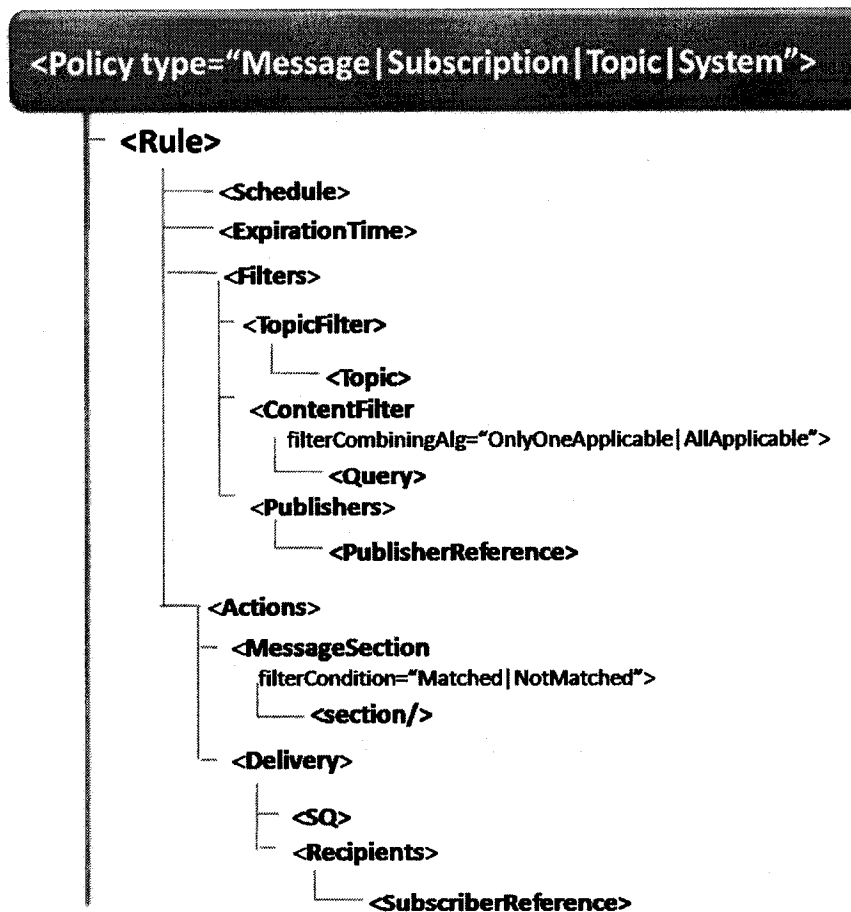


Figure 24. Policy Language representation

A rule supports *topic*, *content* and *publisher* filters. The `<TopicFilter>` element contains one or more `<Topic>` elements that define the topics for messages that it applies to. `<ContentFilter>` element contains one or more XPath queries defined in `<Query>` elements and can be combined on the basis of the filter combining algorithm. Each `<Query>` is an XPath expression that executes to a Boolean result. If multiple queries are defined, the results of individual `<Query>` execution results are combined on the basis of the *filterCombiningAlg* (Table 3). The result is either a “Matched”, “NotMatched” or “Indeterminate” response. The `<Publisher>` element contains a list of allowed publishers described by the `<PublisherReference>` elements.

Table 3. Filter Combining Algorithms

filter Combining Algorithm Conditions	
<i>filterCombiningAlg</i> values	Conditions
<i>AllApplicable</i>	1. If all queries results in a match, a "Matched" result is returned. 2. If any of the queries results in a "NotMatched", then "NotMatched" is returned. 3. If there is an error, the "Indeterminate" result is returned.
<i>OnlyOneApplicable</i>	1. If any of the queries results in a match, a "Matched" result is returned. 2. If there is an error, the "Indeterminate" result is returned.

`<Actions>` describe message transformation and rules for message delivery. Message transformation depends on the result of filter execution. For an “Indeterminate” filter result, no further actions are taken on the message. For the “Matched” or “NotMatched” conditions, message transformation rules can be defined using the `<MessageSection>` elements. Each `<MessageSection>` contains one or more `<section>` elements with XPath queries that execute to an XML Node or Node List. It uses the “*filterCondition*” attribute to define if the `<MessageSection>` is applied to a “Matched” or “NotMatched” filter conditions.

When defined, message transformation results in a new message for delivery. When not defined, the original message is delivered without manipulations. `<Delivery>` element defines message delivery service quality. The `<SQ>` element values can be set as “BestEfforts”, “Reliable” or

“Pullpoint”. *BestEfforts* is default delivery mode. That is, the message is delivered without the need for a positive acknowledgement from the recipient. If defined as “Pullpoint”, then all messages are queued for a later delivery. In “Reliable” delivery mode, a message must be delivered with a positive acknowledgement or it automatically gets queued to be delivered at a later time. Finally, the <Recipient> element lists the policy-based message broker issued identifiers for recipient components. This is used for those special circumstances that require directed message notification.

Subscription policies are parsed and stored in the policy database (See Appendix B). The policy engine executes applicable policy rules on incoming messages. The language supports the definition of multiple rules for various applications in the same subscription.

3.3.4 Types of Policies

The policy framework described in this thesis uses policies to extend an SOA publish/subscribe with added flexibilities and functionalities.

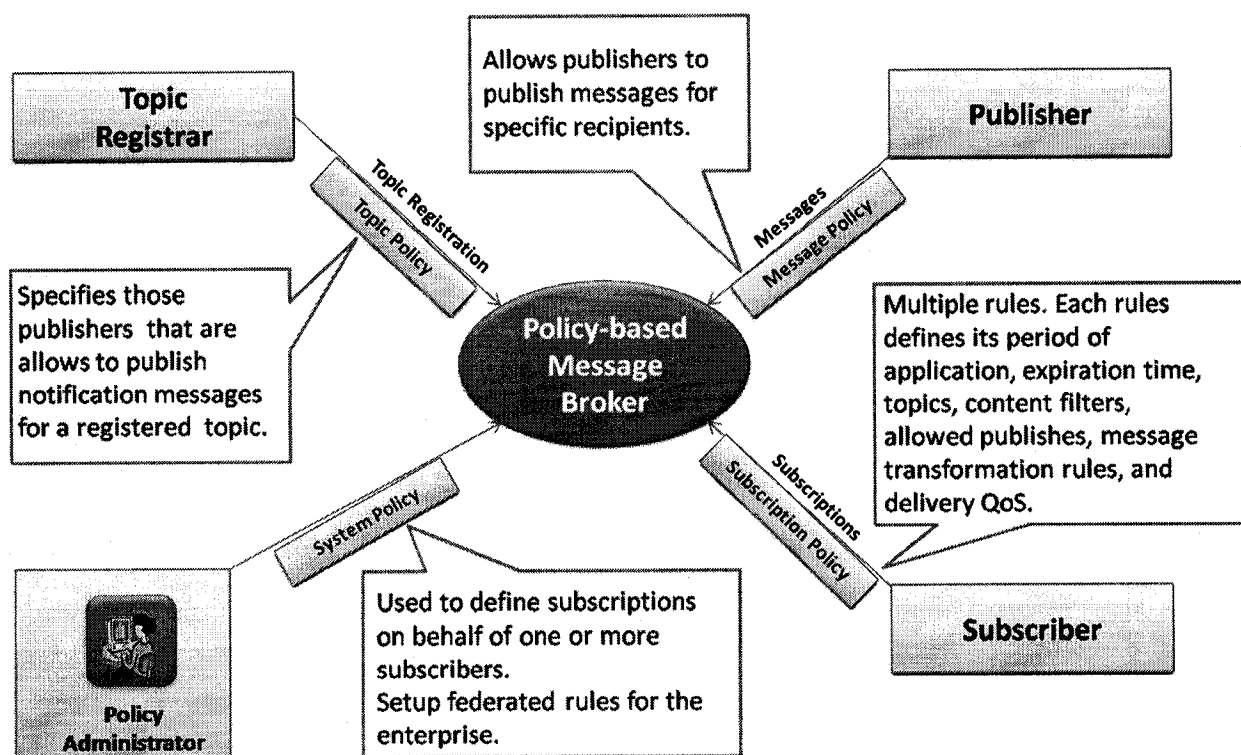


Figure 25. Policy Types

As described in Figure 25, our framework supports four types of policies:

1. **Topic Policy:** Topics describe a message to the message broker. Traditional publish/subscribe does not provide a mechanism for controlling the publishers of information. A *Topic Policy* provides a means of controlling “who has access to publish what”. Topic creation is controlled by delegating this responsibility to *Topic Registrars*.
2. **Message Policy:** In the simple SOA publish/subscribe, messages are sent off from publishers to be routed to subscribers based on their subscriptions. Therefore publishers of messages have no mechanism of controlling the recipients. Messages are simply broadcast to subscribers based on their subscriptions to a topic. In B2B, it is often required that certain communication be restricted to select parties. Message policies describe a means of achieving this flexibility in publish/subscribe. They define intended recipients to a message, thus allowing us to support various variants of broadcast type messaging as well as point-to-point messaging using publish/subscribe.
3. **Subscription Policy:** Simply stated, subscriptions are used by subscribers to indicate interests to a type of message usually by the topic or other meta-data as seen in content-based subscription. Our framework extends subscription using the policy language described in 3.3.3 to support scheduling, a more efficient combination of topic and content filtering, context-based message transformation and delivery Service Quality.
4. **System Policy:** It is often not appropriate security wise or in terms of management of data-driven rules of association to rely on applications to describe their interest to data appropriately. System policies are subscriptions policies subscribed on behalf of a subscriber or subscribers by a policy administrator. System policies help define federated rules for data sharing with an enterprise view to its application. They can be automated and tied to an organization security policy and operations.

3.3.5 Subscription Management

The strength and flexibility offered by our framework is tied to its declarative subscription policy model. Subscriptions management is done through subscribe, renew, pause and unsubscribe operations.

▪ Subscribe

PS Components use subscription policies to track the data sharing preferences and requirements of the applications they represent. As illustrated in Figure 26, the subscribe operation is carried out by sending the *PartnerReference*, the *SubscriptionPolicy* describing the details of the subscription and an optional *SubscriptionReference* (for renewals). After the usual PS component validation, the broker sends the subscription to the policy engine for parsing and storage. It then acknowledges with the *SubscriptionReference*. This resource identifies subscriptions and can be used to refresh the subscription policies by the PS whenever desired.

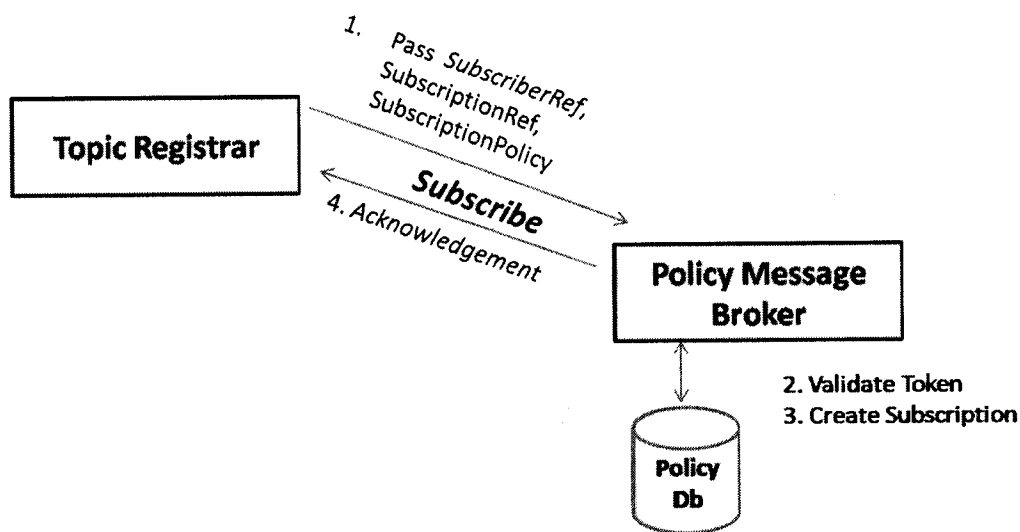


Figure 26. Subscribe operation

▪ Renew/Pause/Unsubscribe Operations

PS components can *Renew* expired subscriptions, *Pause* a subscription or *Unsubscribe* by simply sending the *SubscriptionReference* with operation to the policy broker (Figure 27). The policy broker processes the request by changing the status of the referenced subscriptions and returning an acknowledgement.

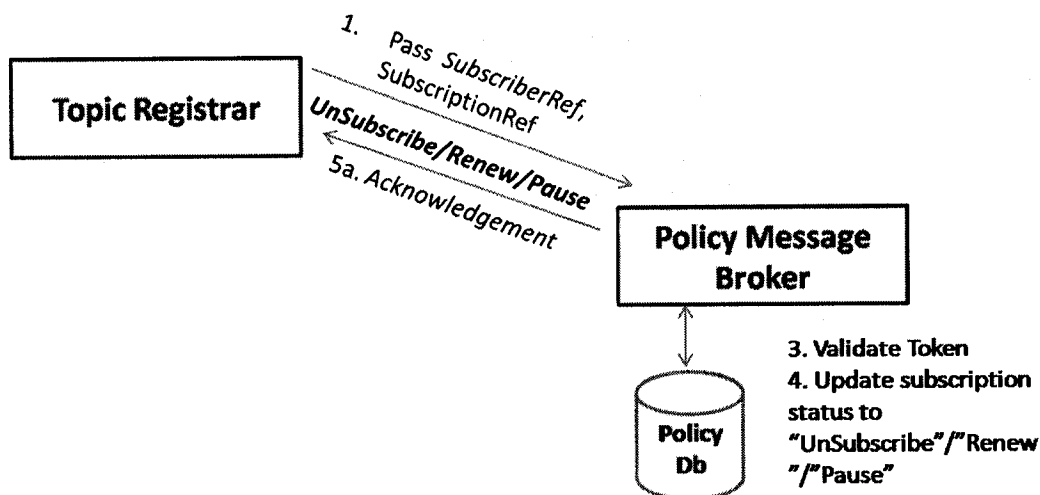


Figure 27. Renew/Pause/Unsubscribe operation

3.3.6 Notification Message

Notification messages are handled as wrapped data structures containing Meta information for message identification, routing and management. The Meta-data include *PublisherReference* identifying the sender of the message, *Topic* of the message, *ContentType* identifying the type of payload, *MessageAttributes* which is a set of attribute value pair containing summarized details of the message payload, the message payload (encrypted using the security token) and an optional policy for controlling the routing of the message.

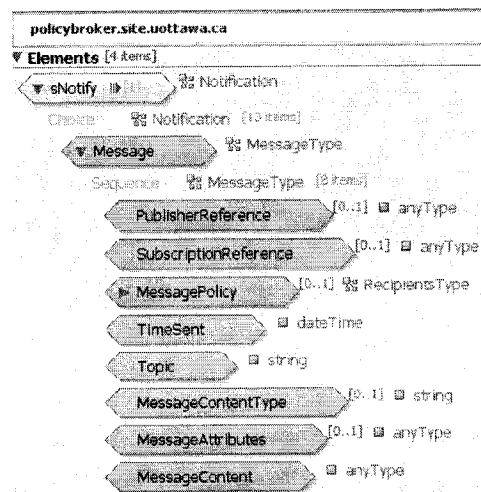


Figure 28. Wrapped Message Structure

Applications that are part of the enterprise send raw application data to the PS component. The component then wraps the application data in the wrapped data structure represented in Figure 28. To secure the application message from alterations, it is encrypted using the PS component security token before being published to the policy broker.

When a notification message is received by the policy broker, the sender is authenticated similarly like other management messages. It is then passed to the policy engine for processing. Figure 29 shows the processing steps for a message by the **policy engine**.

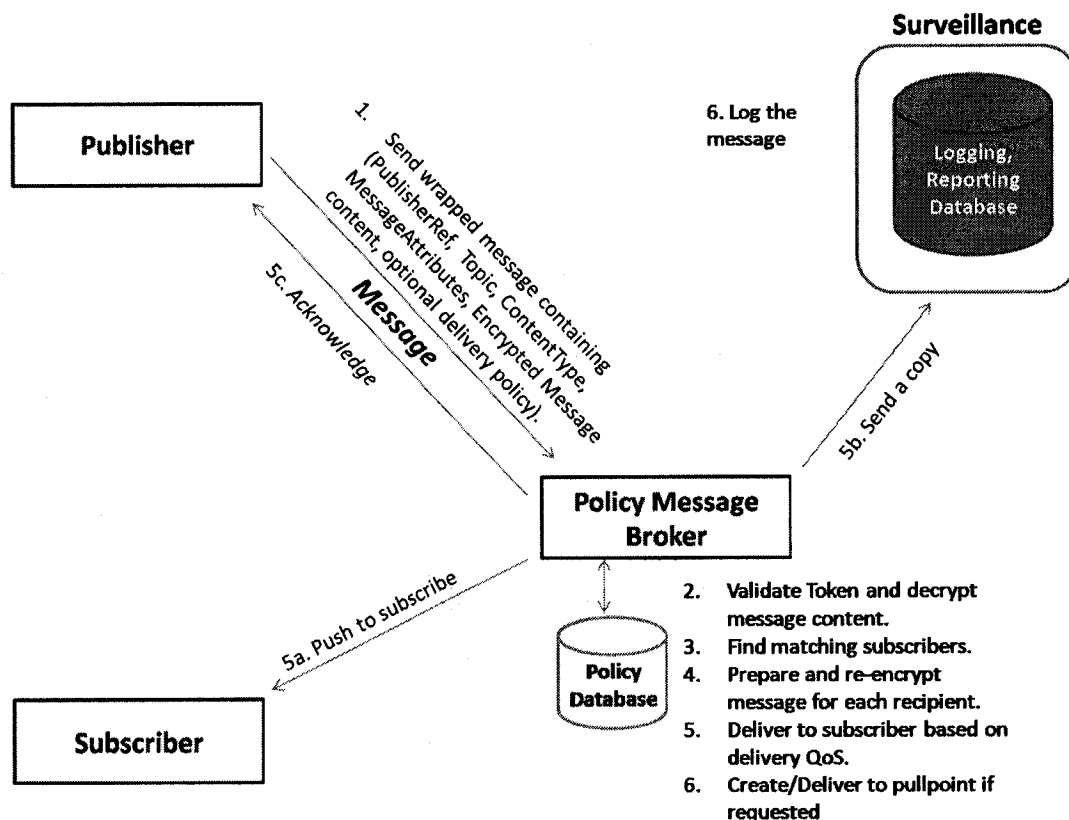


Figure 29. Notification Message Processing

The policy engine passes each message through the following stages:

1. Message-based filtering

This stage of the process tries to evaluate the message against the topic and message policies for matches.

- a. **Message Decryption:** Retrieve and use the publishing PS component security token to decrypt the message attribute and content.
- b. **Topic Policy processing:** Check if there is a topic policy attached to the topic of the message. If there is, retrieve this policy and validate that the sender is authorized to publish on the topic. If the publisher is not authorized, the message is discarded.
- c. **Message Policy processing:** Check if the message contains any policy. If there is one, pull the list of the attached recipients.

If the message passes through steps b and c, then the execution from stage 1 will output the original message and optionally a list of subscribing recipient PS components.

2. Subscription-based Filtering

This stage uses the output from the previous stage to determine the subscribers to the message based on their interests. The optional list of recipients if presented helps accelerate this process as the steps below will take place only for applicable subscribers. If the recipient list is empty then subscription-based filtering is applied on the message. The filtering process steps include:

- a. **Topic Filtering:** the policy engine then pulls all the subscribing PS components to the topic of the message and their subscription policy references. The resultant list is then passed on for optional content filtering.
- b. **Content Filtering:** For every matching subscription policy, the content filters are executed against the message attribute using the specified XPath queries. At this stage, both positive and negative results are important and are returned with the subscription policy rules.

Stage 2 returns a list of matching subscription policy rules which are essentially recipients of the message depending on the context of the message defined for the subscription.

3. Message Transformation

This optional stage builds the delivery message for each of the recipients. Subscription policies can optionally define message transformation rules. These transformation rules are based on the “*Matched*” or “*NotMatched*” conditions of the subscription-based filtering stage. These rules are XPath queries that execute to a XML Node or Node List. If transformation rules are absent, the original message is sent on to the recipient for a “*Matched*” condition.

At the end of this stage of execution, the policy engine then packages a new set of messages with either the payload of the original message or a new but transformed message, ready to be dispatched. It then sends to the dispatcher each of these messages, with the recipient PS component and the subscription reference that resulted in the message.

4. Message Dispatching

The dispatcher is responsible for pushing messages to the intended recipients. To support this process, subscription policy rules may define delivery Service Quality conditions: “BestEfforts”, “Reliable” and “Pullpoint”. Using the recipient PS component *PartnerReference*, the dispatcher retrieves the service address of the endpoint, binds to it and sends the messages. Depending on the defined QoS, it may review the component acknowledgement, retry or queue the message in a Pullpoint.

3.3.7 Managing Pullpoints

Pullpoints define a temporarily message storage (Yi Huang & Gannon, 2006) for a subscribing PS component on the policy broker. Pullpoints help ensure that messages are not lost because the recipient is offline or too busy to handle message notifications. In WS-Notification (OASIS, 2004), pullpoints must be defined explicitly before messages can be forwarded to it. Our framework uses pullpoints for this purpose but extended its use by attaching it to delivery Service Quality. Therefore, pullpoints are alternate delivery locations for undeliverable messages whose subscriptions require “Reliable” delivery. They can also be defined declaratively as the message delivery point either through the subscription policy or by sending the *CreatePullpoint* request to the policy broker (Figure 30).

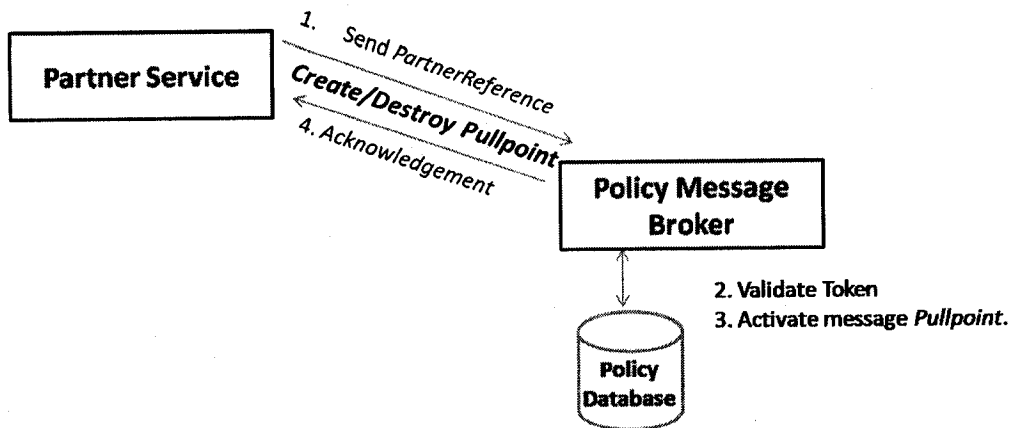


Figure 30. Create Pullpoint operation

Having a pullpoint overrides other delivery service quality parameters referenced in a subscription. PS components retrieve messages from their pullpoint in batches and when no longer needed, the pullpoint is destroyed through an explicit *DestroyPullpoint* request.

3.3.8 Batched Message Retrieval

Stored messages are retrieved from pullpoints storage using the *GetMessages* command. The policy broker in response wraps messages up to the specified batch size and delivers them to requesting PS component (Figure 31).

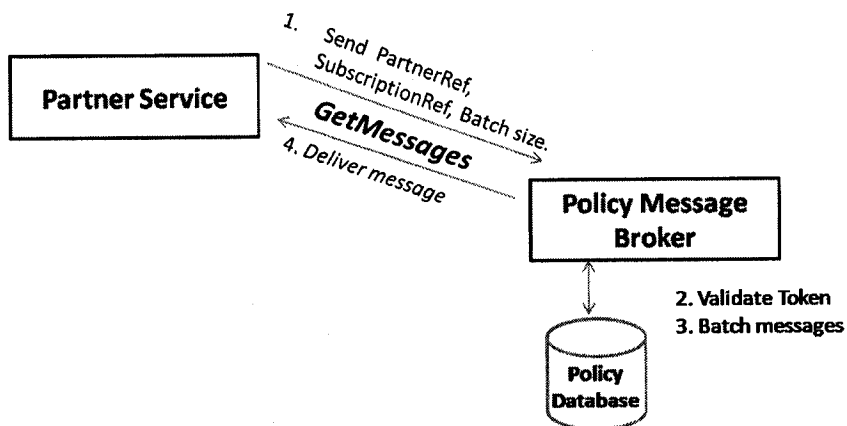


Figure 31. Batched message retrieval illustrated

Batch message delivery is dependent on the type of request made by the requesting PS component to the policy broker. A synchronous request receives an immediate response while asynchronous requests are acknowledged asynchronously.

3.3.9 Data Queries

Publish/subscribe push data to subscribers on an on-going basis and as the data becomes available. As much as this is a big advantage over the pull based systems, it fails to provide the ability for applications to pull data or reports from other applications in the enterprise when desired. Our framework supports a *DataQuery* operation to allow applications to make explicit requests for data from the policy broker or other components like the surveillance infrastructure.

As shown in Figure 32, a requesting PS component sends a *DataQuery* request passing along the *ProducerReference*, its *PartnerReference* as consumer reference, a description of the data requested and one or more attributes or parameters.

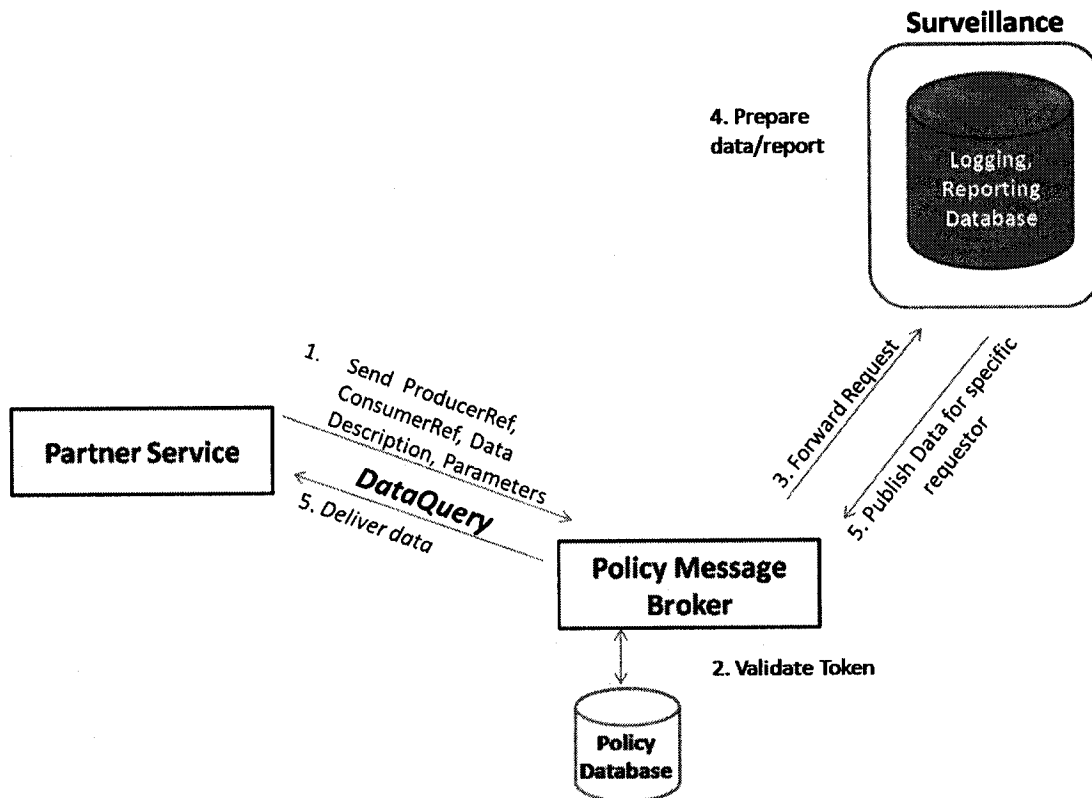


Figure 32. The use of data queries

The policy message broker validates the sender's credentials and forwards this request to the referenced producer for processing. In some cases, the request may be for internal broker data like list of subscriptions, topics, etc. Processed data is returned as a published message with a message policy indicating the consumer as the recipient.

3.4 Summary

In this chapter we discussed our policy-based SOA publish/subscribe framework. Our analysis highlighted the following critical points to our approach:

1. Identified the essential requirements for flexible data sharing and BPI in a B2B network.
2. Described the simple SOA publish/subscribe architecture, emphasized its weaknesses in terms of expressiveness of subscriptions, lack of dynamic control and flexibility in the handling of messages. We showed how our framework extended this architecture through a data sharing model that incorporates a Partner Service component, a Surveillance infrastructure and a Topic Registrar.
3. Proposed a common, flexible, streaming, relational, B2B data model for event-driven BPI based on streaming events organized into topics and their attributes.
4. Proposed a policy model and language that is based on XACML syntax, for defining broadcast and target messages as well as flexible subscription policies that combine topic and content filtering, message transformation, and delivery service quality.
5. Proposed a policy-based SOA publish/subscribe architecture. Policy-based approach uses subscription policies to provide full flexibility to SOA publish/subscribe in support of dynamic business configuration.

Next section will illustrate this with a case study in palliative care.

Chapter 4. Case Study

A Palliative Severe Pain Management case study was used to validate our policy-based SOA publish/subscribe framework. In doing so, we setup and analysed four different architectures:

1. Web Application Portal,
2. Integrated Circle of Trust (CoT) BPEL,
3. Simple SOA publish/subscribe (first iteration of our framework),
4. Policy-based SOA publish/subscribe (final iteration of our framework).

In Section 4.1, we introduce Palliative Care. We then narrow our focus by describing a particular scenario for palliative severe pain management independent of any BPI framework in Section 4.2. In Section 4.3, we discuss and analyze a Web Application Portal, proposed for a Local Health Integration Network (LHIN) in Ontario. In Section 4.4, we discuss and analyze the implementation using an Integrated CoT BPEL framework. This is followed by our initial SOA publish/subscribe implementation of the scenario (Section 4.5). Finally, in Section 4.6, we show the implementation of the scenario using our final policy-based SOA publish/subscribe framework. To further highlight the advantages of our final framework, we show how it can handle an extended scenario involving a Roaming Nurse as well as multiple roles and groups within the B2B network.

4.1 Palliative Care

Palliative care is a form of medical care that strives to improve the quality of life of patients and their families when facing the problems associated with life threatening illnesses. The focus of this type of care is usually not to find a cure but to reduce disease symptoms severity and increase the quality of life of patients (WHO, 2008).

Palliative care is a comprehensive patient care delivery that incorporates many dimensions of medicine through interdisciplinary teams that cut across many healthcare domains (Kuziemy, Weber-Jahnke, Lau, & Downing, 2008). Patients are usually cared for from their home by different doctors, and nurses from different clinics and hospitals. It also involves social workers, paramedics, etc that need to share accurate information on patient care in order to be able to save life on emergencies. Patient care could also involve automatic monitoring devices and sensors.

The IT requirements for Palliative care can be quite complex and dynamic, involving multiple heterogeneous domains and changing patient circumstances. Therefore, determining the appropriate data sharing infrastructure is a big challenge because of its multi-dimensional application.

4.2 Palliative Severe Pain Management Scenario

According to (Kuziemy, Weber-Jahnke, Lau, & Downing, 2008), severe pain is a pain score of 8 and above on a 10-point numeric rating scale.

As shown in Figure 33, we describe our scenario as follows:

Nancy Smith is a patient admitted into Palliative Care, giving her the chance to receive medical care from home. She is able to check and post periodic Pain Scores using her home PC, PDA or through automatic sensors. A patient assigned nurse is able to automatically cumulate and analyse these pain scores to determine how the pain medications are performing on the patient. If the Pain reports indicate severe pain above the acceptable threshold, the nurse sends a Pain alert to the patient physician for action. The patient physician can pull historical reports, perform analysis and determine if the patient needs a new prescription. If required, the physician sends a new pain relief prescription back to the nurse who then administers the new prescription on the patient. The entire interaction is captured by the Palliative Care Portal (PCP) for reporting, non-repudiation and legal compliance.

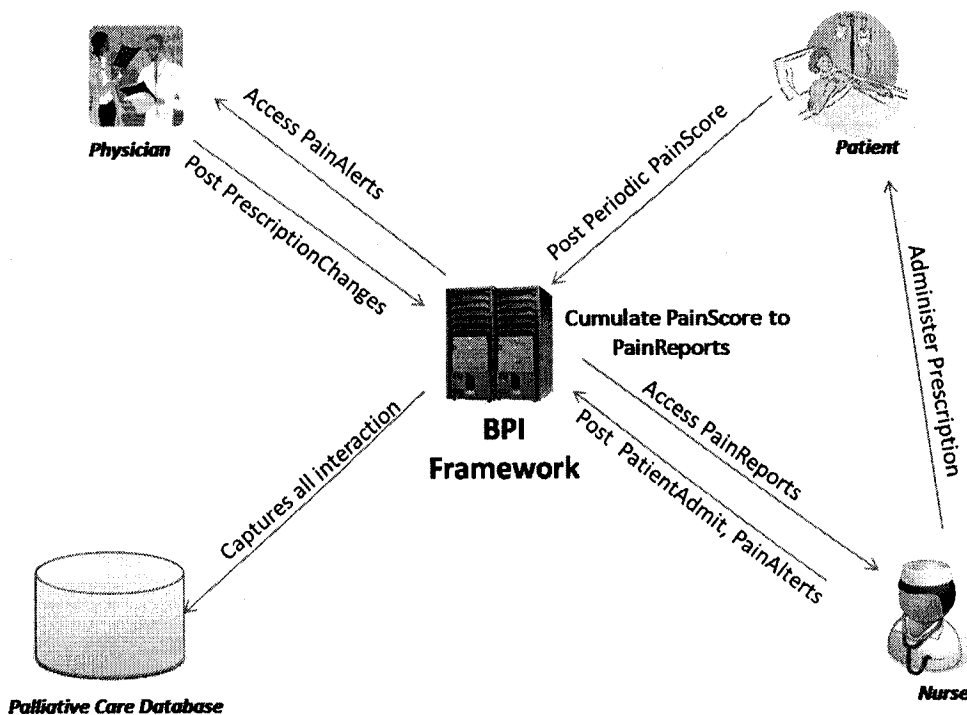


Figure 33. Palliative SPM Scenario

Palliative SPM Scenario requires an appropriate BPI framework for coordinating the data-sharing between the patient, nurse, doctor and the database infrastructure. Sections below describe alternative architectures for this scenario.

4.3 Web Application Portal

4.3.1 Web Application Portal Architecture

(Kuziemy, Weber-Jahnke, Lau, & Downing, 2008) proposed a portal-based Information System for the Ontario LHIN which we analyzed for our scenario. Figure 34 shows a high-level architecture of this implementation. The Web application portal connects the clinical management, patient care, and severe pain management applications as well as a number data sources into one application portal. Patients are admitted into Palliative Care when the nurse creates a profile for them on the Palliative Care Portal. The patient is setup at home to use her home PC to periodically enter her *PainScore* through the Patient interface. The nurse periodically runs *PainReports* on each of the patients to analyze and review the performance of their prescriptions.

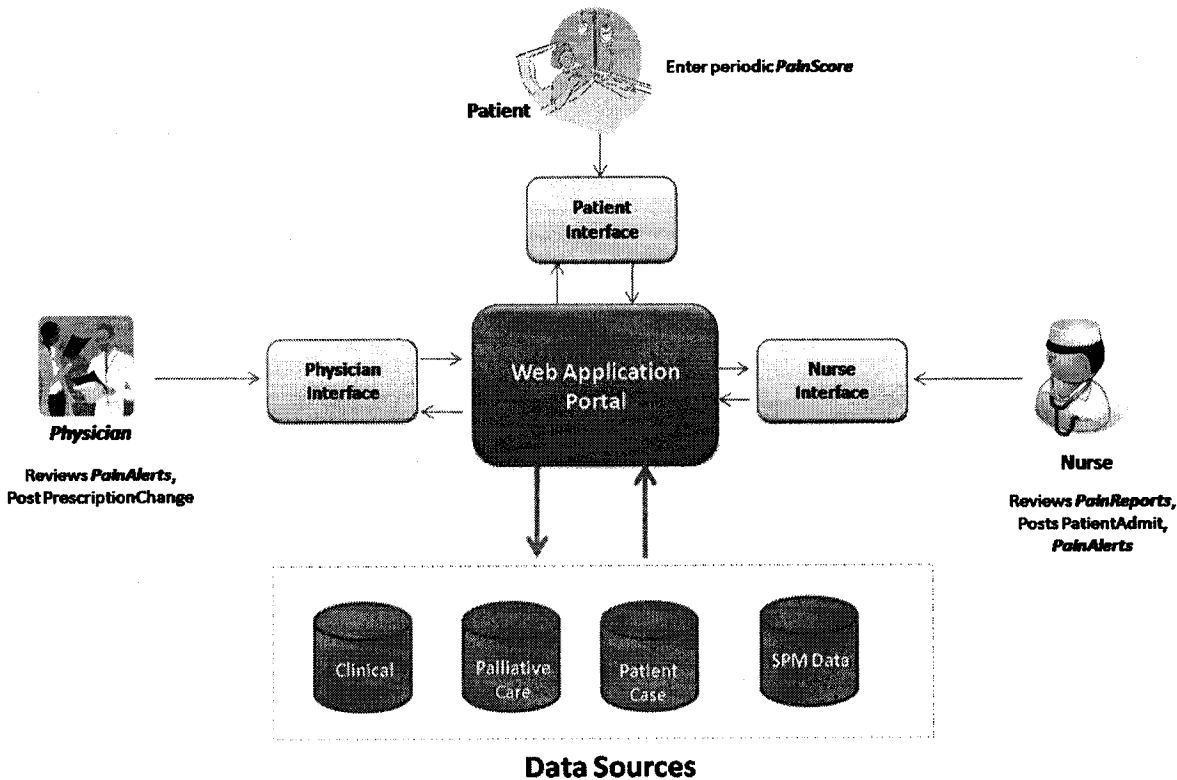


Figure 34. Web Application Portal Architecture

If a patient *PainReport* shows severe pain, the nurse promptly creates a new *PainAlert* and sends it to the listed patient physician for prescription changes and intervention. The on call physician uses the physician interface to check for new tasks. For every new *PainAlert*, the physician prepares a new prescription which gets sent back to the nurse and then administered on the patient. In this implementation, the Web application portal interacts with four data sources. The Clinical database, the Palliative Care database, a Patient Care database and the severe pain management (SPM) database used specially to capture the patient pain scores.

4.3.2 Analysis

The functional requirements for the scenario were met but with constraints to true BPI. These shortcomings are described as follows:

- All parties: patients, nurses, and physicians are required to access and use the Web application portal in order to participate in Palliative Care. The implementation does not

represent a true B2B data sharing since it provides no mechanism for automatically integrating external sources of data like existing other hospital systems.

- The application is web-based using HTTP. It is therefore accessible to customers via the Internet. Authentication and Sign-on is done centrally through the Web application portal. However, it still doesn't represent a true SOA since there is no service abstraction to the framework. It also does not leverage SOA technologies like SOAP, WSDL and UDDI.
- *PainScores* have to be keyed-in by the patient through the Web interface from a web browser or through a PDA-browser. This framework has no support for published data. For instance, nurses do need to periodically "pull" *PainReports* on each of the patients. Event-based activities are not logged, so it is difficult to review trends during emergencies and intervention.
- The implementation is not data-driven but interface-driven. Interfaces define data and they are not easily modified or new data types defined without changing the design of the entire Web application portal.
- Data sources are statically linked and strongly coupled. Rules of exchanges and message transformation are defined in low-level business rules embedded in views, stored procedures, methods and objects that define the application logic. These are usually not dynamic enough to capture new requirements dynamically.
- It presents a monolithic framework: static processes, interfaces, roles, and permissions.

4.4 Integrated BPEL CoT

4.4.1 Integrated BPEL CoT Architecture

The Integrated BPEL CoT framework was introduced in Section 2.7 as a framework for secure sharing of consumer identities and data using SOA and access control technologies. It also extends WS-BPEL to support collaborating organizations in a CoT. This framework, built by (Doshi, 2008) was used for our scenario.

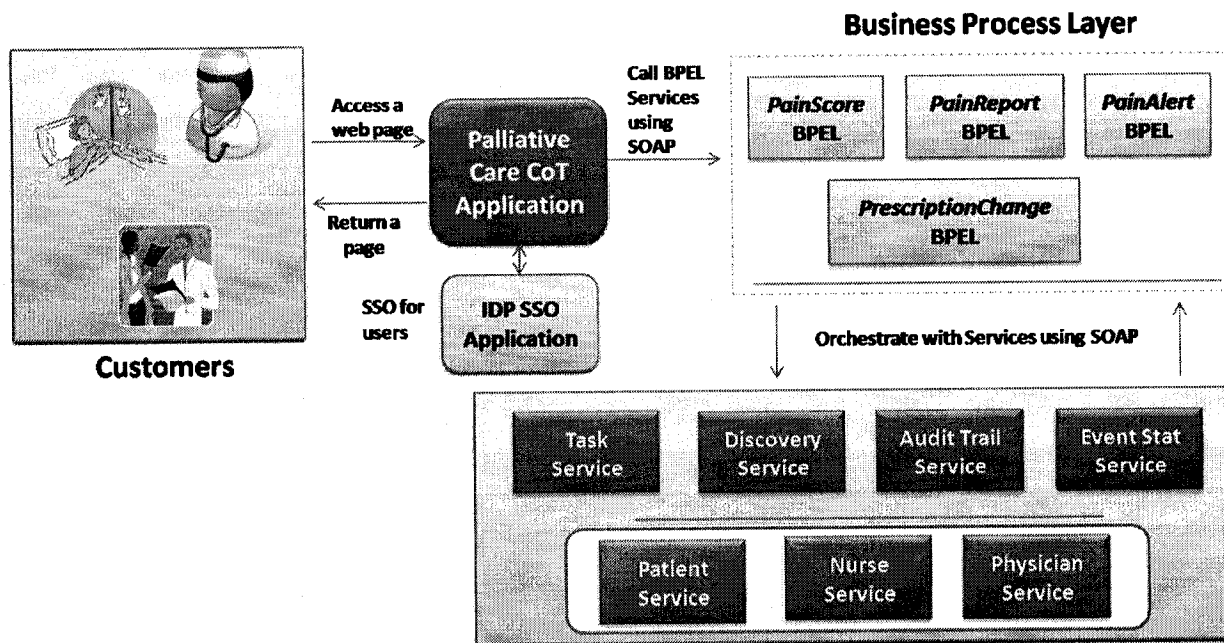


Figure 35. Integrated BPEL CoT Architecture

Figure 35 presents a high-level overview of the Integrated BPEL CoT architecture for this scenario. In this framework, the Web Application Portal is extended to invoke BPEL processes that define scripted functionalities. The BPEL processes in turn use the frameworks functionalities for service discovery, audit trail, event logging and human interactions.

The functionalities required by the patient, nurse and the physician are abstracted in a Patient Web service, a Nurse Web service and a Physician Web service respectively. Each of these Web services also implements Liberty SOAP binding.

The Identity Provider (IDP) assigns pseudonymous identifiers to the customers: the patient, nurse and doctor. A customer logs on with his credential and is authenticated by the IDP using the Single Sign-On (SSO) application. After a successful authentication, the customer is then redirected back to the portal to access available services. As part of the authentication, a bootstrap EPR is obtained for interacting with the representative Web service for the customer.

Four BPEL processes are defined to cater for *PainScore* logging, generating *PainReports*, *PainAlerts*, and for interacting with the physician for *PrescriptionChanges*. In using the Integrated BPEL CoT for our scenario, we will point out the roles of the services used within the

BPEL scripts to provide additional CoT functionalities. The role of the *Event Stat Service* is to log and aggregate interaction data. For simplicity, we will refer to service calls like the Event Stat service invocation to log the *PainScore* for a patient whose pseudonymous identifier is Patient_S1, as *LogPainScore* ('Patient_E1'). Using the same approach a call to retrieve the *PainReport* for this patient is *getPainReport* ('Patient_E1').

The Discovery Service enables the discovery of services and customers using pseudonyms. It also discovers the binding bootstrap EPR for participating CoT customer services. For example, our patient whose identifier is 'Patient_E1' with the Event Stat service uses a totally different identifier with the Audit Trail service. By the nature of this architecture, this service is invoked to translate pseudonyms, and retrieve bootstrap EPRs before every service invocation or during data preparation prior to service calls. These steps clutter the business logic represented in the BPEL scripts. All the process defined below will show the first call to the Discovery Service but will skip the rest of the calls for clarity.

Finally we use the *Audit Trail Service* to log identifier sharing activities. We have also avoided showing a lot of its interactions for clarity. The *Task Service* manages steps that require user interaction while the Task Portal provides the interaction interface for updating the status of task requests.

4.4.2 PainScore BPEL

Figure 36 illustrates the *PainScore* BPEL process. The Patient Service uses the patient pseudonym to send the *PainScore* which is picked up through the <Receive> activity, therefore kicking of the process.

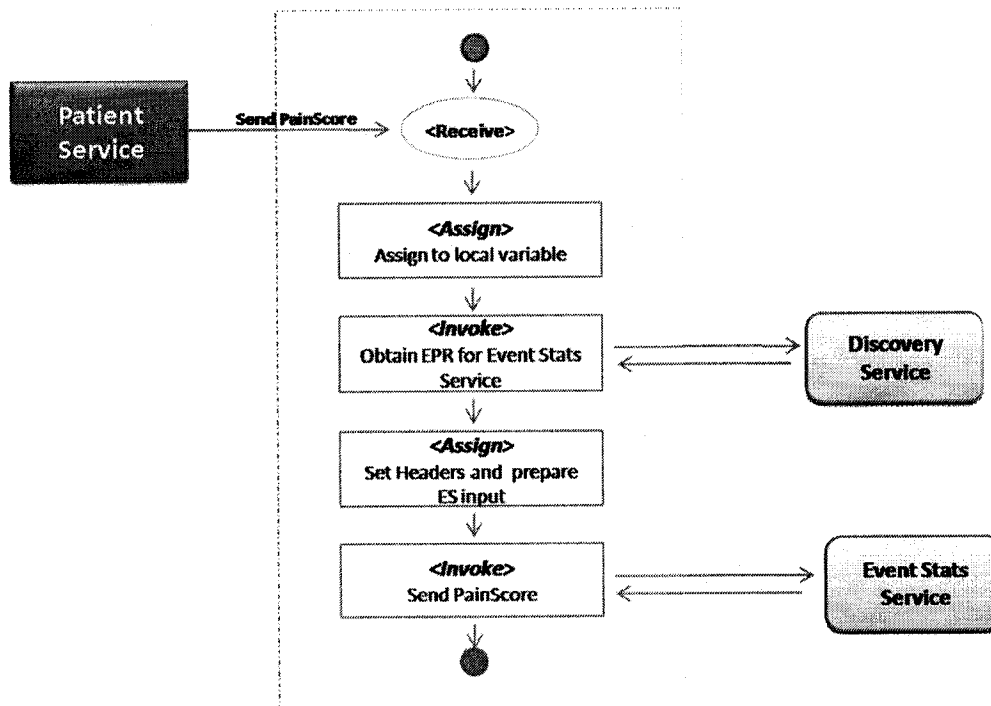


Figure 36. *PainScore* BPEL

The *PainScore* is saved into a local variable using the *<Assign>* activity. The DS is then invoked to obtain the *EPR* for the *Event Stats* service and also retrieve the Event Stats identifier for the patient. *<Assign>* is used to copy that into the SOAP headers before making an *<Invoke>* call to the *Event Stat* service to log the pain Score event.

4.4.3 PainReport BPEL

As shown in Figure 37, the steps for preparing the *PainReport* include:

- a. *<Receive>*: The Nurse Service sends a request for Patient *PainReport*. This request is forwarded by the CoT Portal to the PainReport BPEL. The *<Receive>* activity on receiving this message kicks off a new *PainReport* business process.
- b. *<Assign>*: The request is saved into a local variable.
- c. *<Invoke>*: The Discovery Service is invoked to retrieve the Event Stat service EPR and corresponding identifier for the patient.
- d. *<Assign>*: The SOAP headers for the Event Stat service are set using the received data.

- e. <Invoke>: The Event Stat service *getPaintReport* is the invoked to receive patient pain report.
- f. <Assign>: The retrieved data structure representing the *PainReport* is saved locally.
- g. <Reply>: Finally, the Nurse service receives the *PainReport*.

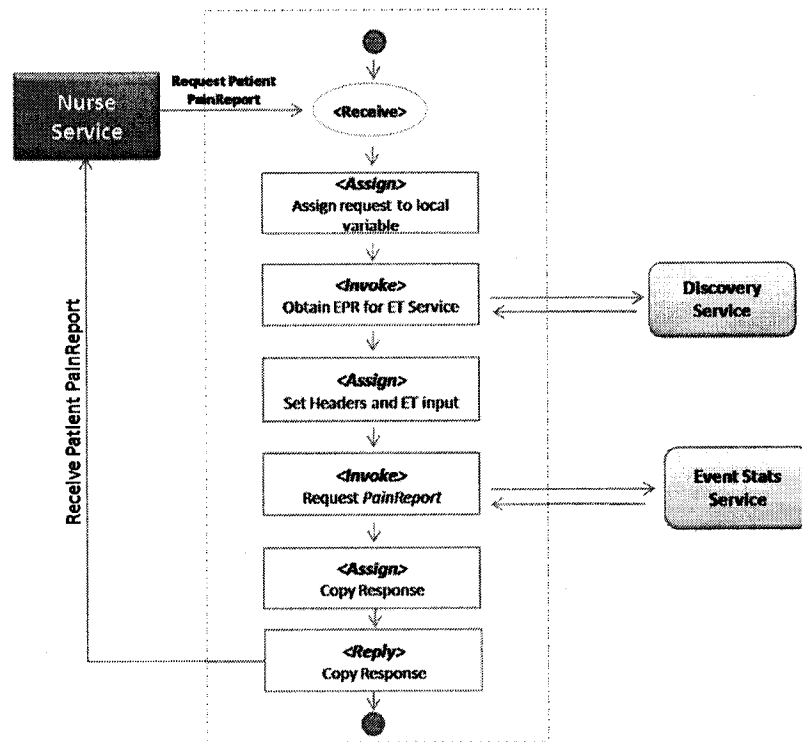


Figure 37. PainReport BPEL

4.4.4 PainAlert BPEL

Figure 38 illustrates the steps taken by the Nurse to send the *PainAlert*. These steps are described as follows:

- a. <Receive>: The Nurse Service sends off a *PainAlert* to the patient doctor. The CoT uses this message to kick-off the *PainAlert BPEL Process*.
- b. <Assign>/<Invoke>: The *PainAlert* is saved locally while the Discovery Service is invoked to discover identifiers and retrieve the bootstrap EPR for the Event Stat service and the physician service.

- c. *<Assign>*: SOAP Headers are prepared for calling these services.

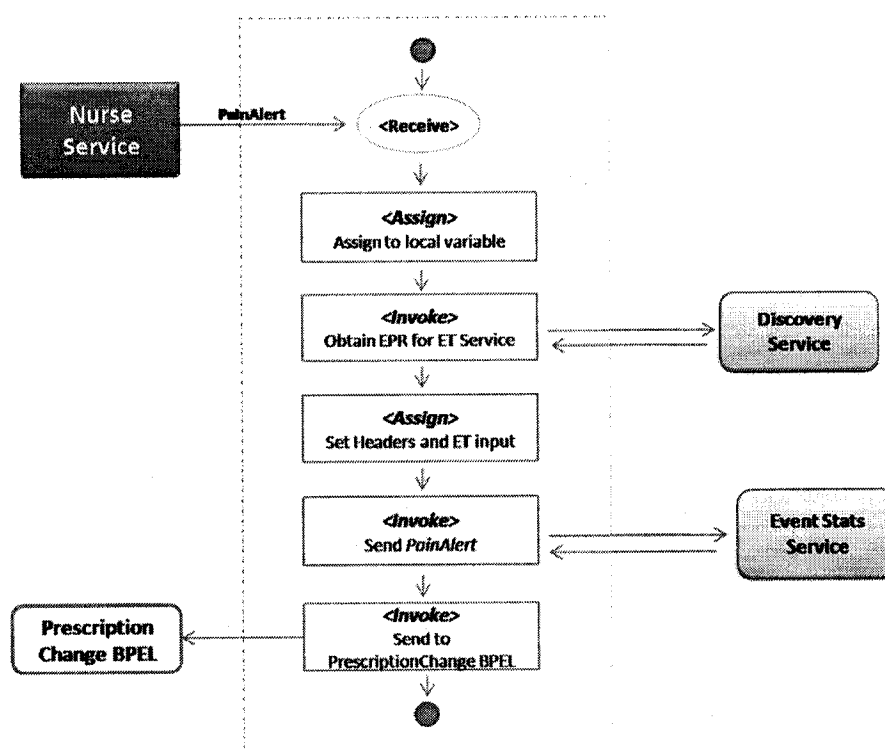


Figure 38. PainAlert BPEL

- d. *<Invoke>*: The Event Stat Service is then invoked to capture the *PainAlert*. Subsequently, the *PresecriptionChange* BPEL process is also invoked to receive the *PainAlert* and interact with the patient physician for intervention.

4.4.5 PrescriptionChange BPEL Process

Figure 39 illustrates the processes involved in sending a *PainAlert* to the patient physician using the Task Service to interact with the physician and the nurse in deciding and administering a prescription change. The first three steps are the same as were defined in the previous processes, Event Stat service is invoked after retrieving the identities and bootstrap EPR from the Discovery service.

The rest of the steps are described as follows:

- a. *<invoke>*: The Task Service *setTask* is invoked with a *taskType* defined as “PainAlert” and *taskData* being the actual *PainAlert* data from the PainAlert BPEL.

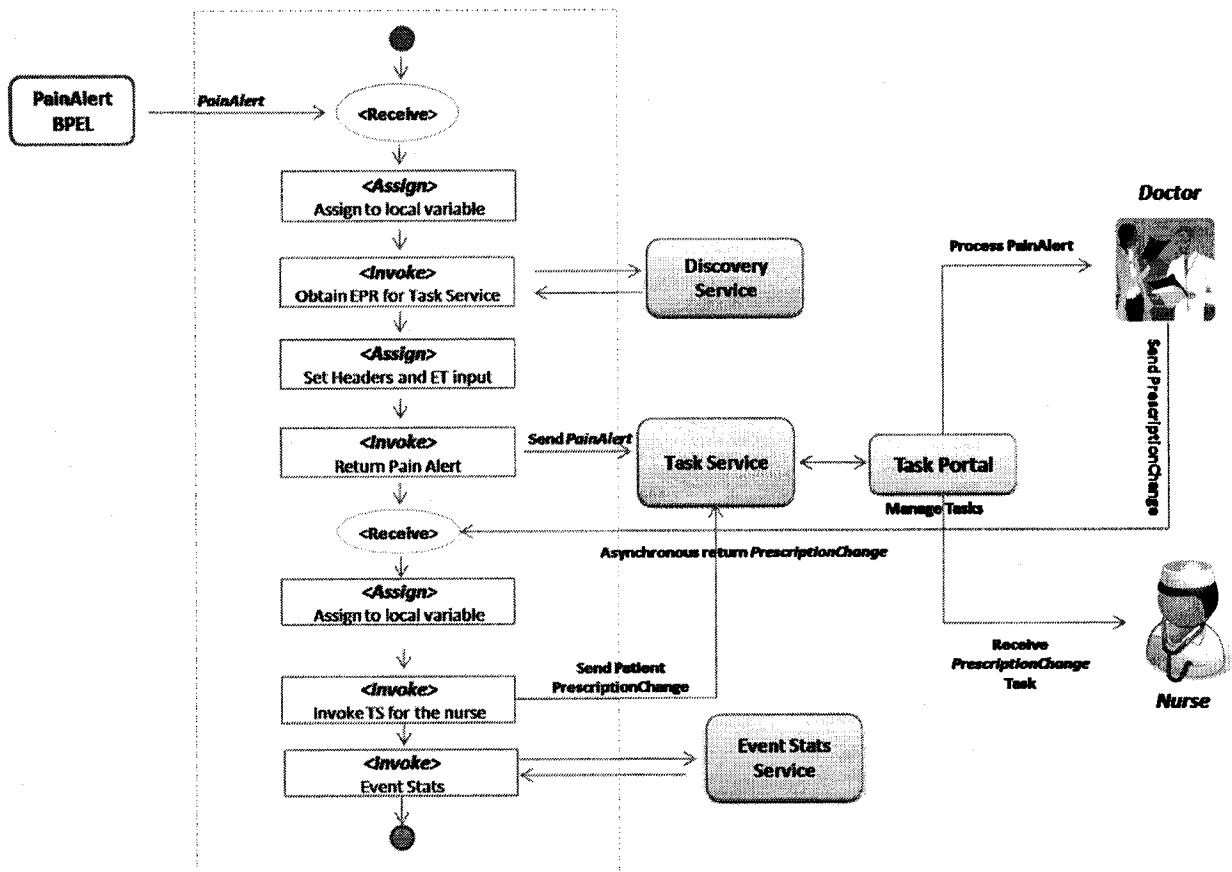


Figure 39. PrescriptionChange BPEL

- b. *Task Portal*: The Task Service then forwards this request to the Task Portal identifying the physician. The physician is then notified of the PainAlert. After acting on it, the doctor sends back a *PrescriptionChange*.
- c. *Asynchronous Callback*: The Task Portal then uses the identifier for the physician to send back the *prescriptionChange* data to the BPEL process. The BPEL *<Receive>* activity uses this variable to correlate the running process and continues execution.
- d. *<Assign>/<Invoke>*: The *prescriptionChange* is copied to a local variable, the Discovery service is used to identify the patient's nurse service. The Task Service is invoked once again to forward the *prescriptionChange* to the nurse as a new task.
- e. *<Invoke>*: Finally the Event Stat service is invoked to log this event while the *Audit Trail* service is invoked to log the activity.

4.4.6 Analysis

This approach combines a Web Application Portal with Liberty Alliance CoT Identity management while using BPEL to describe data-driven BPI for B2B. It shows the following unique characteristics:

- Integrated CoT BPEL is a true SOA B2B. BPI uses SOAP for messaging and WSDL for defining the interfaces. The Discovery service can be adapted to support UDDI. It supports federated identity management through Single Sign-On, and the privacy of customers with service providers through pseudonymous identifiers. Task service provides a framework for incorporating human interaction.
- Each participating customer business process needs to be scripted into separate BPEL processes like *PainScore* BPEL, *PainAlert* BPEL, and *PrescriptionChange* BPEL. This could be complex to integrate. For each BPEL service, there is no dynamic means of adding support for the new data structure. The Event Stat service needs to provide interfaces for each type of event and a means of saving data for the event. There is also no mechanism for automatically setting up tasks, controlling how they are approved and linked asynchronously to running BPEL processes.
- Integrated CoT BPEL assumes that the discovery service provides a bootstrap EPR that is set on the SOAP Headers to dynamically bind to services. Currently this level of binding abstraction is not supported by BPEL. Therefore each of the called service must be linked up with the BPEL process as *<PartnerLinks>*.

4.5 Simple SOA Publish/Subscribe

4.5.1 Simple SOA Publish/Subscribe Architecture

Our Simple SOA Publish/Subscribe framework approaches BPI by describing interactions as a network of events. Each event results in some data being communicated from an event source to an event destination. In this architecture, the Palliative SPM BPI scenario is broken down into a set of Topics representing data exchanges. Figure 40 shows the architectural representation of this interaction. All communication amongst the partners uses SOAP as the transport protocol.

This is realized by implementing a Web service message broker with a publisher component that implements its WSDL. We also implemented a subscriber component, which is a Web service that implements a *notify ()* interface. The message broker also uses a special publishing component to implement the WSDL of the subscriber component.

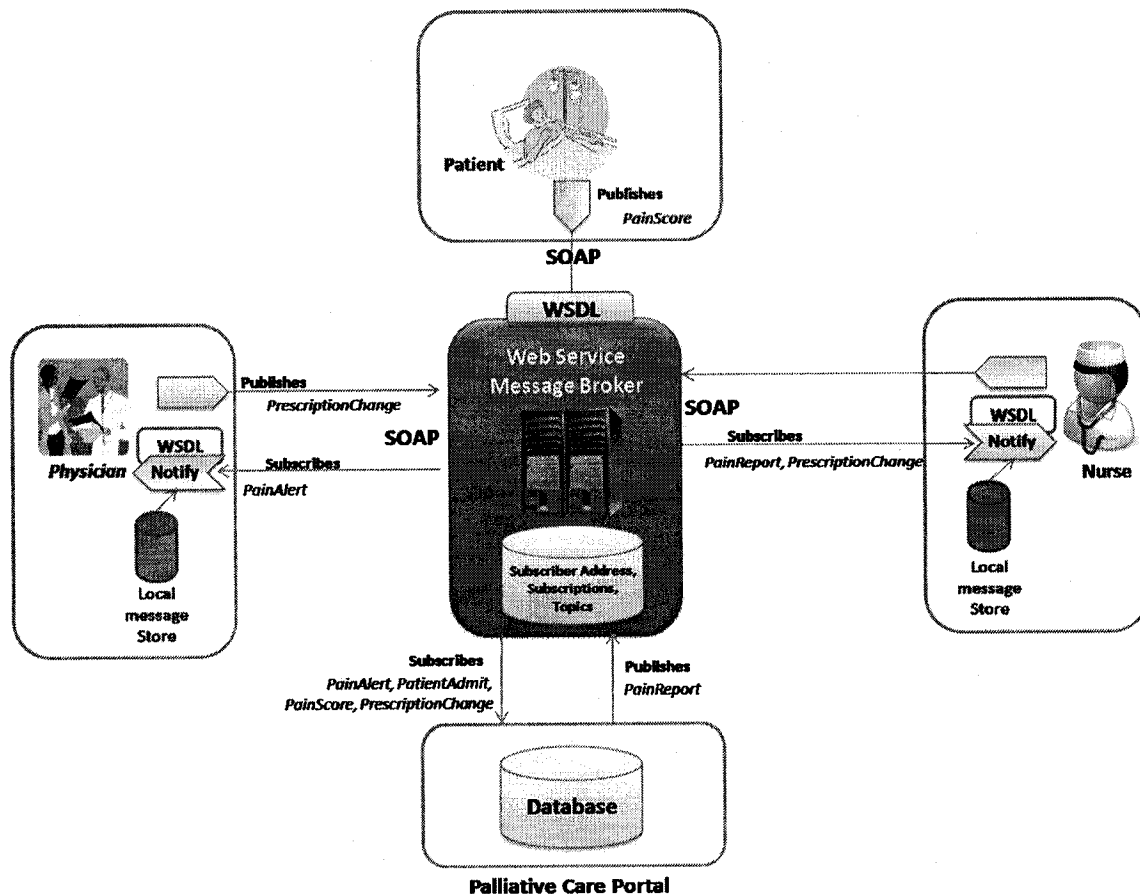


Figure 40. SOA Publish/Subscribe Architecture

As shown in Figure 40, each publishing partner must incorporate the publisher component to its application while those that need to subscribe to the broker's messages must implement the subscriber component. Partners that play the role of both publisher and subscriber need to implement both interfaces. The message broker plays the role of a message router. It receives all communication and routes them to the destination subscribers. In addition, the message broker maintains a list of all participating partners through a registration process for both publishers and subscribers. It also keeps a list of topic subscribers' topic subscriptions. Table 4 summarizes the roles played by the Partners in this scenario. The interaction is similar to the one described in

section 4.2, except that the data producers need to publish data to the broker while the data consumers are required to have registered and active subscriptions to the topic with the message broker.

Table 4. Palliative SPM Management in a Publish/Subscribe Topology

Node/Service	Role	Topic of messages published	Message Topic Subscriptions
<i>Broker</i>	Message Broker	None	None
<i>Nurse</i>	Publisher/Subscriber	PatientAdmit PainAlert	PainReport PrescriptionChange
<i>Patient</i>	Publisher	PainScore	Non
<i>Physician</i>	Publisher/Subscriber	PrescriptionChange	PainAlert
<i>Palliative Care Portal</i>	Publisher/Subscriber	PainReport	PatientAdmit PainAlert PainScore PrescriptionChange

For example the nurse plays the role of a publisher of *PatientAdmit* and *PainAlert* data, as well as a subscriber to *PainReport* and *PrescriptionChange*. Since the topics are pre-defined, the Palliative Care Portal acts as a surveillance infrastructure and subscribes statically to all the topics. This way, it is able to receive all the messages sent on these topics. We have assumed that messages are simply data blobs. As data blobs, we have no support for content-filtering.

4.5.2 Registration

Publisher registration is not defined. This is based on the assumption that there are many publishers of data. Subscribers are identified by their Web service endpoint address, the Subscriber EPR. To register with the broker, the subscriber sends its identification, addressing and service EPR data to the *registerSubscriber* interface. The broker validates this subscriber and returns a token that is used by the subscriber to sign subsequent interactions. For every message exchange, the broker renews this token to ensure that the subscriber's credentials are never compromised.

Figure 41 shows the SOAP exchanges for *NurseBetty*'s service registration with the broker. The response to a registration is a security token.

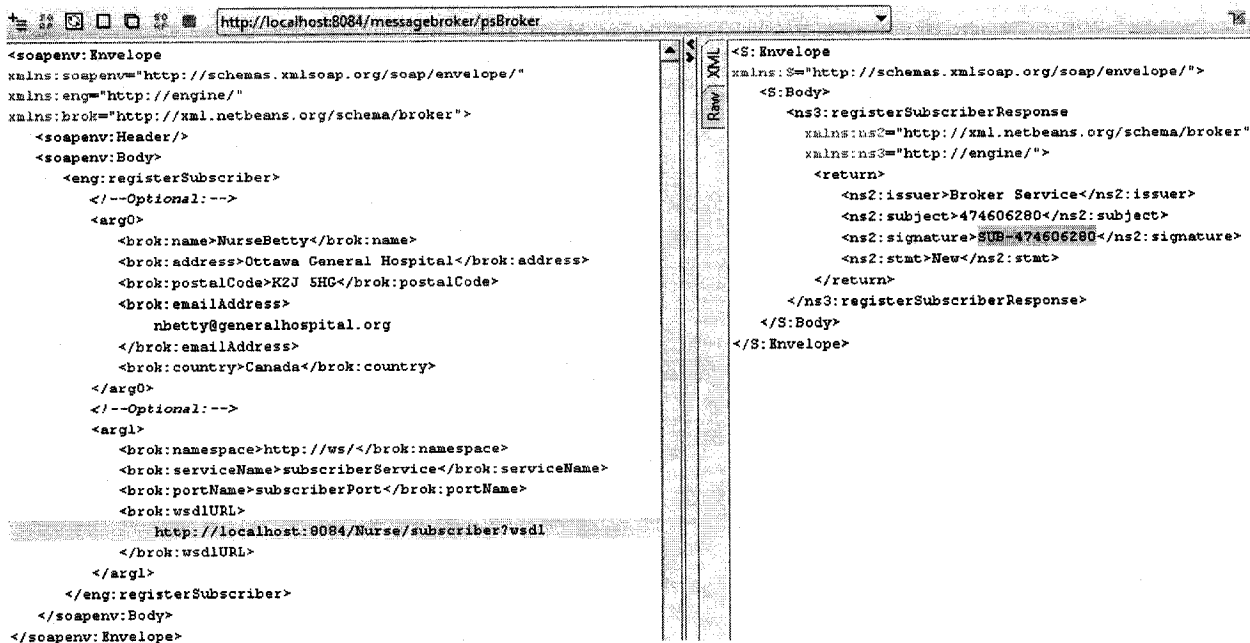


Figure 41. SOA Message Exchange for Partner Registration

4.5.3 Defining Data

In SOA publish/subscribe the responsibility of defining and categorizing data is left to the publishing application. The publisher calls the message broker's *<advertiseTopic>* interface specifying the name of the topic. This is a one-way asynchronous call. The message broker simply updates its topic registry with the new topic.

4.5.4 Subscription

Subscription is simply an indication of interest to a particular type of message identified by its topic. During a subscription, the subscriber sends a *subscribeRequest* to the message broker's *subscribe* interface. This request contains three arguments: the security token of the subscriber, the subscriber's endpoint address and the topic of the subscription. Figure 42 shows the nurse's subscription to *PainReport*. Every valid subscription is acknowledged by a renewed security token for the subscriber.

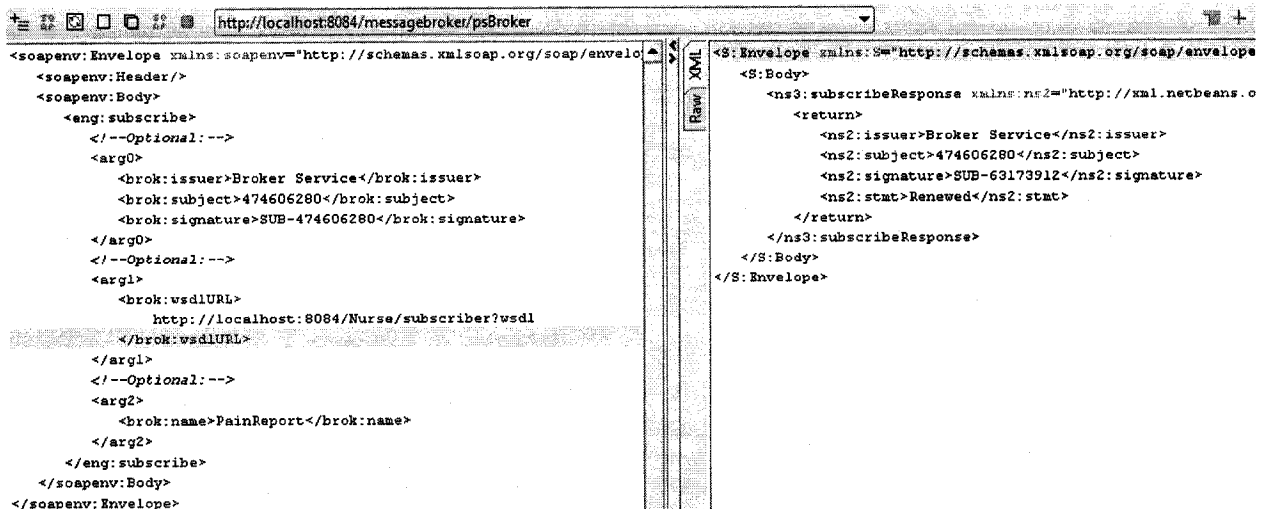


Figure 42. Subscription to PainReport

All subscriptions must be sent by the subscriber. There is no mechanism for dynamically notifying subscribers of new topics. For instance, the surveillance database must explicitly seek the list of topics using the *getSubscriptionTopic* operation and subsequently subscribe to the new topics. Subscribers can unsubscribe from a topic through a similar process but calling the *unsubscribe* interface.

4.5.5 Receiving and routing Messages

After the initial setup described above, a subscriber is then ready to receive message notifications from the message broker. Publishers prepare messages, then call the message broker's *notify* interface with a data structure containing the message topic and the actual content representing the payload.

```

<eng:notify>
  <!--Optional:-->
  <arg0>
    <brok:topics>
      <!--1 or more repetitions:-->
      <brok:Topic>
        <brok:name>PainReport</brok:name>
      </brok:Topic>
    </brok:topics>
    <brok:payload>
      <PainReport>
        <PreparedBy>PCP Portal</PreparedBy>
        <Patient>ABC-134554</Patient>
        <Period>29/11/2008:10:00-11:00</Period>
        <MaxPainScore>9</MaxPainScore>
        <MinPainScore>7</MinPainScore>
        <AveragePainScore>8.3</AveragePainScore>
      </PainReport>
    </brok:payload>
  </arg0>
</eng:notify>

```

Figure 43. Message Structure: *PainReport*

On receipt of the message, the broker reviews the subscription registry for subscriptions matching the message topic. For each of the subscribers, it retrieves the endpoint address, and automatically binds to the subscriber Web service to deliver the message payload through the subscriber *notifySubscriber* interface.

4.5.6 Analysis

The simple SOA publish/subscribe implementation described in this section supports our Palliative SPM scenario by decomposing complex BPI problems into multiple atomic events with multiple interactions using publish/subscribe. However, it demonstrates the following unique characteristics:

- Unlike the Web Application portal and Integrated BPEL CoT, the BPI definition is not statically tied to a B2B. Rather they are dynamically defined as run-time attributes.
- All partners implement the same communication interface therefore making it possible to use dynamic binding of endpoints at runtime.

- Asynchronous web service calls took place for all message distribution and topic advertisement. All components were decoupled in time, space and synchronization. Collaboration rules were defined at run-time through subscriptions.
- We introduced a trust model by using security tokens for authenticating message subscribers. This way messages only get forwarded to secure subscribers. This trust model can also be successfully extended to the publishers.

Some of the limitations of this framework include:

- Topic advertisement from arbitrary publishers is not a secure way of decomposing processes.
- Publishers need to be identified like we do subscribers. There is need for a better mechanism of describing registration, setting up publishing rights around topics and uniquely identifying subscriptions.
- Subscribers are identified primarily by the endpoint address. The effect is that if the subscriber is redeployed to a different address space, it is not able to use its subscriptions as they are tied to the endpoint address.
- Except for *notify()* which is asynchronous, the rest of the broker's management operations are request/response strongly coupled calls.
- It utilizes publish/subscribe broadcast type interaction. There is no mechanism for defining more flexible subscriptions that filters message contents for specific patterns, support message transformation and describe how and when the notification should be delivered.

4.6 Policy-based SOA Publish/Subscribe

4.6.1 Policy-based SOA Publish/Subscribe Architecture

Figure 44 presents a high-level overview of our architecture. This architecture is functionally similar to the simple SOA publish/subscribe described in the previous section but with additional

components for a dynamic BPI using a policy model. Our architectural framework automates the process described in section 4.5 to support those BPI requirements described in section 3.1.

In this architecture, we use a partner service (PS) component as an adapter interface between the policy-based message broker and the application or process that publishes or subscribes to data. For example, a patient “John Smith” using a monitoring device attached to his wrist sends a *painscore* periodically. This device communicates locally and drops this *painscore* data structure into the *message outbox* of the partner service component. The Patient’s PS component is hosted on the patient home computer or a specially issued PDA from the hospital. It is pre-configured to dynamically bind and publish this data to the policy-based message broker. In other instances, like that of “NurseBetty” that subscribes to *PainReport* and *PrescriptionChange*, it stores these subscriptions and correlates them to outbound and in-bound messages. This way, it is able to coordinate data for multiple applications.

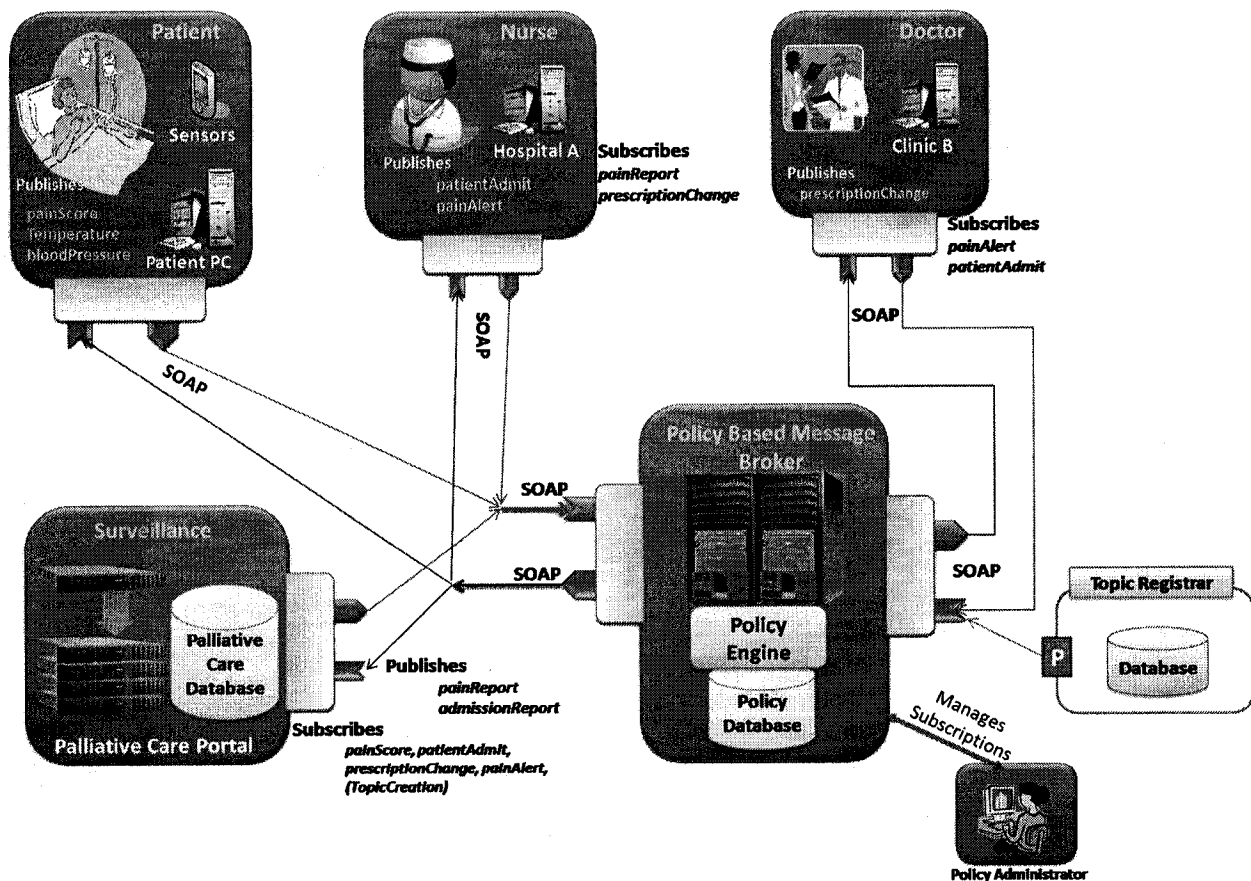


Figure 44. Policy-based SOA Publish/subscribe Architecture

Unlike the implementation using SOA publish/subscribe; topics are defined by special publishers, the *Topic Registrars*. Each topic contains an attribute set that describes the messages published on the topic. The policy-based message broker is the core middleware. It manages interactions with all the customers PS components on behalf of collaborating applications through a Policy engine and the policy database. A policy administrator uses a special interface to deploy subscription policies for collaborating partners. A special surveillance infrastructure hosted by the Palliative Care Portal subscribes to the policy message broker internal topic creation event and is able to receive the schema definition for each new topic. It subsequently uses this schema to create automatic table representation for capturing the event data. It then subscribes to messages attached to this topic. Unlike the SOA publish/subscribe, all these operations are performed automatically.

The entire infrastructure uses SOAP as the communication protocol. The policy-based message broker and all the interacting PS components are Web services endpoints.

4.6.2 Policy-based Subscriptions

To illustrate the flexibility offered by our framework in defining BPI, let us define a specific business need for data using a subscription policy.

“Nurse Betty is a Gerontological nurse working the hospice Palliative Care. She roams from location to location to cater for her patients. The patient “John Smith” is a cancer patient and requires constant monitoring of his pain medication. This patient with a PatientID: PSPM-120032 has just been assigned to Betty to the end of year. Like other Palliative patients, John publishes periodic pain scores. The surveillance infrastructure packages and publishes PainReport for all the Palliative SPM patients every hour. Nurse Betty subscribes to this pain report but require only those from her patients. Unfortunately, Nurse Betty infrastructure is only able to pick up messages during the hours that she is on the Internet. At other times, the messages need to be queued to be retrieved at the time her system is online again. She also needs to retrieve John’s patient records and prescriptions in order to familiarize herself with the specifics of his needs.”

Nurse Betty’s data requirement on John can be summarized as follows:

- a. Needs to have a subscription that will automatically force the policy-based message broker to archive message when delivery is not acknowledged.
- b. Requires a mechanism that allows Nurse Betty's patient management system to automatically pull batched pain reports on her patients.
- c. Requires a mechanism for allowing the nurse to request historical data on her patients.

Figure 45 illustrates a subscription policy for Nurse_Betty by the Policy Administrator. This policy takes care of requirements (a) defined above. The *<type>* attribute shows that the policy was defined through the Policy Administrator. In this scenario, this policy contains one rule. *<ApplicablePeriod>* define the time this rule would apply to incoming messages. The rule becomes invalid at the end of this period. The *<TopicFilter>* contain only *one <Topic>* since we are referring to *PainReport* alone. The *<contentFilter>* is an XPath query that filters the report for a patient record whose *PatientID* equals "PSPM-120032" representing the John Smith and whose *Provider* field contains *Nurse_Betty*.

When executed on incoming messages, a valid topic match followed by a valid content filter match will trigger the policy message broker to review the *<actions>* defined by the rule. The new message required is defined by the XPath expressions in the *<MessageSection>* for only the "Matched" filter results. No messages are defined for "NotMatched" filter result. In this case, the message that will be delivered will be a Node representing the *PainReport* entry for John Smith only. It is a subset of the original message from the surveillance infrastructure.

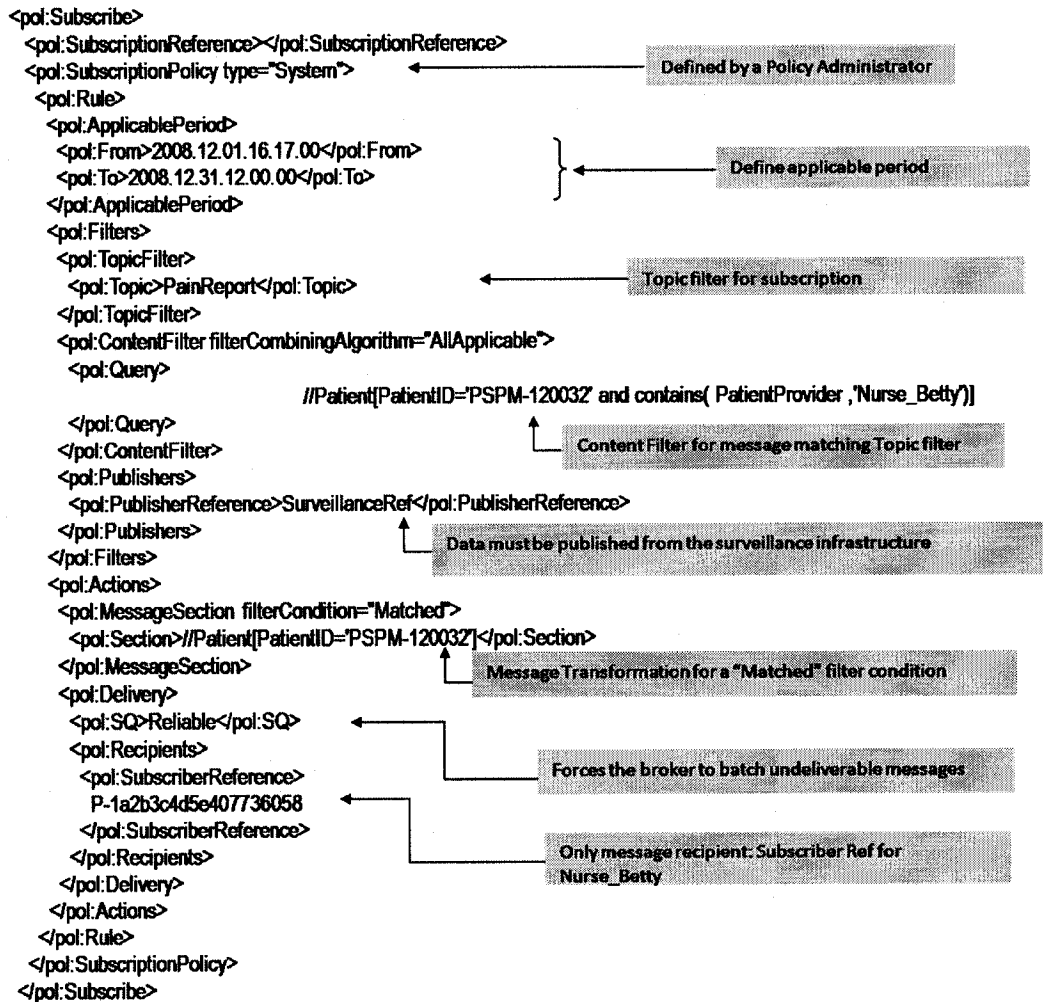


Figure 45. Policy-based Subscription

In defining delivery, `<SQ>` dictates a *Reliable* message delivery. In keeping with this condition, the message dispatcher for the policy-broker will handle the delivery as follows:

- a. Automatically bind and deliver the message to Nurse Betty's PS component through the *sNotify* interface. Unlike the "*BestEfforts*" Service Quality, the dispatcher will await a receipt acknowledgement. If the PS component returns an acknowledgement, then the message is marked as delivered and the operation is concluded.
- b. If on the other hand, there is a binding error to the remote endpoint or no acknowledgement is received, the dispatcher will activate a *pullpoint* for the component.

It then marshals the current message and subsequent message to the Pullpoint until normal delivery is restored.

- c. Finally, the <Recipient> element is set to the *PartnerReference* for Nurse Betty's PS component. The <Recipient> element is used to further restrict message delivery to specific recipients, in this case only Nurse Betty.

After the execution of the subscription, a response containing a unique *SubscriptionReference* is returned back the subscribed PS component for its own internal tracking with in-coming messages.

4.6.3 Retrieving Batched Messages

PS Components that go offline often need a mechanism for retrieving messages for subscriptions that were set with a Service Quality value of "Reliable" or "Pullpoint". In certain circumstances the PS component is able to send the *CreatePullpoint* message to the policy message broker before going offline. In our modified scenario, the second requirement is that the nurse is able to retrieve batch *PainReports* for her patients.

To support this, the PS Component sends the <getMessages> request with the subscription references as parameters and a buffer limit that dictates the maximum number of atomic messages that should be batched in each delivery. The broker then sends aggregated messages for the number of deliveries it takes to convey the entire pullpoint queue.

4.6.4 Requesting Stateful data through a DataQuery

We have discussed in section 3.3.9 that one of the weaknesses of publish/subscribe is the lack of a defined mechanism for accessing historical data. *DataQuery* message is used by our framework to overcome this limitation. The last requirement for Nurse Betty is to have a mechanism for requesting historical data on her patient. Figure 46 shows a SOAP message illustrating a typical report request from the nurse.

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:pol="policybroker.site.uottawa.ca">
  <soapenv:Header/>
  <soapenv:Body>
    <pol:sNotify type="DataQuery" token="S-1a2b3c4d5e2915898120">
      <pol:DataQuery>
        <pol:QueryReference>
          <pol:Description> getPatientRecord </pol:Description>
          <pol:Parameter>PSPM-120032</pol:Parameter>
        </pol:QueryReference>
        <pol:ProducerReference> SurveillanceRef </pol:ProducerReference>
        <pol:ConsumerReference>P-1a2b3c4d5e407736058</pol:ConsumerReference>
      </pol:DataQuery>
    </pol:sNotify>
  </soapenv:Body>
</soapenv:Envelope>

```

Figure 46. Data Query SOAP Message

This is based on the assumption that each station knows the name of the report being requested and the parameters to go with each of the reports. In this case, the name of the report is *getPatientRecord*, and the only parameter is the *PatientID*. On receiving this request, the policy broker performs a general request authentication, checks the *<ProducerReference>* to know the Partner Service that handles this data request. This allows us to define various report sources. In this particular case, the Surveillance infrastructure handles the request. The policy message broker subsequently sends the report request to the Surveillance infrastructure. The report is then prepared and republished to the broker with a message policy *<recipient>* defined as originator of the request. The policy forces the policy message broker to send the message only to the request originator.

4.6.5 Support for Multiple Groups and Roles

To illustrate our framework's support for multiple working groups, roles, applications in the same domain space, let us describe this scenario:

"Ottawa General Hospital uses a group of nurses and physicians to support Palliative SPM. The hospital management system supports shifts amongst the different groups in support of the same or different roles. Therefore all the Palliative care physicians should be able to change patient prescription whenever there is a PainAlert. At the same, the nurses should be able admit new

patients as well as receive and act on the PainReports for the patients under the care of the hospital. In spite of the different groups, the system requires that the Patient provider responsible for every action be stamped on all out-going messages to the Policy message broker.”

In implementing this scenario, the PS component uses multiple application message boxes for each of the roles. In this case, there are two message boxes, the first for Physician messages and another for the nurses’ messages. We also attached a special lookup database for the PS component to correlate in-coming and out-going messages to the message boxes, users, roles, topics and subscriptions.

Table 5. Correlation Lookup Tables

Users/Roles		Subscriptions/Roles		Topics/Roles	
User	Role	Subscription	Role	Topic	Role
James	Physician	PainReport_Sub	Nurse	PainAlert	Nurse
Betty	Nurse	PrescriptionChange_Sub	Nurse	PatientAdmin	Nurse
Dan	Nurse	PainAlert_Sub	Physician	PrescriptionChange	Physician
Carmen	Physician				
Amanda	Nurse				

Table 5 shows a representation of the lookup tables. Users are associated with specific roles. For each out-going application message to be published, PS Component adds to the *<MessageAttribute>* element, the username and the role of the user that produced the message.

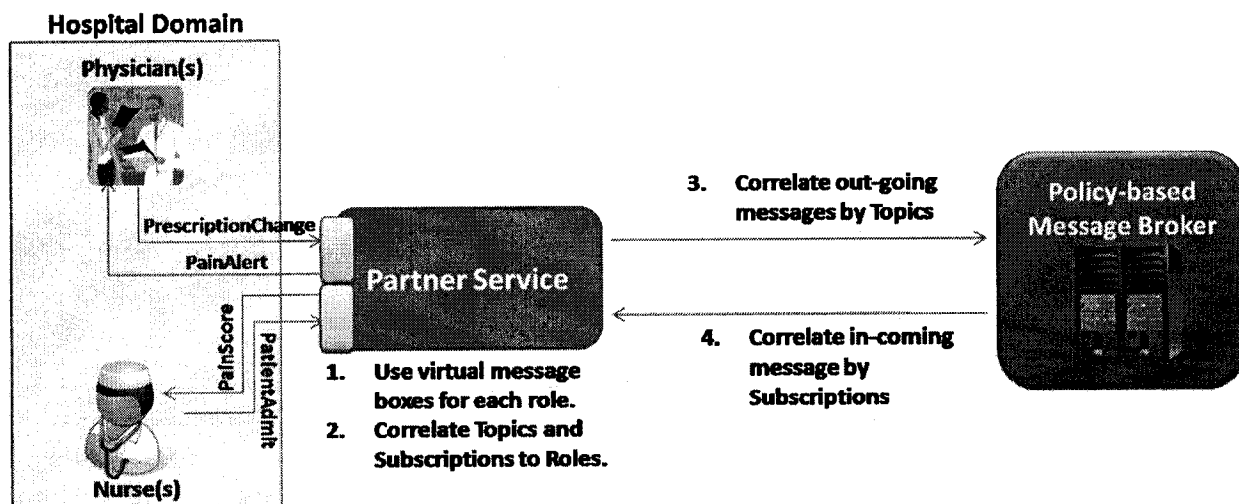


Figure 47. Supporting multiple groups and roles in a Partner Service Component

As illustrated in Figure 47, based on the type of message and the role of the user, the component automatically uses the lookup table to identify the topic before publishing the message. Incoming messages contain the *SubscriptionReference* of the subscription that created the message. These are associated with roles which results in their delivery to appropriate application message boxes.

4.6.6 Analysis

Data is dynamically defined at run-time using a combination of topic name and data attributes, thus providing an automatic connection between the management and logging of streaming data to reporting and historical information. Data topics are not advertised by any arbitrary publisher as it is with the simple SOA publish/subscribe. Rather, the Topic Registrar defines BPI event topics alongside policies that control the rights to publish on the topics. For example, by publishing the *PatientAdmit* topic creation event, we automatically have the surveillance infrastructure create logging table for the topic and subscribe to data from the topic publishers.

All communication and data exchanges on this framework are received by the policy-based message broker as a message using similar interfaces as the actual notification. Messages representing management operations are handled similarly like notification messages. Various handlers tie to the message context handler to process various types of message. This makes it very convenient to handle authentication, synchronous and asynchronous communication for all interactions.

The Partner Service components is a handy tool for integrating applications participating in SOA publish/subscribe. We have demonstrated its use as a local middleware for publishing, subscribing, receiving and correlating messages for applications and users.

Most importantly, our framework demonstrated through the scenarios described in this section, the flexibility of defining XACML-based subscription policies on a streaming data model. Policy-based subscriptions provide the means of defining interest to data through topic and content filtering, message transformation and delivery Service Quality. Data queries provide access to stateful as well as historical data and reports to customers. Message policies provide a means of performing directed point-to-point notification using a publish/subscribe infrastructure.

Chapter 5. Evaluation

The results of our case study scenario were used to evaluate this thesis along a number of dimensions. In line with the characteristics identified as driving force behind requirements in 3.1, the scenarios used in our evaluation are data-focused, dynamic, event-driven, and operate on reactive and heterogeneous environments not well served by process-driven B2B BPI approaches. We expect the same results in any other case study with those characteristics. However we cannot make any claim in situations where those characteristics do not hold.

In section 5.1 we summarize the candidate BPI frameworks used in evaluating our framework. In sections 5.2-5.5 we evaluate the benefits and flexibilities to BPI offered by our policy-based SOA publish/subscribe framework over a traditional Web Application Portal, an Integrated BPEL Liberty Alliance CoT, and the simple SOA Publish/Subscribe frameworks. In line with that, we evaluate each approach against the essential requirements for BPI in a B2B. Our approach will not present the same results for BPI processes that are solely work-flow driven, and not linked to B2B data sharing. In 5.6, we also compare the engineering effort associated with each framework. Finally, in Section 5.7, we evaluate the relationships between our framework and related WS-specifications and in section 5.8 we evaluate its relationship with XACML Role-based Access Control (RBAC).

5.1 Candidate BPI Frameworks

In this section, we introduce the BPI frameworks used in our scenarios. Each of these frameworks is described as it relates to BPI data sharing, SOA compliance and interface to users.

5.1.1 Web Application Portal

Web application portal is still the most popular means of integrating information sources in a single customer facing application. This approach uses custom connectors to integrate external sources of data with a central application that coordinates all the interactions and data exchanges

including user profiles, rights, permissions and activity logs. Web services can be utilized to abstract these information sources and present similar integration interfaces to the developers. Web application portals present a homogenous interface to users. Users simply access web pages through their browsers or handheld devices to enter data, search or query information and access reports. Low level business rules setup around the application components, as well as access rights and permissions setup between the web interfaces and database tables, control data sharing.

5.1.2 Integrated BPEL Liberty Alliance CoT

A CoT is the SOA framework described in section 2.7, which allows businesses to share data on their customers but giving the customers control over permissions and access to data being shared. It leverages SOA technologies: SOAP, WSDL, and UDDI. WS-BPEL is used to provide service orchestration abstraction to a CoT. An Identity Provider allows customers to use Single Sign-On to gain access to the CoT portal. It also uses the Discovery Service to coordinate the discovery and sharing of customer pseudonymous identifiers with CoT framework services. A Trusted Process provider manages BPEL automation of processes. In addition, it integrates an Audit Trail Service into a CoT to provide privacy compliance and uses an Event Stats service for performance monitoring.

5.1.3 Simple SOA Publish/Subscribe

The Simple SOA publish/subscribe was described in detail in section 3.2. It uses the W3C Web services stack comprising of SOAP, WSDL and an infrastructure that can support UDDI. WS-Addressing is used for service addressing and call-backs. It implements publish/subscribe interaction using Web services as the communication platform. SOA publish/subscribe provides BPI integration through event-driven interaction. Messaging is asynchronous and its focus is to interconnect application and processes, on the basis of the data they produce and share with others.

5.1.4 Policy-based SOA Publish/Subscribe

Our Policy-based SOA Publish/subscribe framework extends the simple SOA Publish/subscribe framework using declarative policies. Our goal is to achieve flexible data sharing and BPI in

B2B through a flexible communication pattern, message delivery and an information management infrastructure. A Policy Model based on XACML is used to achieve this declaratively. Central to this framework is a policy-based SOA publish/subscribe infrastructure that uses a Partner Service component to integrate information publishers and subscribers. It uses Topic Registrars to create “self describing” topics and a surveillance infrastructure for message logging. Like the simple SOA publish/subscribe, our framework leverages SOA technologies: SOAP, WSDL, and is adaptable to UDDI. Services are identified using WS-Addressing and the policy language is based on XACML. It is also adaptable to publish/subscribe SOA specification like WS-Notification and WS-Eventing.

5.2 Information Topic

Table 6 summarizes our policy-based SOA publish/subscribe against the other approaches, as it relates to Information Topics. By Information Topic, we are referring to how information is defined, categorized, discovered, aggregated and communicated in B2B. It helps us evaluate the frameworks’ abilities to meet the requirements for information discovery and categorization (Section 3.1.1) as well some elements of dynamic business configuration (Section 3.1.6). Each listed feature in the table is described in more details in the proceeding subsections.

Table 6. Comparison of BPI frameworks on Information Topic

Features	Web Portal Application	Integrated BPEL CoT	Simple SOA Publish/Subscribe	Policy-Based SOA Publish/Subscribe
Information Topic Definition	Design-time through <i>user Interfaces</i> , explicitly code methods and objects, database tables and views.	Design-time <i>XSD Complex types, Message Types and Variables</i> .	On the fly Topic definitions.	On the fly Topic but with attribute sets that describe the messages they represent (Self describing topics).
Active Information Discovery	Information is usually not discoverable	None. CoT Discovery Services provides a mechanism for the discovery of permissions to statically defined services.	Topic-based subscriptions.	Subscription policies
Information Selection	Searches and database queries	<i><query>, <if-else> BPEL scripts</i>	<i>Topic filters</i> . Some implementations	Build around <i>Topic and Content filters and filter</i>

Features	Web Portal Application	Integrated BPEL CoT	Simple SOA Publish/Subscribe	Policy-Based SOA Publish/Subscribe
		XPath expressions	support content filters using XPath queries.	<i>combining algorithms</i> using XPath queries.
Information Aggregation	Static Design-time connectors	Static Web service Orchestration done at design time.	Dynamic Service composition built through subscriptions.	Dynamic Subscription policies define complex relationships through rules.
Information Transformation	Yes (non-declarative) Embedded in the application logic	Yes <Assign>, <copy> XPath expressions	No	Yes Declaratively uses <MessageSection> constructs in subscription policies.

5.2.1 Information Topic Definition

B2B networks require a means of categorizing information. By Information Topic definition, we are referring to how information is defined and identified by collaborating processes. Web application portals define information through static customer interface forms and database tables. The interfaces are tied to explicit code methods and object references defined by developers and designers. The back-end systems use tables in the database to store the data captured by the interfaces.

In the Integrated BPEL Liberty Alliance CoT, information is defined through XSD complex types in the BPEL and Web services type definition files. Messages are defined by associating them to these base types. Interfaces in turn describe the input and output messages for the BPEL process and orchestrating Partner services. Internally, the BPEL process represents information as variables corresponding to the type definitions.

Publish/subscribe define information through *Topics*. SOA publish/subscribe follow a similar structure. *Topics* are categorization of messages that can be published or subscribed to. Unlike the web application portal and the Integrated CoT, topics are defined as on-the-fly runtime attributes. In our framework, topics are treated similarly but do provide more meta-data through message attributes that describe the type of information associated with streaming data/messages.

5.2.2 Information Discovery

An adhoc B2B network is mesh of information sources and sinks. Producers and consumers of information join and leave the network all the time. A flexible mechanism for Information discovery is an important requirement for BPI. In the web application portal, discovery is through search queries to the central database and other connected data sources. This type of discovery is based on stateful data usually meant for human interpretation. The Integrated BPEL CoT ties a discovery service to BPEL to enable the discovery and sharing of pseudonymous customer identifiers to support trusted data-sharing between collaborating services. Despite supporting a good mechanism for discovery of identities by collaborating services, it has no automated mechanism for information discovery. The simple SOA publish/subscribe discovers information through subscriptions to message *Topics*. Policy-based message broker uses policy rules to define a more flexible subscription model for discovering and manipulating stateless data streams. Stateful data discovery is defined through *DataQuery* requests that are sent to the surveillance infrastructure.

5.2.3 Information Selection

If we view a B2B network as a streaming database, then timely access to data is essential to running processes. As an important business configuration evaluation criteria, Information Selection is the mechanism for sieving through available data to select required information. Web application portal approaches this through low-level constructs defined by the developer. Integrated BPEL CoT relies on BPEL scripts `<query>`, `<if-else>` constructs for filtering data to retrieve needed patterns. These constructs use XPath expressions to match messages. SOA publish/subscribe utilize topic names as filters with an options for content-filters defined as XPath expressions. Our framework builds on this approach but uses *filter combining Algorithms* constructs to define various combinations of *topic and content* filters in a subscription.

5.2.4 Information Aggregation

Information aggregation refers to how B2B combine information from different sources. It is an important requirement for dynamic business configuration in BPI. For instance, CoT WS-BPEL communicate and receive messages through a series of `<invoke>` and `<receive>` operations. The joins are defined at design time through a BPEL designer. SOA publish/subscribe define

information composition on stateless data through subscriptions. Unlike WS-BPEL where joins become very complex as soon as you have more than three or four services, SOA publish/subscribe can define very complex interactions joins using subscriptions. By using subscription policies with multiple rules, our framework provides a dynamic means of aggregating streaming data.

5.2.5 Information Transformation

Information transformation refers to the mechanism for changing the format of messages for application specific purposes. Dynamic business configuration requires a framework with the capacity to automatically transform streaming data according to changing business needs and requirements. Web application portal defines static transformation for variables in methods embedded deep into the application business logic. Integrate CoT BPEL supports transformation only in BPEL processes by using `<assign>`, `<copy>` constructs to transform data using XPath expressions. The simple SOA publish/subscribe does not support information transformation since messages are delivered as a whole from publisher to subscribers. Declarative XPath Message transformation `<section>` defined in subscription policies is the basis for stateless information transformation in our framework. Our framework uses `<MessageSection>` constructs to define a set of XPath expressions in subscription policies that transform messages based on a *Matched* or *NotMatched* filter conditions. It is a good tool for maintaining privacy when the transmitted data is an aggregated report or query.

5.3 Communication Pattern

Communication pattern define how messages get exchanged between service providers and consumers in a B2B. In section 3.1.3, we identified configurable communication pattern as an important requirement for flexible data-driven BPI. Table 7 show a comparison of the BPI frameworks by their communication patterns.

Table 7. Comparison by Communication Patterns

Features	Web Portal Application	Integrated BPEL CoT	Simple SOA Publish/Subscribe	Policy-Based SOA Publish/Subscribe
Coupling	Tight	Tight	Loose	Loose
Interaction Type	Procedure-centric	Procedure-centric. Data-centric at design time	Data-centric	Data-centric
Interaction Pattern				
One-to-one	Yes	Yes	No	Yes
One-to-many	No Except when statically hard-coded.	No Except when statically built into the scripts	Yes Broadcast-based	Yes Controlled broadcast through Message, Topic and subscription policies.
Many-to-one	No	No	Yes	Yes
Many-to-many	No	No	Yes	Yes

5.3.1 Coupling

By coupling, we mean the ways processes get tied to one another. Web application portal integrate various applications through local and remote procedural calls. Each application depends on another for its internal process execution. The failure of one application usually stalls the entire process execution. Integrated BPEL CoT couple CoT services in a BPEL-process through *<partnerlinks>* relationships created at design time. Despite its orchestration abstraction, this type of relationship is still tightly coupled because each service must execute appropriately for the entire process execution to be successful.

The simple SOA publish/subscribe uses loosely coupled *asynchronous push* for message delivery. Message publishers, subscribers and the message broker are totally decoupled in time, space and synchronization. However, the management operations especially those used for subscription management are tightly coupled. Our policy-based framework extends this loose coupling to all interactions. We use a *message context handler* to receive and process all calls, both for management operations and message delivery. It defines a coupling framework that is not only loose but adaptable to the calling application.

5.3.2 Interaction Type

By interaction type, we mean how every interaction is perceived by the system. Web application portal and Integrated BPEL CoT use a procedure-centric interaction. A BPEL process internally represents data as messages but uses its *<Assign>* operation to relate messages to service interfaces. It is an abstracted form of making remote procedural calls. For instance, most BPEL scripts are littered with many *<Assign>* and *<invoke>* commands that take care of its many point-to-point invocation of external services. Interaction pattern in these frameworks are monolithic, hard-coded one-to-one linkages.

The simple SOA publish/subscribe define interaction based on data alone. That is, data is exchanged between systems without any strict compliance placed on the exchanged data structures. Being data-driven, the infrastructure doesn't care about the validation of the message exchanges. Messages are handled similarly and only treated simply as incoming and outgoing data. In our framework, a combination of declarative topic, message, and subscription policies determine how components are integrated and how their type of interaction.

5.3.3 Interaction Pattern

The Web application portal and the integrated CoT BPEL interconnect services based on point-to-point static linkages that are not adaptable to runtime manipulations. SOA publish/subscribe, on the hand supports only broadcast type interaction where messages are sent to the message broker which checks its subscriptions to determine the subscribing recipients, and have the message broadcast to those subscribers. Our Policy-based SOA publish/subscribe framework allows for point-to-point request/response communication and asynchronous broadcast interactions with PS components. However broadcasts can be controlled. Our framework uses both subscription and message policies to control message recipient. In addition to subscriptions, processes are able to exchange data the same way we exchange e-mails simply by pushing messages to the policy-based message broker and specifying the message recipients. Topic policies restrict publishing to only authorized publishers.

5.4 Message Delivery Service Quality

Message delivery service quality is an important requirement for data-driven B2B process integration (Section 3.1.4). A flexible service quality helps us ensure that messages are delivered on timely basis without manipulations and to the correct recipient. It accounts for delivery during down times, as well as disruptions and delays. Table 8 shows a comparison of our framework to other BPI frameworks in terms of message delivery techniques and service qualities.

Table 8. Comparison by message delivery techniques and service qualities

Features	Web Portal Application	Integrated BPEL CoT	Simple SOA Publish/Subscribe	Policy-Based SOA Publish/Subscribe
Delivery Mode	Pull	Push/Pull	Push	Push/Pull
Delivery Service Quality	No Dependent on transport protocol	No Depends on external web service standards like WS-Policy, WS-Reliability.	No Dependent on other WS standards.	Declarative Delivery Service Quality ("Best Efforts", "Reliable Delivery", "Pullpoint") and support for other WS-* standards.
Batched/Store and Forward	No	No	No	Yes Can be enabled through explicit <i>createPullpoint</i> request or embedded in subscriptions

5.4.1 Delivery Mode

Different applications require data to be delivered differently. Delivery modes depend on technologies and applications. Web portals *pull* data from various information sources. By design, custom connectors are setup with active information sources which get queried for data based on user activities. CoT WS-BPEL defaults to *polling* collaborating CoT services for information using the `<invoke>` command. In certain circumstances, external services can *push* information to a running process by through the `<onMessage>` construct. The simple SOA publish/subscribe default message delivery mode is *push* for all message notification needs. Likewise, our framework's default delivery mode is *push*. To support polling, services can define Pullpoints explicitly or through subscriptions, causing the policy broker to forward all message

to their Pullpoint queue. These services then use the *getMessage* request to pull the queued messages in batches when needed.

5.4.2 Delivery Service Quality

In our framework, subscription policies are used to define delivery service quality through `<Action>/<Delivery>/<SQ>` policy element. Default delivery is “*BestEfforts*” – *push* without delivery acknowledgment. Other options are “*Reliable*” – *push* with delivery acknowledgment and “*pullpoint*” – queue messages for the PS component to retrieve at a later time. Like other SOA frameworks, our framework can leverages other WS-standards like WS-Reliability and WS-Security to ensure reliable delivery.

5.4.3 Batched/Store and Forward

This support requires messages to be stored or batched for delivery at a more convenient time. Web portals and the Integrated CoT WS-BPEL have no mechanism for storing and forwarding messages. They also have no automated mechanism for dealing with offline processes that suddenly become unreachable during processes execution. Our first pass, simple SOA publish/subscribe implementation have no support for batched messages. However, the policy-based framework ties message batching to subscription policy delivery Service Quality. Except for subscriptions with “*BestEfforts*” Service Quality, messages are batched automatically to a *pullpoint* when a PS component is unreachable or when message delivery is not acknowledged.

5.5 Information Management

Information management evaluates the frameworks on the BPI requirement for dynamic business configuration (Section 3.1.6) based on their support for a common data model and use of declarative policies to control data sharing. It also evaluates the frameworks on the requirement for capturing and logging streaming data in a B2B using the surveillance infrastructure.

Table 9 below summarizes the comparison of the BPI frameworks by their information management approaches.

5.5.1 Common Data Model

BPI requires a common data model for easy integration with business partners and collaborators. Services implemented in conformance with this data model automatically integrate seamlessly with existing services and processes. Web application portal provide no such integration model with the exception of mashups. Liberty Alliance CoT specification is about common communication and messaging model. The entire specification is a complex of interface and data models defined in the WSDL of each of the CoT services.

Table 9. Comparison by Information Management approaches

Features	Web Portal Application	Integrated BPEL CoT	Simple SOA Publish/Subscribe	Policy-Based SOA Publish/Subscribe
Common Data Model	No	Yes (Complex – many WSDL documents for each service)	Yes (Interface-based)	Yes (Data-centric model)
Active Information Subscription Management	None Embedded in the application logic	None Described at design-time in the BPEL scripts	Managed by the subscriber through explicit calls like <i>Subscribe</i> , <i>Unsubscribe</i> , and <i>Renew</i> .	Declarative, rule-based subscriptions policies managed through the <i>Partner Service Component</i> .
Surveillance Infrastructure support	No Transactional databases are used for reporting	None	Not automated	Yes Surveillance infrastructure logs streaming data.

Simple SOA publish/subscribe message broker WSDL and type definitions provide a model of the communication interfaces to be implemented by publishers and subscribers. Our framework is built on common data model that define a general construct for all messages. It is built to support incremental updates and implementation. The policy message broker's defined interfaces individually support all data structures used in message exchanges.

5.5.2 Active Information Subscription Management

Interest to data is defined in subscriptions. A Web application portal defines these interests as the portal is being developed. If required, changes have to be made to different modules and adapters to reflect changes in subscriptions of connected processes. The same goes with Integrated CoT BPEL. Each customer business process defines a custom subscription to information in BPEL

scripts. Changes are not easily made without redesigning the set of entire processes. Our framework and the simple SOA publish/subscribe support declarative subscription to data. Message Broker operations like *Subscribe*, *Unsubscribe*, and *Renew* are used to manage subscriptions. Declarative subscription policies use XACML type construct to define rule-based subscriptions. Policy-based subscriptions include additional attributes like availability schedule, expiration, topic and content filters, delivery actions like optional Service Quality, message transformation and recipients. Our framework removes the burden of subscription management from the applications through the Partner Service component.

5.5.3 Surveillance Infrastructure Support

In a Web application portal, data sources are transactional databases using static structures to represent both active and historical information. In a streaming database model, where information representation and structure are continuously changing, this static nature of data representing and presentation is not appropriate. Integrated CoT BPEL provides surveillance through the Event Stats and the Audit Trail services. However, these are for performance management, legal compliance and non repudiation. The simple SOA publish/subscribe has no mechanism for capturing streaming data. It is left to each individual customer service to correlate and report on the data received. Integration with a surveillance infrastructure is not automatic and not adaptable to a dynamic information representational model. On the other hand, our framework integrates with the surveillance infrastructure to automatically capture and log all streaming data by adapting itself to changing data structure (topics) and messages. It supports stateful reporting through the surveillance infrastructure.

5.6 Evaluation of Engineering Effort

We have shown that our framework provides an enhanced set of features for creating flexible data-driven BPI in B2B networks. In this section, each of our four architectures was evaluated with respect to the engineering effort to develop, deploy, maintain as well as the complexity for adapting and debugging BPI processes. Table 10 summarizes the evaluation of efforts and complexity to creating data-driven processes.

Table 10. Evaluation of efforts and complexity to creating data-driven processes

Features	Web Portal Application	Integrated BPEL CoT	Simple SOA Publish/Subscribe	Policy-Based SOA Publish/Subscribe
Development and Deployment efforts	Low-level programming (complex and static)	More abstraction, Less low-level programming but gets complex with more services.	More abstraction to integration. Decomposition of services by message Topics.	Integration is abstracted in PS Components. Presents an easy-to-use interface to applications and processes. Requires zero knowledge of message structures.
Maintenance and Debugging efforts	Complex, embedded in low-level logic. Good development skills.	Done at the BPEL script. Easier with GUI BPEL designer. Debugging is done through SOAP monitors.	Easy maintenance through modification to topics and subscriptions. No GUI support, requires some programming skills.	No GUI support. Most maintenance operations is done through declarative XML configuration files. Requires little programming after framework is setup.
Complexity	Yes	Yes	Development (Complex), Integration (Less complex)	Development (Complex), Integration (Not complex)
Protocol Independence	No	At the enterprise level because of WS-BPEL processes.	Internet-scale	Internet-scale
Business agility	No	Moderate: Through BPEL commands	Moderate: Only through subscription to topics.	High: Subscriptions, scheduling, flexible filtering and transformation, flexible delivery.
Competencies of Developers	Highly Technical	Technical	Moderate technical skills (Data)	Moderately technical skills (Data and Policies)

In web application portals business activities are defined in low-level programming details and rules of object interactions. Defining process integration using this approach requires lots of development skills, extensive knowledge of programming languages, service message details, and handling of messages from external sources. Much time and effort is usually invested in low-level technical details that require users with high technical competencies.

Integrated CoT WS-BPEL provides a higher abstraction to process composition. It requires services to be abstracted as Web services and it uses BPEL scripts to define their interactions in a workflow fashion. These scripts are easy to modify as long as message-level details are not affected. A downside to BPEL is that it becomes more and more complex as more services get integrated with a business process. Though understanding BPEL scripts is not highly technical, orchestrating Liberty Alliance services with BPEL scripts does require substantial technical skills.

SOA publish/subscribe totally eliminates infrastructure details as each service in the process is either a publisher or a subscriber. However a process is usually decomposed into a set of message topics that describes the message exchanges from each of the parties. It requires each service to be able to relay and receive messages to the message broker using broker specific message structure. Defining and handling asynchronous interaction can be quite complicated in terms of development time and user competence. However it becomes much simpler when message handlers are already developed and deployed. Integrating the handlers with applications requires little development skills. After coupling, managing and debugging processes interactions require very minimal development skills.

Like the simple SOA publish/subscribe, the Policy-based SOA publish/subscribe requires a highly technical implementation. However, the components are easily integrated with SOA and legacy application. Processes are tied to Partner Service components which automatically provides an interface for them participate in SOA publish/subscribe. Setup for registration, subscription is done flexibly through configuration files containing subscriptions and other policy broker settings. Messages are defined as topics and their attributes are used to define content filters and message transformation rules. Most modifications are done declaratively and require little or no development skills.

5.7 Relationship with related Web Service specifications

In this section we relate our framework to the two most relevant WS specifications for BPL. Sections 2.5.5 and 2.5.6 describe WS-Notification and WS-Eventing in details. We analyze the specifications against our framework, pointing out the similarities and the reasons why both

specifications are not adequate for data-driven BPI. Table 11 below summarizes the relationship of both specifications to our framework.

Table 11. Relationship with WS-Notification and WS-Eventing

Feature	Specifications		Policy-based SOA Publish/Subscribe
	WS-Notification	WS-Eventing	
Topic/Content Filtering	<i>Hierarchical TopicExpression</i> (Defined in WS-Topics), <i>ProducerProperties</i> and <i>MessageContent</i> . Content filters use XPath/xQuery/Any Expressions.	Any filter (Depends on implementation) Content filters use XPath/xQuery/Any Expressions.	Self-describing topics. Combination of <i>Topic</i> and <i>Content-based</i> filters using XPath/xQuery Expressions.
Message Transformation	Not Defined	Not Defined	Declarative and part of the subscription policy
Message Delivery and Service Quality	<i>Push</i> (No Acks) <i>Pull</i> (cannot be defined within subscriptions, but supported through Pullpoints) Service Quality is dependent of other WS specifications	Push (Supports Acknowledgment) Pull, batched. Service Quality is dependent of other WS specifications	Push, Pull, DataQuery, batched (Support Ack). All the delivery modes can be declaratively defined in subscription policies. Declarative Service Quality and also uses other WS specifications
Use of an Intermediary	<i>Yes: Brokered</i>	No: Direct	Yes: Brokered
Publisher Registration	Yes	Not Defined	Yes
Subscriber Registration	Subscribers are identified using WS-Addressing endpoint Reference	Subscribers are identified using WS-Addressing endpoint Reference	Subscriber identification is through broker issued unique identifier and can change WS-Addressing endpoint reference for the same identifier.
Security	Not Defined	Not Defined	Message level authentication
Dependency on other specifications	WS-Addressing, WS-BaseNotification, WS-BrokeredNotification, WS-Topics, WS-ResourceFramework	WS-Address	WS-Addressing
Policy-based approach	No	No	Yes
Support for Surveillance Infrastructure	No	No	Yes

Feature	Specifications		Policy-based SOA Publish/Subscribe
	WS-Notification	WS-Eventing	
Support for Legacy Systems	No	No	Yes

5.7.1 Topic/Content Filtering

WS-Notification defines Topic filters through hierarchical *TopicExpressions* using the WS-Topics specification. Topics are represented by XPath-like tree structures and may optionally add patterns to describe message types and optional attributes. WS-Eventing on the other hand, places no restriction to the definition of a Topic and how it is represented. Similarly, our framework does not impose a specific pattern for a topic but requires an attribute set representing the fields of the topic message. This attribute set is also used by the surveillance infrastructure to prepare persistence tables for messages linked to the topic. WS-Notification and WS-Eventing supporting proprietary filtering approaches but use XPath queries (*xQueries*) for content filtering on XML-based messages. In our approach, we use XPath queries but currently without support for multiple namespaces. Depending on the scenario, other types of filtering could be implemented in the policy broker.

5.7.2 Message Transformation

WS-Eventing and WS-Notification do not provide a mechanism for manipulating messages before delivery. Message that match the filters are delivered as published from the publisher. In our framework, message transformation is declarative and attached to a subscription policy. Aggregated messages like *PainReport* described in the scenario, usually sent as reports from the surveillance infrastructure can be transformed as they get sent to individual subscribers.

5.7.3 Message Delivery and Service Quality

Like most publish/subscribe system, both specifications support the *Push* type message delivery. WS-Notification does not require acknowledgements to published messages. Except during the retrieval of messages from Pullpoint queues, WS-Notification has no support for pull type delivery in subscriptions. WS-Eventing on the other hand supports both acknowledged and unacknowledged *push* message delivery. Like WS-Notification, it supports batched delivery by

using messages queues to batch messages for subscribers. *Pull* delivery is then used by the subscriber to retrieve these messages batches when needed. Our framework supports *push*, *push* with acknowledgement, *pull*, and batched (Pullpoints) delivery modes. It also provides support for describing message delivery service quality declaratively in subscriptions using policies.

Message delivery service quality is not defined in both specifications. Their delivery service quality is dependent on other WS specifications. Our framework declaratively ties delivery modes to subscriptions thus providing a structured means of internally controlling delivery Service Quality within the messaging infrastructure.

5.7.4 Use of an Intermediary

WS-Notification supports both direct and brokered notification through WS-BaseNotification and WS-BrokeredNotification specifications. WS-Eventing does not use an intermediary for message delivery. However it utilises a subscription manager in managing subscriptions while publishers publish messages directly to the subscribers. Our framework is brokered, that is the message broker routes all messages. We currently provide no infrastructure support for direct notification. However we support point-to-point notification through message policies. Therefore message recipients can be defined and tied to messages, forcing the broker to relay the messages only to specified recipients.

5.7.5 Identification of Subscribers and Publishers

Most publish/subscribe systems including the WS-Notification and WS-Eventing does not identify the publishers of information. In addition, Subscribers are identified by their WS-Addressing endpoint references. Our framework identified the need for a flexible identification system for subscribers since endpoint addresses are usually not very appropriate. The fact that Web services belonging to an organization change WSDL addresses for various reasons should not affect the subscription references to that service. Since this information is likely to change over time, we employ a different identification pattern that uses globally significant identifier to uniquely identity an endpoint. The Partner Service registration process identifies the role of the service as either a publisher or a subscriber or both. PS components can change their EPR information without affecting the subscriptions references from other collaborating services.

5.7.6 Security

Security is a very important consideration to the Policy-based broker. As a part of the registration process, each PS component is issued a security token. This token is only known by the policy broker and the customer service. The PS component uses this token to encrypt wrapped messages before sending them to the broker. The broker uses the *PartnerID* sent in plain text to identify the source of the document, before retrieving the *SecToken* to decrypt and process the message. *SecTokens* are refreshed after each session with the broker. Services that go offline renew their *SecTokens* when they come online before sending and receiving messages.

5.7.7 Dependency on other specifications

WS-Notification is a very complex specification. It is tightly integrated with a number of WS-* specifications. WS-Addressing is used for endpoint addressing, WS-BaseNotification describes the interfaces of the notification producer, notification consumers, while WS-BrokeredNotification describes the interfaces for the notification broker, WS-Topics describes the topic namespaces, WS-ResourceFramework is used to describe resources tied to notification messages.

Our framework and WS-Eventing presents a simple model to notification. WS-Addressing is used for addressing the subscriber endpoint addresses. The data model contains the structure for all message exchanges and does not require additional complex namespaces.

5.7.8 Policy-based approach

WS-Notification and WS-Eventing provide no support for policies. In comparison, our framework uses declarative policies to define event-driven BPI for B2B. It views a B2B as a streaming database where policies not only define topic and content filtering but also availability, message transformation, and flexible delivery Service Quality. Messages can define recipients through policies and topics can be restricted to specific publishers using policies.

5.7.9 Stateful data and Surveillance infrastructure

WS-Notification and WS-Eventing describe no mechanism for supporting transactional systems often required access to stateful data from storage databases. The policy-based message broker does not provide the storage by itself but its topic registration integrates with a surveillance

infrastructure to automatically generate representational tables for data. Messages are forwarded also to the surveillance system for logging. *DataQuery* requests are used by the policy broker to provide stateful data and historical information to applications through the same infrastructure.

5.7.10 Support for Legacy systems

Applications supporting SOA publish/subscribe interactions may need some complex re-engineering of their architecture. We use Partner Service (PS) components to interface the policy broker with other applications. These component adapters implement both the publisher and the subscriber to the policy broker. They also implement the messaging model and its context based management operations. Legacy application could use the simple interfaces similar to the traditional email inbox and outbox to exchange application messages.

5.7.11 Summary

We identified WS-Notification (OASIS) and WS-Eventing (W3C, 2006) as the two popular specifications to SOA notification (Sections, 2.5.5 and 2.5.6). WS-Notification supports both direct and brokered notifications. WS-Eventing on the other hand supports only direct notification. In comparison to WS notification, it is lightweight, but does not support messaging intermediaries like the Message Broker. The specifications are not compatible because of their strengths in various application areas. Our framework is a Brokered model but without the complexity of the WS Notification specification. It attempts to reduce this flexibility by defining a flexible messaging and communication model. It also tries to marry the flexibility of WS-Eventing subscription management operations as well as support for declarative push/pull/batched (pullpoint) subscriptions to a brokered publish/subscribe framework. Using the right adapter components, our framework is able to support applications developed using both specifications.

5.8 Relationship of Policy Model to XACML

Our policy-based SOA publish/subscribe framework employs a similar framework to XACML RBAC but applied to streaming data model. In this section, we use Table 12 to illustrate the similarities between these frameworks.

Table 12. Similarities of Functionality between XACML RBAC and the Policy-based SOA Publish/Subscribe

	XACML RBAC	Policy-based SOA Publish/Subscribe
Resources	Application messages, file systems, xml data	Messages received as application data, file system data, xml data
Policy Enforcement	Policy Enforcement Point (PEP) Receives requests, packages them as for the Policy Decision Point (PDP)	Partner Service Component receives application data, packages data as wrapped messages for the message broker
Policy Execution	PDP executes XACML policies against assertions	Policy engine executes subscription policies against messages.
Context Handling	Uses context handing to provide semantics to request processing	Uses a message context handler to provide semantics to the handling of notification messages
Subject Categorization	Described as Roles	Described as Topics
Information Handling	Access to subject, resource and environmental attributes are provided by the Policy Information Point (PIP)	Access to reports, Stateful data is provided by the surveillance infrastructure.

Resources: XACML RBAC infrastructure coordinates access and enforces restrictions to resource constrained resources like application messages, file system and other types of xml resources. Our framework provides a framework for controlling the sharing of similar resources.

Policy Enforcement: In XACML, polices are enforced by the Policy Enforcement Point (PEP). It mediates between the subject and security-constrained resources. It creates authorization requests, pushes them to the Policy Decision Point (PDP) and subsequently handles access control response after policy execution. In our framework, the Partner Service component performs a similar role. It receives application data, packages them and forwards them to policy broker to be routed to subscribers according to result of the execution of subscription policies.

Policy Execution: In XACML, the PDP access available policies from the Policy Administration Point (PAP) and executes them against access control assertions. In our framework, policies are parsed and stored in the Policy database and executed against incoming messages through the policy engine.

Context Handling: In XACML, assertion contexts are handled to provide semantics to request processing. All assertions are received and processed in the same manner. Context requirements

could mean accessing the Policy Information Point (PIP) for environmental variables that help in the execution of policies on assertions. Our framework uses a similar approach. Messages are received centrally with their initial processing done through the message context handler. Depending on the context presented applicable low-level handlers are engaged in the processing of the request. Context handling provides semantics to the handling of notification messages.

Subject Categorization: XACML policies are usually enforced on a subject's access to an object (a security constrained resource). Subjects are categorized by their roles which are defined in attributes. In our framework, messages are categorized into Topics. Further classification can be provided by combining different content filters of the message attribute.

Information handling: In processing assertions, XACML uses the Policy Information Point (PIP) to access attributes of the subjects, resources and environmental variables. Essentially, these represent stateful data. Our framework handles information similarly but applied to a B2B streaming database. The surveillance infrastructure plays the role of the PIP but instead provide access to stateful reports and data.

Chapter 6. Conclusions

6.1 Summary of Contributions

In this thesis, we presented a new approach to achieving flexible data sharing and BPI in B2B networks using a Policy-based SOA publish/subscribe framework. The framework uses declarative policies to define subscriptions that combine topic and content-based filtering, achieve both point-to-point and broadcast communication, automatic message format translation and achieve reliable message delivery through a service quality aware publish/subscribe.

In section 3.1, we identified the essential requirements for a BPI framework to support flexible data sharing in a B2B network. By identifying these requirements, we set the base for conceptualizing our policy-based SOA publish/subscribe framework.

The simple SOA publish/subscribe was leveraged as the basis for achieving publish/subscribe communication pattern using SOA. In Section 3.2.1, we described the simple SOA Message broker as a middleware for tying together publish/subscribe components in loosely coupled, asynchronous interaction pattern. However, this message broker is limited because of its lack of expressiveness.

We introduced the Partner service component as an adapter component that can be used by applications in participating in SOA publish/subscribe. We described how this component can manage subscriptions on behalf of collaborating applications. In section 3.2.2, we proposed a data sharing model and described how we used a Topic Registrar to create “self describing topics”. These topics not only categorize messages by an identifier but use an attribute set to describe the structure of the messages they represent. We showed how the surveillance infrastructure leverages these topics to support automatic persistence of streaming data.

In Section 3.3, we introduced our policy-based SOA publish/subscribe framework. In doing so, we proposed a policy model as well as the policy language that describe declarative policy-based

subscriptions. This policy model is based on XACML Role Based Access Control (RBAC) but applied to a streaming database. We described the policy-based message broker and how it handles policy-based subscriptions and messages as well as flexible delivery formats, Service Quality, and both system-wide and topic-specific constraints. We showed how it handles the batching of message using pullpoints and supported stateful data and reports through the surveillance infrastructure. In section 4.4, we described the policy-based SOA publish/subscribe prototype built to implement the Palliative Severe Pain Management scenario. Using this scenario, we showed details of Partner Service registration, topic creation, creation of subscription policies and the management of notification messages and Pullpoints.

In Chapter 5, we compared this framework to other popular BPI framework based on Information Topic, communication pattern, message delivery service quality and information management. In Section 5.6, each of our four architectures was evaluated with respect to the engineering effort to develop, deploy, maintain as well as the complexity of adapting and debugging BPI processes. In Section 5.7, we evaluated the framework against the popular related Web service specifications. Finally we showed the similarities in infrastructure between our framework and XACML RBAC in Section 5.8.

6.2 Future Work

6.2.1 Distributed Policy-based SOA Publish/Subscribe

Policy-based SOA publish/subscribe framework is based on a single SOA message broker. The policy broker processes subscriptions, topic registration, partner service registration, message routing, handling of pullpoints, etc. In large-scale event management environments with high volume messages, and applications, this could create a single point of failure and performance constraint for the enterprise.

A distributed approach to policy-based SOA publish/subscribe is a good area for future research. It requires a distributed policy model based on a scalable rule-based routing protocol that can be shared by multiple message brokers. Partner Services components broadcast and receive message from the nearest broker infrastructure while distributed routing tables can be used by the message brokers to determine the best route to subscribers. A distributed message matching algorithm and

policy database can be leveraged by the message brokers to match and track policy-based subscriptions.

6.2.2 Identity Management Infrastructure

One of requirements for flexible data sharing and BPI in B2B is identity management. Our approach ensures the security of message exchanges by encrypting message attributes and payloads on transit using the security tokens of the Partner Service component. Our assumption is that users of these applications are authenticated internally in their domain and therefore have the permissions to send and receive messages based on their subscriptions. However, we have no mechanism for controlling the users of the applications using the Partner Service Component. An interesting area of future work is that creating an Identity Management Infrastructure that can be tied to the policy broker to define access control permission to subscription resources linking application users and their permissions.

6.2.3 Automatic Definition of Subscription Policies

Our policy language is based on XML. However, our framework did not tie subscription policies to user permissions and rights. Rather they were tied to Partner Service identifiers. We have not defined a standard interface for the policy administrator to automatically setup policies as new users and roles are being created in the enterprise. An important area of future research is to define a more flexible mechanism for automatically defining subscription policies and tying them to users' roles and permissions.

6.2.4 Partner Service Component Support for BPEL

A BPEL process orchestrates the execution of Web services in a workflow manner. Each collaborating service is defined as a Partner Service within the BPEL scripts and its WSDL must be imported during the design of the business process using the BPEL designer. However, when there are multiple services that play similar roles and exchange similar messages, it is impossible to build them into the BPEL script dynamically. A Partner Service component can be used to create this interface to BPEL. It will link multiple publishers and subscribers dynamically to the message exchanges of the BPEL process while defining a single interface to the BPEL designer and the running business process.

References

- Bailey, J., Poulouvassilis, A., & Wood, P. (2002). *An event-condition-action language for XML*. Paper presented at the Proceedings of the 11th international conference on World Wide Web, Honolulu, Hawaii, USA.
- Barreto, C., Bullard, V., Erl, T., Evdemon, J., Jordan, D., Kand, K., et al. (2007). Web Services Business Process Execution Language Version 2.0. Retrieved December 2008
- Boglaev, Y. (2005). *Interchange of ECA Rules with Xpath expressions*. Paper presented at the W3C Workshop on Rules Language for Interoperability.
- Brambilla, M., Ceri, S., Passamani, M., & Riccio, A. (2004). *Managing Asynchronous Web Services Interactions*. Paper presented at the Proceedings of the IEEE International Conference on Web Services.
- Brogi, A., Corfini, S., & Popescu, R. (2005). *Composition-Oriented Service Discovery* (Vol. 2628/2005): Springer Berlin.
- C.Cahill, C. Canales, H. Le Van Gong, P. Madsen, E. M., & G.Whitehead. (2008). Liberty Alliance Web Services Framework: A Technical Overview., from <http://www.projectliberty.org/specs>
- Carzaniga, A., Rosenblum, D., & Wolf, A. (2001). Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3), 332-383.
- Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., & Weerawarana, S. (2002). Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2), 86-93.
- Damiani, E., Vimercati, S., Jajodia, S., Paraboschi, S., & Samarati, P. (2003). *Balancing confidentiality and efficiency in untrusted relational DBMSs*. Paper presented at the Proceedings of the 10th ACM conference on Computer and communications security, Washington D.C., USA.
- Doshi, C. (2008). *An Integrated Trusted Processes Framework for Consumer-facing B2B networks*. University of Ottawa, Ottawa.
- Dufour, P. (2007). Event Stream Processing: Distributed Detection and Active Databases. Retrieved December 2008
- Eidson, B., Maron, J., Pavlik, G., & Raheja, R. (2005). *SOA and the future of Application Development*. . Paper presented at the First International Workshop on Design of Service-Oriented Applications (WDSOA'05).

- Erradi, A., Maheshwari, P., & Tasic, V. (2007). *WS-Policy based Monitoring of Composite Web Services*. Paper presented at the ECOWS apos;07. Fifth European Conference on Web Services
- Eugster, P., Felber, P., Guerraoui, R., & Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2), 114-131.
- Ferraiolo, D., & Kuhn, R. (1992). *Role-Based Access Control*. Paper presented at the Proceedings of 15th National Computer Security Conference.
- Foster, I., Kesselman, C., Nick, J., & Tuecke, S. (2002). Grid Services for Distributed System Integration. *Computer*, 35(6), 37-46.
- Gisolfi, D. (2001). Web Services Architect, Part 3: Is Web services the reincarnation of CORBA? [Electronic Version]. *IBM DeveloperWorks*. Retrieved December 2008 from <http://www.ibm.com/developerworks/webservices/library/ws-arc3/>.
- Gupta, A., Sahin, O., Agrawal, D., & El-Abbadi, A. (2004). *Meghdoot: content-based publish/subscribe over P2P networks*. Paper presented at the Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, Toronto, Canada.
- Hahn, R. (2006). Oracle Unveils Event-Driven Middleware Suite :Enables Real-Time Response to Rapidly Changing Business and Market Conditions [Electronic Version]. *Oracle Press Release*. Retrieved December 2008 from http://www.oracle.com/corporate/press/2006_jun/oracle-eda-suite.html.
- Hongzhou, L., Venugopalan, R., & Emin, G. S. (2005). *Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews*. Paper presented at the Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement, Berkeley, CA.
- Huang, Y., & Gannon, D. (2006). *A Comparative Study of Web Services-based Event Notification Specifications*. Paper presented at the Proceedings of the 2006 International Conference Workshops on Parallel Processing.
- Huang, Y., & Garcia-Molina, H. (2004). Publish/subscribe in a mobile environment. *Wirel. Netw.*, 10(6), 643-652.
- Ira. Lewis, & Alexander, T. (2000). Third-party logistics: Leveraging information technology. *Journal of Business Logistics*, 21(2), 173-185.
- Jeyaraman, R., Modi, V., Driscoll, D., Bullen, G., & Nixon, T. (2008). Understanding Devices Profile for Web Services, WS-Discovery, and SOAP-over-UDP: MSDN, .NET Framework Developer Center.
- Kuziemsky, C., Weber-Jahnke, J., Lau, F., & Downing, M. (2008). An Interdisciplinary Computer-based Information Tool for Palliative Severe Pain Management. *Journal of the American Medical Informatics Association*, 15(3), 375-382.

- Le Hegaret, P. (2003). W3C Vision of Standards. Retrieved December 2008, from <http://www.w3c.org/2003/Talks/1211-xml2003-wssoa/slide2-0.html>
- Leymann, F., Roller, D., & Thatte, S. (2007). Goals of the BPEL4WS Specification. Retrieved December 2008
- Liang-Jie, Z., Haifei, L., Henry, C., & Tian, C. (2002). *XML-based Advanced UDDI Search Mechanism for B2B Integration*. Paper presented at the Proceedings of the Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS'02).
- Liang-Jie, Z., Tian, C., Henry, C., & Jen-Yao, C. (2004). XML-Based Advanced UDDI Search Mechanism for B2B Integration. *Electronic Commerce Research*, 3(1-2), 25-42.
- LibertyAlliance. (2008). Liberty Specification. Retrieved December 2008, 2008, from http://www.projectliberty.org/liberty/resource_center/specifications
- Liu, Y., & Plale, B. (2003). Survey of Publish Subscribe Event Systems. [Electronic Version]. *Indian University Department of Computer Science, Technical Report TR574*. Retrieved December 2008.
- Ma, J., Xu, G., Wang, J., & Huang, T. (2006). *A Semantic Publish/subscribe System for Selective Dissemination of the RSS Documents*. Paper presented at the Proceedings of the Fifth International Conference on Grid and Cooperative Computing.
- McCarthy, J. (2002). Reap the benefits of document style Web services. *IBM DeveloperWorks*. Retrieved December 2008, from <http://www.ibm.com/developerworks/webservices/library/ws-docstyle.html>
- Mckendrick, J. (2000). UDDI: Too Many Standards, Too Little lime? - Industry Trend or Event. *Technology Industry*.
- Michelson, B. (2006). Event-Driven Architecture Overview, Event-Driven SOA is just part of the EDA Story. *Business-Driven Architecture*. Retrieved December 2008, from <http://dx.doi.org/10.1571/bda2-2-06cc>
- Microsoft. (2006). Devices Profile for Web Services. Retrieved December 2008, from <http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf>
- Mudhakar, S., & Ling, L. (2005). *Securing publish-subscribe overlay services with EventGuard*. Paper presented at the Proceedings of the 12th ACM conference on Computer and communications security, Alexandria, VA, USA.
- Natis, Y. (2003). Service-Oriented Architecture Scenario. *Gartner's Positions on the Five Hottest IT Topics and Trends in 2005*. Retrieved December 2008, from <http://www.gartner.com/>
- Niblett, P., & Graham, S. (2005). Events and service-oriented architecture: The OASIS Web Services Notification Specifications. *IBM Systems Journal*., 44(4).

- Nolan, P. (2004). Understand WS-Policy processing [Electronic Version]. *IBM DeveloperWorks*. Retrieved December 2008 from <http://www.ibm.com/developerworks/webservices/library/ws-policy.html>.
- O'Connel, B., & Stanford-Clark, A. (2005). Using MQ Telemetry Transport with WebSphere Business Integration Message Broker. *IBM DeveloperWorks*.
- OASIS-WSRF. Web Services Resource Framework. Retrieved December 2008, from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf
- OASIS. Web Services Notification (WSN). Retrieved December 2008, from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn
- OASIS. (2001). UDDI Spec TC. Retrieved December 2008, from http://www.uddi.org/pubs/uddi_v3.htm
- OASIS. (2003). eXtensible Access Control Markup Language (XACML) OASIS Standard. Retrieved December 2008, from <http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf>
- OASIS. (2004). Web Services Brokered Notification 1.2 (WS-Notification) Retrieved December 2008, 2008
- OASIS. (2006). Web Services Topics 1.3 (WS-Topics). In W. Vambenepe, S. Graham & P. Niblett (Eds.) (Vol. 2008).
- Opyrchal, L. (2004). *Content-based publish subscribe systems: scalability and security*. University of Michigan.
- Pallickara, S., & Fox, G. (2005). *An Analysis of Notification Related Specifications for Web/Grid Applications*. Paper presented at the Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume 02.
- Papazoglou, M., & Ribbers, P. (2006). *e-Business organizational and technical foundations*.: John Wiley & Sons Ltd.
- Pereira, J., Fabret, F., Jacobsen, H.-A., Llirbat, F., & Shasha, D. (2001). *WebFilter: A High-throughput XML-based Publish and Subscribe System*. Paper presented at the Proceedings of the 27th International Conference on Very Large Data Bases.
- Peyton, L., Doshi, C., & Seguin, P. (2007). *An Audit Trail Service to Enhance Privacy Compliance in Federated Identity Management*. Paper presented at the Proceedings of CASCON 2007, ACM, Toronto, October 2007.
- Phillipa, O., & Arthur, H. (2007). Guided interaction: A mechanism to enable ad hoc service interaction. *Information Systems Frontiers*, 9(1), 29-51.
- Pieczkowski, E. (2007). Is BPEL Positioned for the Future? [Electronic Version]. *TechTracker*. Retrieved December 2008 from

<http://www.networkcomputing.com/channels/enterpriseapps/showArticle.jhtml?queryText=&articleID=198000132&pgno=2>.

- Premkumar, G., Ramamurthy, K., & Sree, N. (1994). Implementation of electronic data interchange: an innovation diffusion perspective. *J. Manage. Inf. Syst.*, 11(2), 157-186.
- Ross-Talbot, S., Tabet, S., Chakravarthy, S., & Brown, G. (2003). *A generalized RuleML-based Declarative Policy specification language for Web Services*. Paper presented at the W3C Workshop on Constraints and Capabilities for Web Services.
- Sabbah, D. (2004). *Bring Grid and Web Services Together*. Paper presented at the Globus World, San Francisco, CA.
- SAML. Security Assertion Markup Language (SAML). Retrieved December 2008, from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security
- Scholar, W. (2007). BPEL Service Engine User's Guide. Retrieved December 2008, from <https://open-esb.dev.java.net/kb/preview3/ep-bpel-se.html>
- SDN. (2008). Java Message Service (JMS). *SDN Library* Retrieved December 2008, from <http://java.sun.com/products/jms/>
- Shankar, C., Ranganathan, A., & Campbell, R. (2005). *An ECA-P Policy-based Framework for Managing Ubiquitous Computing Environments*. Paper presented at the Proceedings of the The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services.
- Shin, D., Ahn, G., & Shenoy, P. (2004). Ensuring Information Assurance in Federated Identity Management. *IEEE Intl. Conference on Performance, Computing, and Communications*, 821-826.
- Stanford-Clark, A. (2004). *The importance of Internet standards and other topics*. Paper presented at the Open Control Systems - The importance of Industrial Standards.
- Stefansson, G. (2002). Business-to-business data sharing: A source of integration of supply chains. *International journal of production economics*, 75, 135-146.
- Terpstra, W., Behnel, S., Fiege, L., Zeidler, A., & Buchmann, A. (2003). *A peer-to-peer approach to content-based publish/subscribe*. Paper presented at the Proceedings of the 2nd international workshop on Distributed event-based systems, San Diego, California.
- Themistocleous, M., & Irani, Z. (2001). Benchmarking the benefits and barriers of application integration. *Benchmarking*, 8(4), 317-331.
- Themistocleous, M., & Irani, Z. (2002). *Evaluating and Adopting Application Integration: The Case of a Multinational Petroleum Company*. Paper presented at the Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9 - Volume 9.

- Thomas, N. (2002). When Should I Use JMS? [Electronic Version]. *JDJ - Industry Commentary*. Retrieved 10 Oct. 2008 from <http://java.sys-con.com/node/36813>.
- Tyagi, S. (2004). Patterns and Strategies for Building Document-Based Web Services [Electronic Version], 1-6. Retrieved December 2008 from <http://java.sun.com/developer/technicalArticles/xml/jaxrpcpatterns/index.html>.
- Tzelepi, S., Koukopoulos, D., & Pangalos, G. (2001). *A flexible content and context-based access control model for multimedia medical image database systems*. Paper presented at the Proceedings of the 2001 workshop on Multimedia and security: new challenges, Ottawa, Ontario, Canada.
- USG. (2001). *Evaluating and Adopting Application Integration*. Paper presented at the Aeronautical Telecommunication Network (ATN) Seminar.
- W3C. World Wide Web Consortium. Retrieved December 2008, from <http://www.w3.org/>
- W3C. (2005). Web Services Addressing 1.0 - Core. Retrieved December 2008, from <http://www.w3.org/TR/2005/WD-ws-addr-core-20050215/>
- W3C. (2006). Web Service Eventing. December 2008, from <http://www.w3.org/Submission/WS-Eventing/>
- W3C. (2007). Web Services Policy 1.5 - Framework. Retrieved December 2008, from <http://www.w3.org/TR/ws-policy/>
- WebLogic. (2008). Programming WebLogic Web Services. Retrieved December 2008, from <http://e-docs.bea.com/wls/docs81/webserv/overview.html#1066697>
- Weigand, H., Heuvel, W.-J. v. d., & Hiel, M. (2008). *Rule-Based Service Composition and Service-Oriented Business Rule Management*. Paper presented at the ReMoD 2008.
- WHO. (2008). WHO Definition of Palliative Care. Retrieved December 2008, from <http://www.who.int/cancer/palliative/definition/en/>
- Wikipedia. (2008). Quality of Service [Electronic Version]. Retrieved December 2008 from http://en.wikipedia.org/wiki/Quality_of_service

Appendix A – Policy SOA Pub/Sub Data Model

In this appendix, we show the XML schema defined data model for all message exchanges for the policy-based SOA publish/subscribe. This is then followed by the WSDL definition for the policy-based message broker and the PS component.

Schema

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="policybroker.site.uottawa.ca"
  elementFormDefault="qualified" xmlns:tns="policybroker.site.uottawa.ca">

  <xsd:complexType name="PeriodType">
    <xsd:sequence>
      <xsd:element name="From" type="xsd:time"/>
      <xsd:element name="To" type="xsd:time"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="RecipientsType">
    <xsd:sequence>
      <xsd:element name="SubscriberReference" type="xsd:string" maxOccurs="unbounded"
nillable="true"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="PublishersType">
    <xsd:sequence>
      <xsd:element name="PublisherReference" maxOccurs="unbounded" minOccurs="0"
type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="ContentFilterType">
    <xsd:sequence>
      <xsd:element name="Query" type="xsd:string" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="filterCombiningAlgorithm" type="xsd:string"/>
  </xsd:complexType>

  <xsd:complexType name="TopicFilterType">
    <xsd:sequence>
      <xsd:element name="Topic" maxOccurs="unbounded" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="filterCombiningAlgorithm" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="FiltersType">
    <xsd:sequence>
      <xsd:element name="TopicFilter" nillable="true" minOccurs="0" maxOccurs="1"
type="tns:TopicFilterType" />
      <xsd:element name="ContentFilter" minOccurs="0" nillable="true"
type="tns:ContentFilterType" />
      <xsd:element name="Publishers" minOccurs="0" nillable="true"
type="tns:PublishersType" />
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

```

<xsd:complexType name="MessageSectionType">
  <xsd:sequence>
    <xsd:element name="Section" minOccurs="1" maxOccurs="unbounded" nillable="true"
type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="filterCondition" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="DeliveryType">
  <xsd:sequence>
    <xsd:element name="Service Quality" minOccurs="0"/>
    <xsd:element name="Recipients" minOccurs="0" type="tns:RecipientsType" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="ActionsType">
  <xsd:sequence>
    <xsd:element name="MessageSection" minOccurs="0" type="tns:MessageSectionType"
maxOccurs="2"/>
    <xsd:element name="Delivery" type="tns:DeliveryType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="RuleType">
  <xsd:sequence>
    <xsd:element name="ApplicablePeriod" nillable="true" minOccurs="0"
type="tns:PeriodType" />
    <xsd:element name="TerminationTime" type="xsd:dateTime" minOccurs="0"/>
    <xsd:element name="Filters" type="tns:FiltersType" minOccurs="0"/>
    <xsd:element name="Actions" minOccurs="0" type="tns:ActionsType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="FaultType">
  <xsd:sequence>
    <xsd:element name="FaultMessage" minOccurs="0" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="type" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="ResponseType">
  <xsd:sequence>
    <xsd:element name="Status" type="xsd:int"/>
    <xsd:element name="Fault" type="tns:FaultType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="PolicyType">
  <xsd:sequence>
    <xsd:element name="Rule" maxOccurs="unbounded" type="tns:RuleType"/>
  </xsd:sequence>
  <xsd:attribute name="type" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="SubscribeType">
  <xsd:sequence>
    <xsd:element name="SubscriberReference" type="xsd:string"/>
    <xsd:element name="SubscriptionReference" nillable="true" minOccurs="0"
type="xsd:string"/>
    <xsd:element name="SubscriptionPolicy" type="tns:PolicyType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="SubscribeResponseType">
  <xsd:sequence>
    <xsd:element name="SubscriptionReference" type="xsd:string"/>
    <xsd:element name="CurrentTime" type="xsd:dateTime" minOccurs="0"/>
    <xsd:element name="TerminationTime" minOccurs="0" type="xsd:dateTime"/>
    <xsd:element name="Fault" minOccurs="0" type="tns:FaultType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="GetMessagesType">
  <xsd:sequence>
    <xsd:element name="PartnerReference" type="xsd:string"/>
    <xsd:element name="SubscriptionRef" minOccurs="0" type="xsd:string"/>
    <xsd:element name="MaximumNumber" type="xsd:nonNegativeInteger" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>

```

```

<xsd:complexType name="MessageType">
  <xsd:sequence>
    <xsd:element name="PublisherReference" type="xsd:string" minOccurs="0"/>
    <xsd:element name="SubscriptionReference" type="xsd:string" minOccurs="0"/>
    <xsd:element name="MessagePolicy" minOccurs="0" type="tns:RecipientsType"/>
    <xsd:element name="TimeSent" type="xsd:dateTime"/>
    <xsd:element name="Topic" type="xsd:string"/>
    <xsd:element name="MessageContentType" minOccurs="0" type="xsd:string"/>
    <xsd:element name="MessageAttributes" minOccurs="0" type="xsd:string"/>
    <xsd:element name="MessageContent" type="xsd:string" />
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="GetMessagesResponseType">
  <xsd:sequence>
    <xsd:element name="Message" maxOccurs="unbounded" type="tns:MessageType"
minOccurs="0"/>
    <xsd:element name="Fault" type="tns:FaultType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="UnsubscribeType">
  <xsd:sequence>
    <xsd:element name="SubscriberReference" type="xsd:string"/>
    <xsd:element name="SubscriptionReference" type="xsd:string"/>
    <xsd:element name="EffectiveFrom" minOccurs="0" type="xsd:dateTime"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="UnsubscribeResponseType">
  <xsd:sequence>
    <xsd:element name="SubscriptionReference" type="xsd:string"/>
    <xsd:element name="ResponseDetails" type="tns:ResponseType"/>
    <xsd:element name="EffectiveFrom" minOccurs="0" type="xsd:dateTime"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="RegisterTopicType">
  <xsd:sequence>
    <xsd:element name="RegistrarReference" type="xsd:string"/>
    <xsd:element name="TopicName" type="xsd:string"/>
    <xsd:element name="TopicSchema" minOccurs="0" type="xsd:string"/>
    <xsd:element name="TopicPolicy" type="tns:PublishersType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="RegisterTopicResponseType">
  <xsd:sequence>
    <xsd:element name="TopicName" type="xsd:string"/>
    <xsd:element name="ResonseDetails" type="tns:ResponseType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="UnRegisterTopicType">
  <xsd:sequence>
    <xsd:element name="RegistrarReference" type="xsd:string"/>
    <xsd:element name="TopicName" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="UnRegisterTopicResponseType">
  <xsd:sequence>
    <xsd:element name="TopicName" type="xsd:string"/>
    <xsd:element name="ResponseDetails" type="tns:ResponseType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="RegisterPartnerServiceType">
  <xsd:sequence>
    <xsd:element name="PartnerReference" minOccurs="0" type="xsd:string"/>
    <xsd:element name="SubscriberAddress" type="tns:EndPointReferenceType"
minOccurs="0"/>
    <xsd:element name="SecuritySignature" type="xsd:string"/>
    <xsd:element name="OtherDetails" minOccurs="0" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="RegisterPartnerServiceResponseType">
  <xsd:sequence>

```

```

        <xsd:element name="PublisherReference" type="xsd:string" minOccurs="0"/>
        <xsd:element name="SecuritySignature" type="xsd:string" minOccurs="0"/>
        <xsd:element name="Fault" type="tns:FaultType" minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="UnRegisterPartnerServiceType">
    <xsd:sequence>
        <xsd:element name="PartnerReference" type="xsd:string"/>
        <xsd:element name="SecuritySignature" minOccurs="0" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="UnRegisterPartnerServiceResponseType">
    <xsd:sequence>
        <xsd:element name="PartnerReference" type="xsd:string"/>
        <xsd:element name="ResponseDetails" type="tns:ResponseType" minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="QueryType">
    <xsd:sequence>
        <xsd:element name="Description" type="xsd:string"/>
        <xsd:element name="Parameter" maxOccurs="unbounded" minOccurs="0" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="DataQueryType">
    <xsd:sequence>
        <xsd:element name="QueryReference" type="tns:QueryType"/>
        <xsd:element name="ProducerReference" type="xsd:string"/>
        <xsd:element name="ConsumerReference" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="DataQueryResponseType">
    <xsd:sequence>
        <xsd:element name="QueryReference" type="tns:QueryType" minOccurs="0"/>
        <xsd:element name="ConsumerReference" type="xsd:string"/>
        <xsd:element name="DataRequested" type="xsd:string" minOccurs="0"/>
        <xsd:element name="Fault" minOccurs="0" type="tns:FaultType"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="CreatePullPointType">
    <xsd:sequence>
        <xsd:element name="SubscriberReference" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="CreatePullPointResponseType">
    <xsd:sequence>
        <xsd:element name="SubscriberReference" type="xsd:string"/>
        <xsd:element name="ResponseDetails" type="tns:ResponseType" minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="DestroyPullPointType">
    <xsd:sequence>
        <xsd:element name="SubscriberReference" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="DestroyPullPointResponseType">
    <xsd:sequence>
        <xsd:element name="SubscriberReference" type="xsd:string"/>
        <xsd:element name="ResponseDetails" type="tns:ResponseType" minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="EndPointReferenceType">
    <xsd:sequence>
        <xsd:element name="Address" type="xsd:anyURI"/>
        <xsd:element name="ReferenceParameters" type="xsd:string" minOccurs="0" />
        <xsd:element name="Metadata" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Notification">
    <xsd:choice>
        <xsd:element name="Message" type="tns:MessageType"/>
    </xsd:choice>

```

```

    <xsd:element name="Subscribe" type="tns:SubscribeType"/>
    <xsd:element name="UnSubscribe" type="tns:UnsubscribeType" />
    <xsd:element name="PauseSubscription" type="tns:UnsubscribeType" />
    <xsd:element name="RenewSubscription" type="tns:UnsubscribeType" />
    <xsd:element name="GetMessages" type="tns:GetMessagesType"/>
    <xsd:element name="RegisterTopic" type="tns:RegisterTopicType"/>
    <xsd:element name="UnRegisterTopic" type="tns:UnRegisterTopicType"/>
    <xsd:element name="RegisterPartnerService" type="tns:RegisterPartnerServiceType"/>
    <xsd:element name="UnRegisterPartnerService"
type="tns:UnRegisterPartnerServiceType"/>
    <xsd:element name="DataQuery" type="tns:DataQueryType"/>
    <xsd:element name="CreatePullPoint" type="tns:CreatePullPointType"/>
    <xsd:element name="DestroyPullPoint" type="tns:DestroyPullPointType"/>
  </xsd:choice>
  <xsd:attribute name="type" type="xsd:string"/>
  <xsd:attribute name="token" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="NotificationResponse">
  <xsd:choice>
    <xsd:element name="MessageResponse" type="tns:ResponseType"/>
    <xsd:element name="SubscribeResponse" type="tns:SubscribeResponseType"/>
    <xsd:element name="UnSubscribeResponse" type="tns:UnsubscribeResponseType"/>
    <xsd:element name="PauseSubscriptionResponse" type="tns:UnsubscribeResponseType"/>
    <xsd:element name="RenewSubscriptionResponse" type="tns:UnsubscribeResponseType"/>
    <xsd:element name="GetMessagesResponse" type="tns:GetMessagesResponseType" />
    <xsd:element name="RegisterTopicResponse" type="tns:RegisterTopicResponseType"/>
    <xsd:element name="UnRegisterTopicResponse" type="tns:UnRegisterTopicResponseType"/>
    <xsd:element name="RegisterPartnerServiceResponse"
type="tns:RegisterPartnerServiceResponseType"/>
    <xsd:element name="UnRegisterPartnerServiceResponse"
type="tns:UnRegisterPartnerServiceResponseType"/>
    <xsd:element name="DataQueryResponse" type="tns:DataQueryResponseType"/>
    <xsd:element name="CreatePullPointResponse" type="tns:CreatePullPointResponseType"/>
    <xsd:element name="DestroyPullPointResponse"
type="tns:DestroyPullPointResponseType"/>
  </xsd:choice>
  <xsd:attribute name="type" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="sNotify" type="tns:Notification"/>
<xsd:element name="sNotifyResponse" type="tns:NotificationResponse"/>
<xsd:element name="aNotify" type="tns:Notification"/>
<xsd:element name="aNotifyResponse" type="tns:NotificationResponse"/>
</xsd:schema>

```

Policy-based Message Broker's WSDL

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="policybroker"
  targetNamespace="policybroker.site.uottawa.ca"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns="policybroker.site.uottawa.ca" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <types>
    <xsd:schema>
      <xsd:import namespace="policybroker.site.uottawa.ca"
schemaLocation="policybroker.xsd"/>
    </xsd:schema>
  </types>
  <message name="sNotify">
    <part name="parameters" element="ns:sNotify"/>
  </message>
  <message name="sNotifyResponse">
    <part name="parameters" element="ns:sNotifyResponse"/>
  </message>

```

```

<message name="aNotify">
  <part name="parameters" element="ns:aNotify"/>
</message>
<message name="aNotifyResponse">
  <part name="parameters" element="ns:aNotifyResponse"/>
</message>
<portType name="PolicyBroker">
  <operation name="sNotify">
    <input message="ns:sNotify"/>
    <output message="ns:sNotifyResponse"/>
  </operation>
  <operation name="aNotify">
    <input message="ns:aNotify"/>
  </operation>
  <operation name="aNotifyResponse">
    <input message="ns:aNotifyResponse"/>
  </operation>
</portType>
<binding name="PolicyBrokerBinding" type="ns:PolicyBroker">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="sNotify">
    <soap:operation/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="aNotify">
    <soap:operation/>
    <input>
      <soap:body use="literal"/>
    </input>
  </operation>
  <operation name="aNotifyResponse">
    <soap:operation/>
    <input>
      <soap:body use="literal"/>
    </input>
  </operation>
</binding>
<service name="policybrokerService">
  <port name="PolicyBrokerBindingPort" binding="ns:PolicyBrokerBinding">
    <soap:address
location="http://localhost:8080/policybrokerService/PolicyBrokerBindingPort"/>
  </port>
</service>
</definitions>

```

Partner Service Component WSDL

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="policybroker"
  targetNamespace="policybroker.site.uottawa.ca"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns="policybroker.site.uottawa.ca" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <types>
    <xsd:schema>
      <xsd:import namespace="policybroker.site.uottawa.ca"
schemaLocation="policybroker.xsd"/>
    </xsd:schema>
  </types>
  <message name="sNotify">
    <part name="parameters" element="ns:sNotify"/>
  </message>
  <message name="sNotifyResponse">

```

```

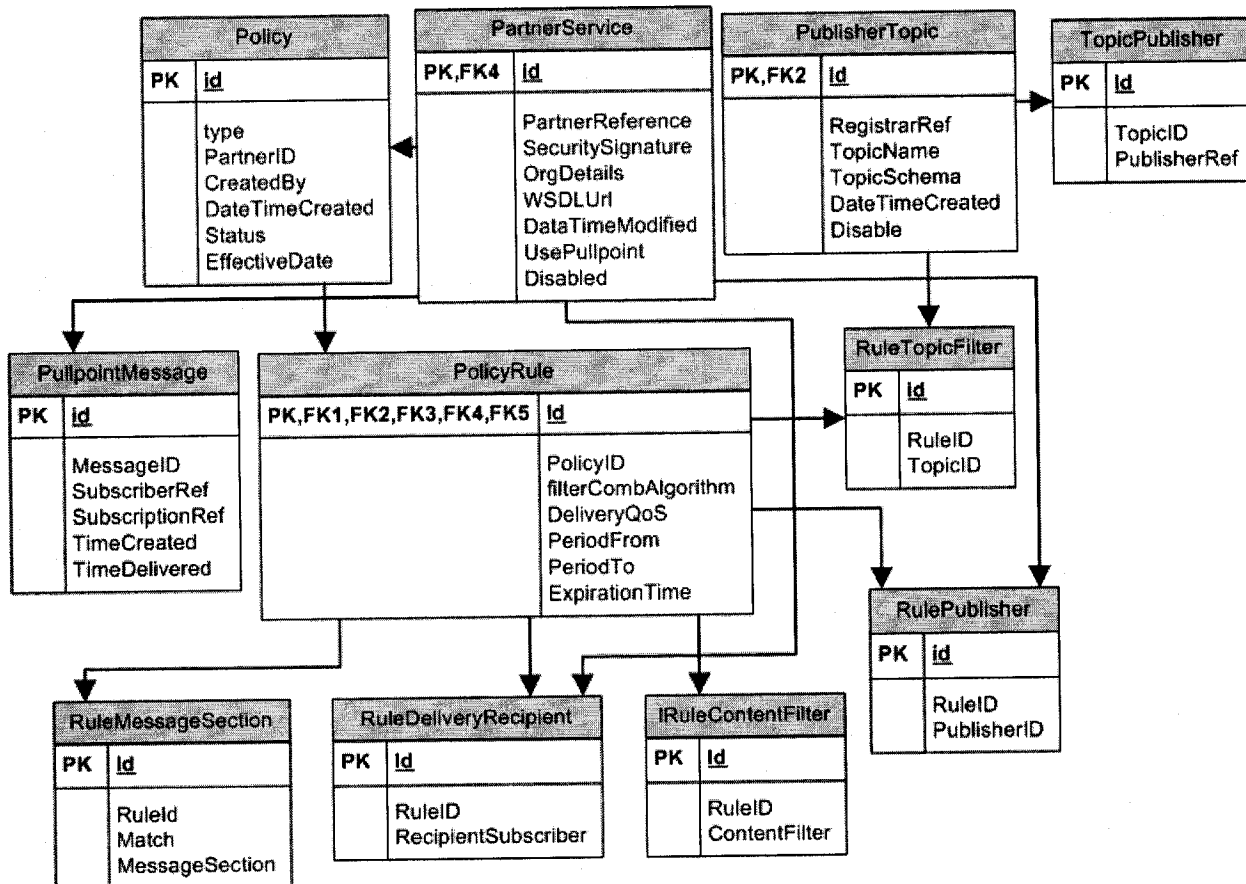
    <part name="parameters" element="ns:sNotifyResponse"/>
</message>
<message name="aNotify">
  <part name="parameters" element="ns:aNotify"/>
</message>
<message name="aNotifyResponse">
  <part name="parameters" element="ns:aNotifyResponse"/>
</message>
<portType name="PSComponent">
  <operation name="sNotify">
    <input message="ns:sNotify"/>
    <output message="ns:sNotifyResponse"/>
  </operation>
  <operation name="aNotify">
    <input message="ns:aNotify"/>
  </operation>
  <operation name="aNotifyResponse">
    <input message="ns:aNotifyResponse"/>
  </operation>
</portType>
<binding name="PSComponentBinding" type="ns:PSComponent">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="sNotify">
    <soap:operation/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="aNotify">
    <soap:operation/>
    <input>
      <soap:body use="literal"/>
    </input>
  </operation>
  <operation name="aNotifyResponse">
    <soap:operation/>
    <input>
      <soap:body use="literal"/>
    </input>
  </operation>
</binding>
<service name="PSComponentService">
  <port name="PSComponentBindingPort" binding="ns:PSComponentBinding">
    <soap:address location="http://localhost:8080
/PSComponentService/PSComponentBindingPort"/>
  </port>
</service>
</definitions>

```

Appendix B –Policy Database

In this appendix, we show the SQL script and the relationship diagram for the Policy database.

Policy Database Relationship diagram



SQL Scripts

```
--
-- Create schema policydb. MySQL Database engine scripts
--

CREATE DATABASE IF NOT EXISTS policydb;
USE policydb;

--
-- Definition of table `tblpartnerservice`
--

DROP TABLE IF EXISTS `tblpartnerservice`;
CREATE TABLE `tblpartnerservice` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `PartnerReference` varchar(100) DEFAULT NULL,
  `SecuritySignature` varchar(100) DEFAULT NULL,
  `OrgDetails` text,
  `WSDLUrl` varchar(200) DEFAULT NULL,
  `DateTimeModified` datetime DEFAULT NULL,
  `UsePullpoint` tinyint(1) DEFAULT NULL,
  `Disabled` tinyint(3) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=latin1;

--
-- Definition of table `tblpolicy`
--

DROP TABLE IF EXISTS `tblpolicy`;
CREATE TABLE `tblpolicy` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `type` int(10) unsigned DEFAULT NULL,
  `PartnerID` int(10) unsigned DEFAULT NULL,
  `CreatedBy` varchar(100) DEFAULT NULL,
  `DateTimeCreated` datetime DEFAULT NULL,
  `Status` int(10) unsigned DEFAULT NULL,
  `EffectiveDate` datetime NOT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_tblPolicy_1` (`PartnerID`),
  CONSTRAINT `FK_tblPolicy_1` FOREIGN KEY (`PartnerID`) REFERENCES
`tblpartnerservice` (`id`) ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Definition of table `tblpolicyrule`
--

DROP TABLE IF EXISTS `tblpolicyrule`;
CREATE TABLE `tblpolicyrule` (
```

```

`id` int(10) unsigned NOT NULL AUTO_INCREMENT,
`PolicyID` int(10) unsigned DEFAULT NULL,
`filterCombiningAlg` varchar(45) DEFAULT NULL,
`DeliveryService Quality` varchar(45) DEFAULT NULL,
`PeriodFrom` datetime DEFAULT NULL,
`PeriodTo` datetime DEFAULT NULL,
`ExpirationTime` datetime DEFAULT NULL,
PRIMARY KEY (`id`),
KEY `FK_tblpolicyrule_1` (`PolicyID`),
CONSTRAINT `FK_tblpolicyrule_1` FOREIGN KEY (`PolicyID`) REFERENCES
`tblpartnerservice` (`id`) ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Definition of table `tblpublishertopic`
--

DROP TABLE IF EXISTS `tblpublishertopic`;
CREATE TABLE `tblpublishertopic` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `RegistraRef` int(10) unsigned DEFAULT NULL,
  `TopicName` varchar(200) DEFAULT NULL,
  `TopicSchema` text,
  `DateTimeModified` datetime DEFAULT NULL,
  `Disabled` tinyint(1) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_tblpublishertopic_1` (`RegistraRef`) USING BTREE,
  CONSTRAINT `FK_tblpublishertopic_1` FOREIGN KEY (`RegistraRef`) REFERENCES
`tblpartnerservice` (`id`) ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Definition of table `tblpullpointmessage`
--

DROP TABLE IF EXISTS `tblpullpointmessage`;
CREATE TABLE `tblpullpointmessage` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `MessageID` varchar(100) DEFAULT NULL,
  `SubscriberRef` int(10) unsigned DEFAULT NULL,
  `SubscriptionRef` int(10) unsigned DEFAULT NULL,
  `TimeCreated` datetime DEFAULT NULL,
  `TimeDelivered` datetime NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Definition of table `tblrulecontentfilter`
--

DROP TABLE IF EXISTS `tblrulecontentfilter`;
CREATE TABLE `tblrulecontentfilter` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `RuleID` int(10) unsigned DEFAULT NULL,
  `ContentFilter` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`),

```

```

    KEY `FK_tblRuleContentFilter_1` (`RuleID`),
    CONSTRAINT `FK_tblRuleContentFilter_1` FOREIGN KEY (`RuleID`) REFERENCES
`tblpolicyrule` (`id`) ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

```

```

--
-- Definition of table `tblruledeliveryrecipient`
--

```

```

DROP TABLE IF EXISTS `tblruledeliveryrecipient`;
CREATE TABLE `tblruledeliveryrecipient` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `RuleID` int(10) unsigned DEFAULT NULL,
  `RecipientSubscriber` int(10) unsigned DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_tblRuleDeliveryRecipient_1` (`RuleID`),
  KEY `FK_tblRuleDeliveryRecipient_2` (`RecipientSubscriber`),
  CONSTRAINT `FK_tblRuleDeliveryRecipient_1` FOREIGN KEY (`RuleID`)
REFERENCES `tblpolicyrule` (`id`) ON UPDATE CASCADE,
  CONSTRAINT `FK_tblRuleDeliveryRecipient_2` FOREIGN KEY
(`RecipientSubscriber`) REFERENCES `tblpartnerservice` (`id`) ON UPDATE
CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

```

```

--
-- Definition of table `tblrulemessagesection`
--

```

```

DROP TABLE IF EXISTS `tblrulemessagesection`;
CREATE TABLE `tblrulemessagesection` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `RuleID` int(10) unsigned DEFAULT NULL,
  `Match` tinyint(1) DEFAULT NULL,
  `MessageSection` varchar(200) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_tblRuleMessageSection_1` (`RuleID`),
  CONSTRAINT `FK_tblRuleMessageSection_1` FOREIGN KEY (`RuleID`) REFERENCES
`tblpolicyrule` (`id`) ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

```

```

--
-- Definition of table `tblrulepublisher`
--

```

```

DROP TABLE IF EXISTS `tblrulepublisher`;
CREATE TABLE `tblrulepublisher` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `RuleID` int(10) unsigned DEFAULT NULL,
  `PublisherID` int(10) unsigned DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_tblrulepublisher_1` (`RuleID`),
  KEY `FK_tblrulepublisher_2` (`PublisherID`),
  CONSTRAINT `FK_tblrulepublisher_1` FOREIGN KEY (`RuleID`) REFERENCES
`tblpolicyrule` (`id`) ON UPDATE CASCADE,
  CONSTRAINT `FK_tblrulepublisher_2` FOREIGN KEY (`PublisherID`) REFERENCES
`tblpartnerservice` (`id`) ON UPDATE CASCADE

```

```
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
--  
-- Definition of table `tblruletopicfilter`  
--
```

```
DROP TABLE IF EXISTS `tblruletopicfilter`;  
CREATE TABLE `tblruletopicfilter` (  
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,  
  `RuleID` int(10) unsigned DEFAULT NULL,  
  `TopicID` int(10) unsigned DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `FK_tblTopicFilter_1` (`RuleID`),  
  KEY `FK_tblTopicFilter_2` (`TopicID`),  
  CONSTRAINT `FK_tblTopicFilter_1` FOREIGN KEY (`RuleID`) REFERENCES  
  `tblpolicyrule` (`id`) ON UPDATE CASCADE,  
  CONSTRAINT `FK_tblTopicFilter_2` FOREIGN KEY (`TopicID`) REFERENCES  
  `tblpublishertopic` (`id`) ON UPDATE CASCADE  
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
--  
-- Definition of table `tbltopicpublisher`  
--
```

```
DROP TABLE IF EXISTS `tbltopicpublisher`;  
CREATE TABLE `tbltopicpublisher` (  
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,  
  `TopicID` int(10) unsigned DEFAULT NULL,  
  `PublisherRef` int(10) unsigned DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `FK_tbTopicPublisher_1` (`TopicID`),  
  KEY `FK_tbTopicPublisher_2` (`PublisherRef`),  
  CONSTRAINT `FK_tbTopicPublisher_1` FOREIGN KEY (`TopicID`) REFERENCES  
  `tblpublishertopic` (`id`) ON UPDATE CASCADE,  
  CONSTRAINT `FK_tbTopicPublisher_2` FOREIGN KEY (`PublisherRef`) REFERENCES  
  `tblpartnerservice` (`id`) ON UPDATE CASCADE  
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```