



Provided by the author(s) and University of Galway in accordance with publisher policies. Please cite the published version when available.

Title	StoRHm: A Protocol Adapter for mapping SOAP-based Web Services to RESTful HTTP Format
Author(s)	Kennedy, Sean
Publication Date	2011-09-29
Item record	http://hdl.handle.net/10379/2621

Downloaded 2024-04-24T05:18:24Z

Some rights reserved. For more information, please see the item record link above.



StoRHm: A Protocol
Adapter for mapping SOAP-based Web Services
to RESTful HTTP Format

Sean Kennedy

PhD Thesis
September 2011

Supervisor : Dr. Owen Molloy
Department of Information Technology
National University of Ireland, Galway
Ireland.

Contents

Contents	ii
List of Figures	vi
List of Tables	ix
List of Program Listings	x
Abstract	xii
Acknowledgements	xiii
Declaration	xiv
Publications	xv
Glossary of Terms	xvii
Chapter 1 Introduction	19
1.1 Motivation	19
1.2 Research Opportunities	22
1.3 Principal Contributions	23
1.4 Thesis Overview	28
Chapter 2 Web Services	30
2.1 Introduction	30
2.1.1 Background	30
2.1.2 Definition	35
2.2 Service Oriented Architecture (SOA) [15] [16]	36
2.2.1 SOA Roles	37
2.3 Web Services Architecture	38
2.3.1 WS-* Interoperability Stack	40
2.4 SOAP	42
2.4.1 SOAP Messaging Model	43
2.4.2 SOAP Headers (vertical extensibility)	46
2.4.3 SOAP Intermediaries (horizontal extensibility)	46
2.4.4 SOAP Processing model	48
2.4.5 SOAP faults (errors)	48
2.4.6 Transport binding	49
2.4.7 Message Exchange Patterns (MEPs)	50
2.4.8 SOAP 1.2	51
2.5 Web Services Description Language (WSDL)	52
2.6 Universal Description, Discovery and Integration (UDDI)	67
2.7 Quality of Service (QoS) in Web Services	68
2.8 Plain Old XML (POX)	76
2.9 HTTP Tunnelling	77
2.10 Web Services Interoperability	77
2.11 Summary	78

Chapter 3	Representational State Transfer (REST)	80
3.1	Introduction	80
3.1.1	Background	80
3.2	REST	83
3.2.1	Definition	84
3.2.2	REST nomenclature	85
3.2.3	Deriving REST	86
3.2.4	Summary	91
3.3	The Web	92
3.3.1	HyperText Transfer Protocol (HTTP)	92
3.3.2	Uniform Resource Identifier (URI)	109
3.3.3	Representations and Media Types	111
3.3.4	Web Caching	112
3.4	Web Service architectures	114
3.5	REST anti-patterns	116
3.5.1	Tunnelling everything through GET	116
3.5.2	Tunnelling everything through POST	117
3.6	Summary	118
Chapter 4	Semantic Web	119
4.1	Introduction	119
4.1.1	“The Web” versus “The Semantic Web”	119
4.1.2	Background	120
4.2	Resource Description Framework (RDF)	121
4.2.1	RDF graphs	121
4.2.2	RDF Identifiers [69]	122
4.2.3	RDF Serialisation	123
4.2.3.1	RDF/XML	124
4.2.3.2	N-Triples	130
4.2.3.3	Turtle (Terse RDF Triple Language)	130
4.2.3.4	Notation 3 (N3)	130
4.3	Adding semantics to RDF data	131
4.3.1	RDFS (Resource Description Framework Schema)	131
4.3.2	Ontologies	136
4.3.3	OWL Web Ontology Language	137
4.3.4	OWL Ontologies [67]	138
4.4	Querying RDF data	140
4.4.1	SPARQL	141
4.5	Semantically annotated Web Services	143
4.5.1	Semantic Web Services	144
4.5.1.1	SAWSDL	145
4.5.2	Semantic annotation of RESTful Web Services	147
4.5.2.1	Microformats/Poshformats	148
4.5.2.2	hRESTS	148
4.5.2.3	MicroWSMO	149
4.5.2.4	GRDDL	154
4.5.2.5	WSMO-Lite	155
4.5.2.6	SA-REST	158
4.5.2.7	RDFa	160
4.6	Summary	162

Chapter 5	StoRHm	163
5.1	Introduction	163
5.2	Architecture overview	163
5.3	Project lifecycle	165
5.4	StoRHm v1	166
5.4.1	StoRHm v1 requirements	166
5.4.2	StoRHm v1 architecture	169
5.4.3	StoRHm v1 Design	170
5.4.3.1	StoRHm v1 (POX) Configuration Wizard	170
5.4.3.2	StoRHm v1 (POX) Protocol Adapter	177
5.4.3.3	StoRHm v1 (SOAP) Configuration Wizard	182
5.4.3.4	StoRHm v1 (SOAP) Protocol Adapter	189
5.5	StoRHm v2	192
5.5.1	StoRHm v2 requirements	192
5.5.2	StoRHm v2 architecture	192
5.5.3	StoRHm v2 Design	193
5.5.3.1	StoRHm v2 Configuration Wizard	193
5.5.3.2	StoRHm v2 Protocol Adapter	200
5.6	Summary	203
Chapter 6	Implementation and Evaluation	204
6.1	Introduction	204
6.2	StoRHm v1 implementation	205
6.2.1	StoRHm v1 (POX) Configuration Wizard	205
6.2.2	StoRHm v1 (POX) Protocol Adapter	208
6.2.3	StoRHm v1 (SOAP) Configuration Wizard	210
6.2.4	StoRHm v1 (SOAP) Protocol Adapter	212
6.4	StoRHm v2 implementation	214
6.4.1	Configuration Wizard	214
6.4.2	Protocol Adapter	217
6.5	Testing	218
6.5.1	Test Environment	219
6.5.2	POX example	220
6.5.3	SOAP example	222
6.6	Evaluation	224
6.6.1	Configuration Wizards	225
6.6.2	Protocol Adapter	227
6.6.2.1	Software artefacts	227
6.6.2.2	LAN performance – StoRHm v2 adapter	229
6.6.2.3	Internet performance – StoRHm v2 adapter	233
6.6.2.4	Message Visibility	238
6.6.3	Ease of Migration	238
6.7	Summary	238
Chapter 7	Conclusion	239
7.1	Thesis Summary	239
7.2	Research Conclusions	240
7.3	Future Work	242

References	244
Appendix	257
Listing 1 – POX Internet-based <i>ServiceView</i> Request	257
Listing 2 – POX Internet-based <i>ServiceAdd</i> Request	258
Listing 3 – POX Internet-based <i>ServiceDelete</i> Request	259
Listing 4 – POX Internet-based <i>ServiceUpdate</i> Request	260
Listing 5 – SOAP Internet-based <i>ServiceView</i> Request	261
Listing 6 – SOAP Internet-based <i>ServiceAdd</i> Request	262
Listing 7 – SOAP Internet-based <i>ServiceDelete</i> Request	263
Listing 8 – SOAP Internet-based <i>ServiceUpdate</i> Request	264
Listing 9 – POX request that involves significant parsing	265
Listing 10 – GET equivalent of Listing 1	269

List of Figures

Figure 1-1	Recent middleware history	20
Figure 1-2	XML Web Services protocols	21
Figure 1-3	Web Infrastructure enabler	23
Figure 1-4	Migration enabler	24
Figure 1-5	SOAP caching	25
Figure 1-6 (a)	Semantic Web Services Interoperability	27
Figure 1-6 (b)	SOAP WS output mapped to RESTful WS input	27
Figure 1-6 (c)	SOAP WS input mapped to RESTful WS input	27
Figure 1-7	Thesis Road Map	28
Figure 2-1	SOA Roles [17]	37
Figure 2-2	Basic Web services architecture	39
Figure 2-3	WS-* Interoperability Stack [16]	39
Figure 2-4	SOAP message path	42
Figure 2-5	SOAP message structure	43
Figure 2-6	Request-Response MEP	50
Figure 2-7	SOAP Response MEP	51
Figure 2-8	Overview of WSDL 1.1 file structure [28]	54
Figure 2-9	<i>service</i> relationship to <i>portType</i> in WSDL [18]	57
Figure 2-10	WSDL 1.1 versus WSDL 2.0 [35]	64
Figure 2-11	WS-RM Design [16]	69
Figure 2-12	Symmetric key encryption	72
Figure 2-13	Asymmetric key encryption	72
Figure 2-14	Digital Signature	73
Figure 2-15	SOAP-REST Web Services interoperability	78
Figure 3-1	REST [5]	84
Figure 3-2	Terminology used in REST	85
Figure 3-3	Deriving REST	87
Figure 3-4	Uniform Interface	89
Figure 3-5	HTTP request message format [53]	93
Figure 3-6	HTTP response message format [53]	94
Figure 3-7	POE example	100
Figure 3-8	HTTPLR example	101
Figure 3-9	Message Reliability [based on 58]	102
Figure 3-10	In-order message delivery	104
Figure 3-11	URI syntax structure	109
Figure 3-12	HTTP POST Tunnelling	118
Figure 4-1	RDF graph of Listing 4-1	122
Figure 4-2	RDF graph	123
Figure 4-3	RDF graph with a blank node	126
Figure 4-4	RDF graph with the <i>age</i> property untyped	128

Figure 4-5	RDF graph with the <i>age</i> property typed	128
Figure 4-6	N-Triples, Turtle and N3 relationships	131
Figure 4-7	UML Diagram of the hierarchy	133
Figure 4-8	Domain and Range of a property expressing eye colour [73]	136
Figure 4-9	Semantic Layers [80]	144
Figure 4-10	SAWSDL annotation using Core Dashboard	146
Figure 4-11	RESTful Hotel WS description	150
Figure 4-12(a)	SWEET: hRESTS annotation	150
Figure 4-12(b)	SWEET: MicroWSMO annotation	151
Figure 5-1 (a)	XML WS Architecture	164
Figure 5-1 (b)	StoRHm Architecture	164
Figure 5-2	Timeline of Architectures	165
Figure 5-3	Relationship of the two versions	166
Figure 5-4	StoRHm Use Case Diagram	167
Figure 5-5	XML Web Service to RESTful Interface Mapping [26]	167
Figure 5-6	StoRHm v1 Architecture	170
Figure 5-7	StoRHm v1 (POX) Configuration Wizard GUI Layout	171
Figure 5-8	StoRHm v1 (POX) GUI Layout Managers	172
Figure 5-9	Sample StoRHm v1 (POX) Configuration Wizard	172
Figure 5-10 (a)	StoRHm v1 (POX) Configuration Wizard sequence diagram (with WADL)	173
Figure 5-10 (b)	StoRHm v1 (POX) Configuration Wizard sequence diagram (without WADL)	174
Figure 5-11	StoRHm v1 (POX) Configuration Wizard package diagram	175
Figure 5-12	StoRHm v1 (POX) Configuration Wizard class Diagram	176
Figure 5-13	StoRHm v1 (POX) CSV file	177
Figure 5-14	StoRHm v1 (POX) Protocol Adapter design	177
Figure 5-15	StoRHm v1 (POX) Protocol Adapter sequence diagram	178
Figure 5-16	StoRHm v1 (POX) Protocol Adapter package diagram	181
Figure 5-17	StoRHm v1 (POX) Protocol Adapter class diagram	181
Figure 5-18	StoRHm v1 (SOAP) Configuration Wizard GUI Layout	182
Figure 5-19	StoRHm v1 (SOAP) GUI Layout Managers	183
Figure 5-20	Sample StoRHm v1 (SOAP) Configuration Wizard	184
Figure 5-21	StoRHm v1 (SOAP) Configuration Wizard sequence diagram (with WADL)	185
Figure 5-22	StoRHm v1 (SOAP) Configuration Wizard sequence diagram (without WADL)	186
Figure 5-23	StoRHm v1 (SOAP) Configuration Wizard package diagram	187
Figure 5-24	StoRHm v1 (SOAP) Configuration Wizard class diagram	188
Figure 5-25	StoRHm v1 (SOAP) CSV file	189
Figure 5-26	StoRHm v1 (SOAP) Protocol Adapter design	189
Figure 5-27	StoRHm v1 (SOAP) Protocol Adapter sequence diagram	190
Figure 5-28	StoRHm v1 (SOAP) Protocol Adapter package diagram	191
Figure 5-29	StoRHm v1 (SOAP) Protocol Adapter class diagram	192
Figure 5-30	StoRHm v2 Architecture	193
Figure 5-31	StoRHm v2 Configuration Wizard GUI Layout	194
Figure 5-32	StoRHm v2 GUI Layout Managers	194
Figure 5-33 (a)	StoRHm v2 Configuration Wizard (POX)	195
Figure 5-33 (b)	StoRHm v2 Configuration Wizard (SOAP)	195
Figure 5-34 (a)	StoRHm v2 (POX) Configuration Wizard Sequence Diagram	196
Figure 5-34 (b)	StoRHm v2 (SOAP) Configuration Wizard Sequence Diagram	196
Figure 5-35	StoRHm v2 Configuration Wizard flow diagram	197
Figure 5-36	StoRHm v2 Configuration Wizard package diagram	199
Figure 5-37	StoRHm v2 Configuration Wizard class diagram	199
Figure 5-38	StoRHm v2 Protocol Adapter design	200
Figure 5-39 (a)	StoRHm v2 Protocol Adapter (SOAP) sequence diagram	201
Figure 5-39 (b)	StoRHm v2 Protocol Adapter (POX) sequence diagram	201
Figure 5-40	StoRHm v2 Protocol Adapter package diagram	202
Figure 5-41	StoRHm v2 Protocol Adapter class diagram	202
Figure 6-1	POX to RESTful HTTP Configuration Wizard	205
Figure 6-2	Configuration Wizard with WADL selected	208
Figure 6-3	Adapter mapping a POX message to RESTful format	210
Figure 6-4	StoRHm v1 Configuration Wizard	211
Figure 6-5	Building the URI Wizard	212
Figure 6-6	Adapter mapping a SOAP message to RESTful format	213
Figure 6-7 (a)	StoRHm v2 Configuration Wizard - SOAP	217

Figure 6-7 (b)	StoRHm v2 Configuration Wizard - POX	217
Figure 6-8 (a)	LAN-based Test environment	219
Figure 6-8 (b)	Internet-based Test environment	219
Figure 6-9	POX to RESTful HTTP Transformation.....	221
Figure 6-10	RESTful response.....	222
Figure 6-11	SOAP to RESTful HTTP Transformation	223
Figure 6-12	RESTful HTTP to SOAP Transformation	224
Figure 6-13	LAN: GET versus POST performance.....	230
Figure 6-14	LAN: <i>ServiceView</i> performance	231
Figure 6-15	LAN: <i>ServiceAdd</i> performance	232

List of Tables

Table 2-1	SOAP 1.2 MEPs [24]	52
Table 3-1	Key properties induced by REST	92
Table 3-2	HTTP Response codes	95
Table 3-3	Safety/Idempotence of HTTP methods	98
Table 3-4	Security and Message Reliability in HTTP	99
Table 3-5	Criteria for evaluating Web Service Architectures	114
Table 4-1	SPARQL SELECT (results)	142
Table 4-2	Existing XHTML attributes used by RDFa	160
Table 4-3	New XHTML RDFa attributes	161
Table 5-1	Specific Interface mapped to Uniform Interface	168
Table 6-1	Software Artefacts	204
Table 6-2 (a)	LAN Performance Tests – GET versus POST	232
Table 6-2 (b)	LAN Performance Tests – <i>ServiceView</i> versus <i>ServiceAdd</i>	232
Table 6-3	POX Internet Performance Tests	233
Table 6-4	SOAP Internet Performance Tests	235
Table 6-5	Adapter statistics when significant Parsing required	237

List of Program Listings

Listing 2-1	Sample SOAP request message	44
Listing 2-2	RPC/Literal SOAP message for <i>myMethod</i> [21]	45
Listing 2-3	Document/Literal SOAP message for <i>myMethod</i> [21]	45
Listing 2-4	Wrapped Document/Literal SOAP message for <i>myMethod</i> [21]	46
Listing 2-5	SOAP Response	46
Listing 2-6	SOAP Fault Message [16]	49
Listing 2-7	Sample WSDL 1.1 file	58
Listing 2-8	One-way MEP	60
Listing 2-9	Request-response MEP	60
Listing 2-10	Notification MEP	61
Listing 2-11	Solicit-response MEP	61
Listing 2-12	WS-Policy example [31]	63
Listing 2-13	Sample WSDL 2.0 file	66
Listing 2-14	Sample WSRM client message [18]	70
Listing 2-15	Sample WSRM server response [18]	70
Listing 2-16	Sample WS-Security document	74
Listing 2-17	Sample POX file [157]	76
Listing 3-1	HATEOAS example [52]	91
Listing 3-2	A sample HTTP request message	93
Listing 3-3	A sample HTTP response message	95
Listing 3-4	GET on a protected resource	105
Listing 3-5	401 Unauthorized access message	105
Listing 3-6	Authorization header	106
Listing 3-7	Digest authentication challenge	106
Listing 3-8	Sample client request using Digest Authentication	108
Listing 3-9	Sample URI	110
Listing 3-10	RESTful request	115
Listing 3-11	RPC-style request	115
Listing 3-12	RPC_REST request [45]	116
Listing 3-13	GET anti-pattern example	117
Listing 3-14	Accidentally RESTful GET anti-pattern example	117
Listing 4-1	Information about the authors	122
Listing 4-2	The graph in Fig. 4-2 in triple notation format	124
Listing 4-3	The general form of a statement in RDF/XML [67]	124
Listing 4-4	RDF/XML of graph in Fig. 4-2	125
Listing 4-5	RDF/XML of graph in Fig. 4-3	127
Listing 4-6	RDF/XML of graph in Fig. 4-5	129
Listing 4-7	Collated RDF/XML of examples so far	129
Listing 4-8	N-Triple notation of Listing 4-7	130
Listing 4-9	N3 notation of Listing 4-7	131
Listing 4-10	Turtle/N3 serialisation of hierarchy [69]	133
Listing 4-11	RDF/XML format of hierarchy [69]	134
Listing 4-12	Truck resource segment from Listing 4-11	135
Listing 4-13	Alternative RDF/XML syntax for Truck resource	135
Listing 4-14	Domain and Range example [73]	136
Listing 4-15	DatatypeProperty and ObjectProperty example	140
Listing 4-16 (a)	SPARQL SELECT (data)	141

Listing 4-16 (b)	SPARQL SELECT (syntax)	142
Listing 4-17 (a)	SPARQL CONSTRUCT (data).....	142
Listing 4-17 (b)	SPARQL CONSTRUCT (syntax).....	142
Listing 4-17 (c)	SPARQL CONSTRUCT (results).....	143
Listing 4-18 (a)	SPARQL ASK (data).....	143
Listing 4-18 (b)	SPARQL ASK (syntax).....	143
Listing 4-19	SAWSDL annotation using Core Dashboard [81].....	147
Listing 4-20	hRESTS and MicroWSMO annotated Hotel WS (HTML).....	152
Listing 4-21	hRESTS and MicroWSMO annotated Hotel WS (RDF/XML).....	153
Listing 4-22	XHTML section annotated with GRDDL metadata	154
Listing 4-23	RDF produced from the GRDDL annotated Listing 4-20	155
Listing 4-24	WSMO-Lite Ontology (Notation 3)	157
Listing 4-25	WSMO-Lite Example	157
Listing 4-26 (a)	SAWSDL Information model references.....	158
Listing 4-26 (a)	SAWSDL Functional semantics reference.....	158
Listing 4-26 (c)	SAWSDL Non-functional semantics reference	158
Listing 4-27	<i>domain-rel</i> property example	159
Listing 4-28	<i>sem-rel</i> property example	159
Listing 4-29	<i>sem-class</i> property example.....	159
Listing 4-30	XHTML annotated with RDFa.....	161
Listing 4-31	RDF extracted from Listing 4-30 (N3 notation)	162
Listing 6-1	Sample WADL file	207
Listing 6-2	SAWSDL example (segment)	214
Listing 6-3	MicroWSMO example (segment)	215
Listing 6-4	Ontology example (segment).....	216
Listing 6-5	StoRHm_Client.xml.....	228
Listing 6-6 (a)	POX ServiceView.xml.....	229
Listing 6-6 (b)	POX ServiceAdd.xml	229
Listing 6-6 (c)	Test data returned.....	229

Abstract

Two of the most popular approaches to Web Services' implementation are SOAP-based Web Services, which are based on an XML messaging protocol and RESTful HTTP Web Services, which use HTTP in line with Representational State Transfer (REST) principles. RESTful HTTP Web services adhere closely to REST principles and therefore avail of existing sophisticated features of Web infrastructure. SOAP on the other hand, uses HTTP purely as a transport for its messages. This is known as "SOAP tunnelling" and is against REST principles because it disables Web intermediaries, as they are unable to inspect the SOAP message.

The research documented in this thesis proposes a protocol adapter named StoRHm (Soap to Restful HTTP mapping) that, within certain constraints, addresses SOAP tunnelling by seamlessly transforming opaque SOAP messages into visible RESTful HTTP format. The previously opaque messages are now visible to Web intermediaries, thereby enabling advanced Web features such as caching. The adapter described in this research enables SOAP clients to interact seamlessly with back-end RESTful Web services. Thus, StoRHm is also a technology migration enabler i.e. an enterprise can migrate from SOAP-based Web Services to RESTful HTTP Web services, in a gradual fashion, by adopting the adapter. StoRHm proposes to leverage the Semantic Web to automate the mapping element of its solution.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Owen Molloy, for his advice and guidance throughout the process.

Secondly, I would also like to thank my colleagues at Athlone Institute of Technology (AIT) for their support and advice. They are Mary Giblin, Frank Doheny, Dr. Robert Stewart and Dr. Paul Jacob.

Lastly, I want to thank my wife Maria for her patience and understanding.

Declaration

I hereby declare that the work presented in this thesis is my own and that a degree in this university or elsewhere has not been obtained with it.

Signed,

Sean Kennedy

Publications

Journal Publications

Kennedy S, Molloy O, Stewart R, Jacob P " StoRHm: A protocol adapter for mapping SOAP based Web Services to RESTful HTTP format ", published in the Electronic Commerce Research Journal, Volume 11, Issue 3, pp245-269, 2011, available at <http://dx.doi.org/10.1007/s10660-011-9075-3>.

Conference Publications

Kennedy S, Molloy O, Stewart R, Jacob P, Daglioglu G, "StoRHm: a semantically automated protocol adapter for mapping SOAP Web Services to RESTful HTTP format", in the 4th International Conference on Internet Technologies and Applications, Wrexham, Wales, 2011.

Kennedy S, Molloy O, Stewart R, "Leveraging the Semantic Web to automate the mapping of SOAP Web Services to RESTful HTTP format", in the 6th International Conference in Networking and Electronic Commerce Research, Lake Garda, Italy, 2010.

Kennedy S, Molloy O, Stewart R, "An adapter for mapping Plain Old XML (POX) Web Services to RESTful HTTP", International Conference on e-Society, Porto, Portugal, 2010.

Kennedy S, Molloy O, Stewart R and Jacob P, "StoRHm: A novel framework for RESTifying SOAP based Web Services", in the 5th International Conference in Networking and Electronic Commerce Research, Lake Garda, Italy, 2009.

Kennedy S and Molloy O, "A novel approach to mapping Web Services to REST", in the 3rd International Conference on Internet Technologies and Applications, Wrexham, Wales, 2009.

Kennedy S and Molloy O, "A framework for transitioning Enterprise Web Services from XML-RPC to REST", International Conference Information Resources Management, Al Ain, United Arab Emirates, 2009.

Kennedy S, Molloy O and Richardson I, "eCommerce Web Services : REST or WS-*", in the 4th International Conference in Networking and Electronic Commerce Research, Lake Garda, Italy, 2008.

Glossary of Terms

ADSL	Asymmetric Digital Subscriber Line
BPEL4WS	Business Process Execution Language for Web Services
CA	Certificate Authority
CICS	Customer Information Control System
CORBA	Common Object Request Broker Architecture
DCE	Distributed Component Environment
DCOM	Distributed Component Object Model
DTD	Document Type Definition
FOAF	Friend Of A Friend
GUI	Graphical User Interface
GRDDL	Gleaning Resource Descriptions from Dialects of Languages
HATEOAS	Hypermedia As The Engine Of Application State
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IANA	Internet Assigned Numbers Authority
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IIOP	Internet Inter-ORB Protocol
ISBN	International Standard Book Number
Java EE	Java Platform, Enterprise Edition
JMS	Java Message Service
JSON	JavaScript Object Notation
MD5	Message-Digest algorithm 5
MEP	Message Exchange Pattern
MQ	Message Queue
MSS	Mashup Services System
OLTP	Online Transaction Processing
OASIS	Organization for the Advancement of Structured Information Standards
OMG	Object Management Group
ORB	Object Request Broker
PDF	Portable Document Format
POSH	Plain Old Semantic HTML
QoS	Quality of Service
RDF	Resource Description Framework
RDFa	Resource Description Framework in Attributes
RDFS	Resource Description Framework Schema
REST	Representational State Transfer
ROA	Resource Oriented Architecture
RPC	Remote Procedure Call
RSS	Rich Site Summary
SMTP	Simple Mail Transfer Protocol
SA-REST	Semantic Annotation of Web Resources
SOA	Service Oriented Architecture
SOAP	SOAP (no longer an acronym, used to stand for Simple Object Access Protocol)

SPARQL	SPARQL Protocol and RDF Query Language
SSL	Secure Sockets Layer
StoRHm	<u>S</u> OAP to <u>R</u> ESTful <u>H</u> TTP <u>m</u> apping
TLS	Transport Layer Security
Turtle	Terse RDF Triple Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
UDDI	Universal Description Discovery and Integration
W3C	World Wide Web Consortium
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Services Description Language
WS-I	Web Services Interoperability Organisation
XHTML	eXtensible HyperText Markup Language
XML	eXtensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

Chapter 1

Introduction

1.1 Motivation

Since the very first centralised mainframes, computer systems have evolved over time, into decentralised distributed systems. These decentralised systems have become increasingly heterogeneous due to the continuous change in technology and the fact that creating and managing decentralised systems is, in itself decentralised. This is obviously true on the Internet, given its immense size, but also true in enterprises with a smaller machine base. Add factors such as mergers, takeovers and e-business and the heterogeneity in the overall system rises sharply.

Due to the increase in heterogeneous systems and their distributed nature, distributed application integration, under the umbrella term “middleware”, has been an area of research for many years. Middleware refers to “*software that resides between an application and the operating system, network or database underneath it*” [1]. Middleware systems can trace their origins back to Online

Transaction Processing (OLTP) systems such as IBM's Customer Information Control System (CICS) system. From these origins, other middleware systems have developed, such as: IBM's Websphere Message Queue (MQ), Common Object Request Broker Architecture (CORBA), Distributed Component Environment (DCE), Microsoft's Distributed Component Object Model (DCOM), Java Platform, Enterprise Edition (Java EE) and more recently, Web Services. Figure 1-1 gives an overview of the more recent middleware solutions.

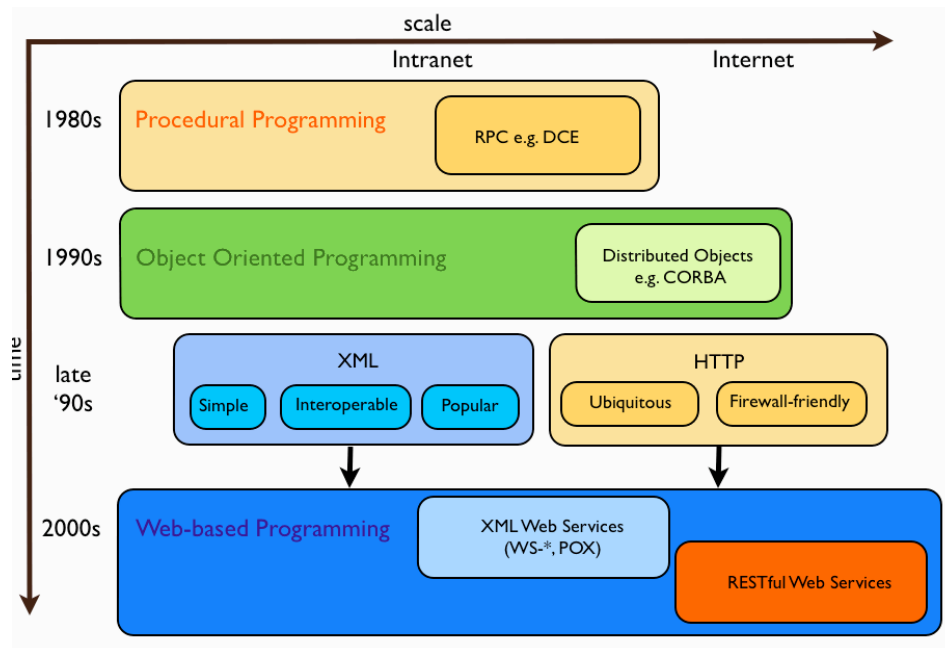


Figure 1-1 Recent middleware history

The explosion of the Internet in the 1990's ensured that HTTP (HyperText Transfer Protocol) was ubiquitous and ushered in the era of e-commerce. The advent of e-commerce resulted in Web browsers needing access to enterprise Web servers residing behind the corporate firewall, resulting in ports 80 and 443 (HTTP/HTTPS respectively) being opened up. The surge in e-commerce also led to the development and widespread adoption of XML (eXtensible Markup Language) as an information exchange representation. XML is a simple, text-based "meta-language" i.e. it allows users to invent their own languages. Due to its text-based nature, XML is both language-independent and platform-independent, making XML highly interoperable.

Distributed object middleware solutions such as CORBA were ideally suited to corporate intranets [2]. However, CORBA ports were not opened on the firewall by administrators leading to firewall traversal issues [119]. This led to CORBA's HTTP adapter e.g. VisiBroker GateKeeper, which is used "to tunnel the HTTP requests" [120]. In addition, the scale of the Internet created a gap in the market place for a middleware solution that would provide a worldwide interoperable distributed computing solution. The widespread deployment of firewall-friendly HTTP, coupled with the simple, interoperable XML format heralded the invention of Web Services [133]. "It is the gap between what

the Web provides (HTTP and XML) and what application integration requires that Web Services are trying to fill” [117].

A major part of the research in this thesis relates to XML Web Services. There are two types of XML Web Services: WS-* and POX (Plain Old XML). Both are used in the area of application integration in a heterogeneous distributed environment. WS-* Web Services are a suite of standardised, XML-based technologies organised conceptually into a stack model. WS-* Web Services use SOAP as its message format and WSDL (Web Services Description Language) for describing the Web Services interface, protocol and location to potential consumers. (Note: SOAP is no longer an acronym. SOAP used to stand for Simple Object Access Protocol but since SOAP was neither simple nor object based, the acronym was dropped). WS-* Web Services are “protocol agnostic” i.e. they can be accessed over any transport or application protocol e.g. HTTP, TCP, Simple Mail Transfer Protocol (SMTP) and Java Message Service (JMS) [2]. However, HTTP is the default transport. POX Web Services on the other hand, are a more lightweight approach in that there is no stack of standards and the de facto protocol used is HTTP. As POX Web Services have no standard for message description or format, enterprises use proprietary XML Schema definitions to describe their interfaces and messages. Fig 1-2 contrasts the protocol used by POX Web Services with the protocol agnostic nature of WS-* Web Services.

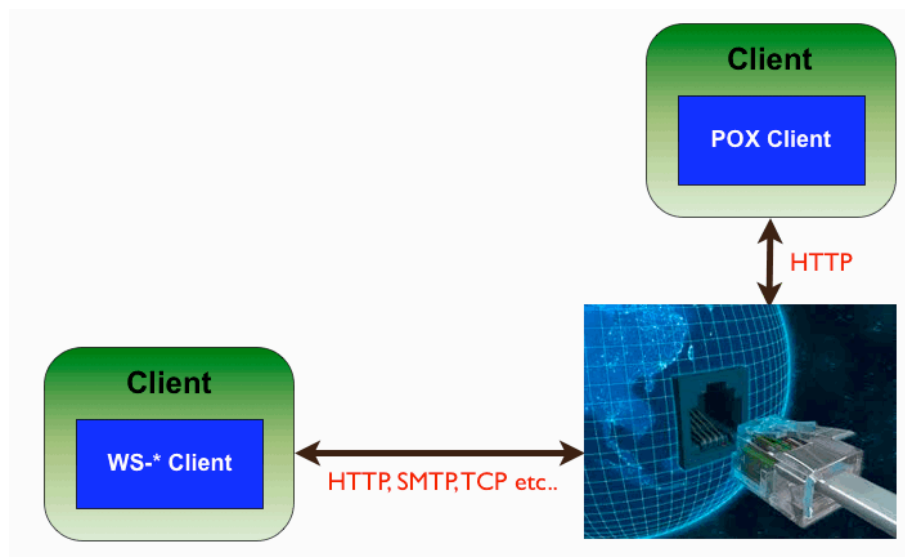


Figure 1-2 XML Web Services protocols

Due to its ubiquity and firewall-friendly nature, HTTP is widely used to transport both WS-* and POX Web Service messages. HTTP is an application protocol following the principles of Representational State Transfer (REST), an architectural style defined by Roy Fielding in his PhD thesis “*Architectural Styles and the Design of Network-based Software Architectures*” [3]. The research presented in this thesis explores the issues of WS-* and POX Web Services’ use of HTTP.

The modern Web is primarily intended for human use. The Semantic Web on the other hand, annotates Web content so that machines can make sense of it. One of the benefits of the Semantic Web is the intelligence it lends to software solutions i.e. the Semantic Web facilitates automation.

An architecture that addresses the issues of WS-* and POX Web Services' use of HTTP is presented and discussed in later chapters. The architecture leverages the Semantic Web to provide intelligence/automation.

1.2 Research Opportunities

During the extensive review of the work carried out in XML Web Services, a number of shortcomings in the body of research have become evident. The following is a list of these, as yet unanswered questions:

- XML Web Services do not follow the REST (REpresentational State Transfer) style when using HTTP (an instantiation of REST principles) to transport its' messages. This can be attributed to the fact that XML Web Services use HTTP, as a transport protocol, when HTTP is in fact, an application protocol [4] [131]. XML Web Services operations are encapsulated in XML and are therefore opaque to Web intermediaries, such as caches and proxies [5]. XML Web Services interact remotely *"through the Web but remain "outside" of the Web"* [101]. This is referred to as 'HTTP tunnelling'. The question arises: is it possible to design an architecture to address "POST Tunnelling"?
- Technology changes are a common occurrence within the enterprise. For example, a merger or acquisition can lead to a change in technology: *"during a merger, there is often a winner and loser; and often for simplicity, one technology will lose out"* [6]. An enterprise wishing to migrate from XML Web Services to RESTful Web Services is faced with the prospect of a wholesale replacement of one technology with another. This approach causes concern for enterprises, as *"they prefer approaches that allow them to gradually shift from one technology to another"* [7]. This results in the question: can an architecture be designed to enable this gradual migration?
- SOAP to RESTful Web Services protocol adapters in literature are server based and do not take into account the current SOAP interface [8]. Thus, the SOAP request messages are still opaque to Web intermediaries and to avail of this type of adapter requires a re-compilation of the client software. In addition, these adapters do not avail of the Semantic Web to add intelligence to the mappings. This leads to the following questions:

- can messages be immediately visible to all intermediaries?
- is it possible to design an architecture that has minimal impact on the client?
- can the architecture avail of Semantic Web intelligence?

1.3 Principal Contributions

From the research questions raised in the previous section, this section contains a number of contributions made by the research presented in this thesis. These contributions are listed below and are discussed at various stages throughout the thesis.

Web Infrastructure enabler

This research proposes an approach, which transforms the opaque XML Web Services messages into visible RESTful HTTP format. As a consequence, Web intermediaries such as caches and proxies are enabled as they can now inspect the visible messages. In essence, the proposed research aims to avail of the existing, proven Web infrastructure capability by enabling it. Figure 1-3 demonstrates this.

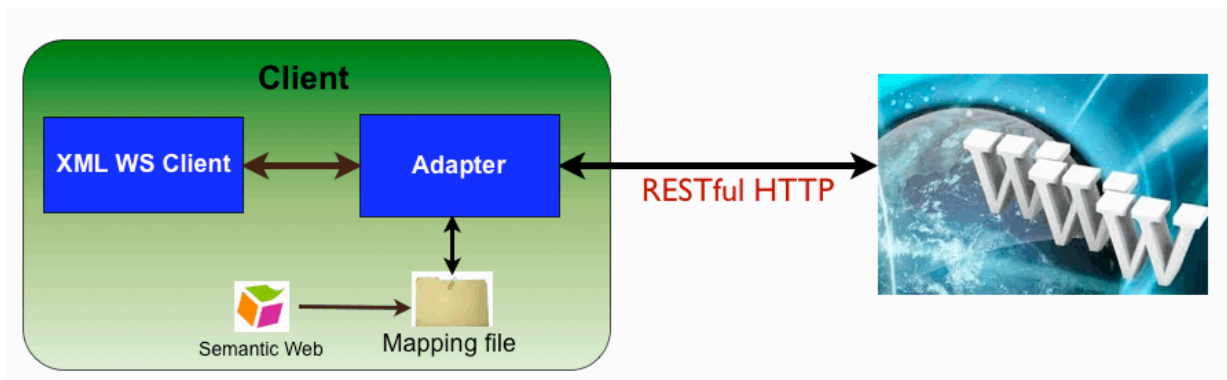


Figure 1-3 Web Infrastructure enabler

The proposed solution provides, via mappings, the ability to transform the opaque Web Service message to visible RESTful HTTP format. These mappings are set up a priori and are dynamically referenced before the message ever leaves the client. Consequently, all messages are RESTful on leaving the client, thereby enabling the Web infrastructure. As a result, there is no need for developers to provide their own custom caching mechanisms.

Migration Enabler

The research proposals presented in this thesis include an approach that enables enterprises to gradually migrate from XML Web Services to RESTful HTTP Web Services. The research presents a service-specific gateway that maintains the SOAP/POX interface thereby enabling the back-end servers to migrate to RESTful HTTP Web Services. This allows the server to migrate “*without breaking existing clients, allowing those clients to migrate when convenient*” [7]. Figure 1-4 illustrates this.

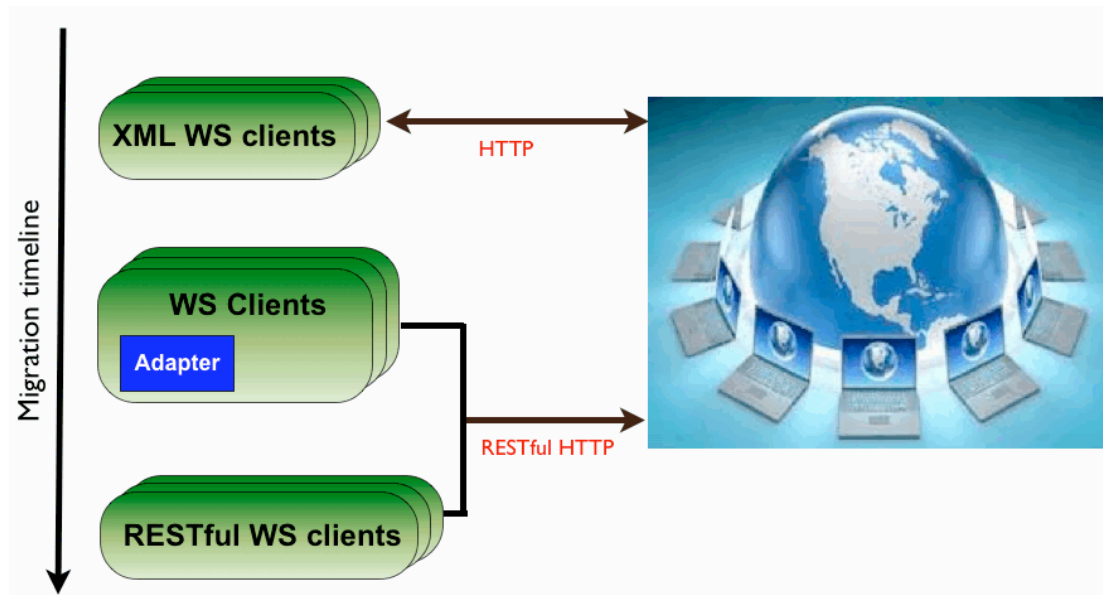


Figure 1-4 Migration enabler

Novel XML Web Services caching approach

As far back as 2003 people have examined the caching issue with XML Web Services: “*XML Web services present new challenges ..., as well as their lack of involvement in the caching process*” [9]. Microsoft Research implemented a client-side SOAP cache to mimic continued access to XML Web services from mobile devices during disconnections [9]. This cache was implemented as a data store. Microsoft’s findings are interesting. Firstly, they comment on how REST’s Uniform Interface principle simplifies the caching issue - “*Web-browser caches, map URIs to HTML pages and need worry about only one operation – namely, the HTTP GET operation*”. Secondly, they note that the specific-interface nature of SOAP increases the complexity of the solution - “*The diverse nature of Web services poses a major problem in identifying the semantics of the operations exposed by the Web service*”.

Andersen et al note that: “*SOAP, being based on XML, inherits not only the advantages, but its relatively poor performance*” [10]. Andersen et al outline an approach to improve SOAP performance by implementing a server-side SOAP cache [10]. Given a client request, the server side cache is checked using the client request as the key; if an entry exists, the associated response object, serialized in XML, is returned. Otherwise the request is satisfied using normal business logic and the client request and its associated response object are stored in the cache (for the benefit of subsequent, identical client requests). Figure 1-5 represents both approaches to SOAP caching outlined above:

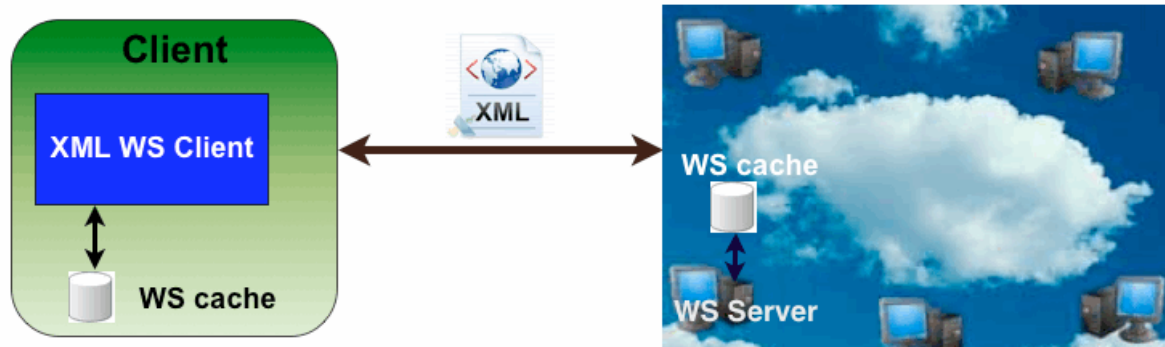


Figure 1-5 SOAP caching

Our proposal to use REST principles addresses the caching issues outlined in [9][10]. The Uniform Interface constraint of REST is particularly relevant and has major advantages over the specific interface approach of SOAP in the area of visibility into system interactions [11]. For example, a cache can understand a HTTP GET request by simply inspecting the URI (Uniform Resource Identifier) in the message to determine the resource being accessed. This is not possible with the HTTP POST requests used by XML Web Services because the actual operation to be performed (whose semantics are only known to the SOAP receiver) is located inside the XML payload document [5]. Even if the cache were programmed to understand a particular SOAP vocabulary, this would not scale; as every new SOAP application, each with its own new vocabulary would require the cache to be modified [5].

Vinoski states that the Uniform Interface constraint makes it easier for developers “*to apply critical distributed systems concepts such as proxying, caching...*” [12]. Vinoski also states that systems conforming to the Uniform Interface constraint gain the following advantages with regard to caching: “*developers can easily insert caches...without breaking the client or needing to specialize the caches*” [11].

So rather than implementing a client or server side SOAP cache (as in Figure 1-5), we propose to seamlessly migrate SOAP messages, with its specialized interface, to RESTful HTTP, with its Uniform Interface (as in Figure 1-3). The messages will conform to the Uniform Interface constraint,

be visible to Web intermediaries (e.g. caches) and enable us to automatically avail of the benefits outlined in [11] and [12]. In essence, we are going to leverage the Web's proven caching architecture.

Novel Web Services Interoperability mapping

In similar research, Briggs outlines a design, which, from a high level, is similar in appearance to the design proposed in this thesis (a RESTful back end with a SOAP front end) [8]. There are several distinct differences to the model outlined in this thesis:

- Briggs is SOAP enabling RESTful Web Services by wrapping them with a SOAP interface. In contrast, the approach outlined in this thesis is REST(ful HTTP) enabling SOAP clients.
- Briggs' adapter is on the server whereas the adapter presented in this thesis resides on the client. Thus, the on-the-wire client messages by Briggs are opaque and tunneled whereas our messages are visible. As outlined earlier this has ramifications for Web intermediaries.
- Briggs does not take the existing SOAP interface into account. In Briggs solution, there is no intention to migrate existing clients from SOAP and new clients are SOAP based. The adapter outlined in this thesis is exactly the opposite: the existing SOAP interface *is* taken into account leading to a seamless migration tool for transitioning SOAP clients to RESTful HTTP format. In addition, new clients are RESTful HTTP based.

Novel use of Semantic Web Services' stacks

The research proposals in this thesis include an automated, intelligent mapping approach. This is achieved via the use of Semantic Web Services technologies. These technologies identify their concepts using URI references and consequently enable matching of SOAP Web Services with their equivalent RESTful HTTP counterparts.

Semantic Web Service technology stacks exist for both SOAP and RESTful HTTP-based services. These stacks support composability, integration and mediation within and across the stacks (see Figure 1-6 (a)). The novelty in this thesis is the *use* of the stacks. These stacks were invented for mediating and integration within and across the SOAP and RESTful HTTP paradigms. For example, Figure 1-6 (b) demonstrates the mapping of a SOAP WS output up to the Semantic layer so that the output can then be mapped to the input format required by the RESTful WS. The complete replacement of the SOAP message with its RESTful HTTP counterpart was not envisaged when the stacks were created. Rather than mediate an output response from a SOAP Web Service to the input

of a RESTful HTTP Web Service, we are mediating the SOAP Web Service *input* to the RESTful HTTP Web Service input (see Figure 1-6 (c)).

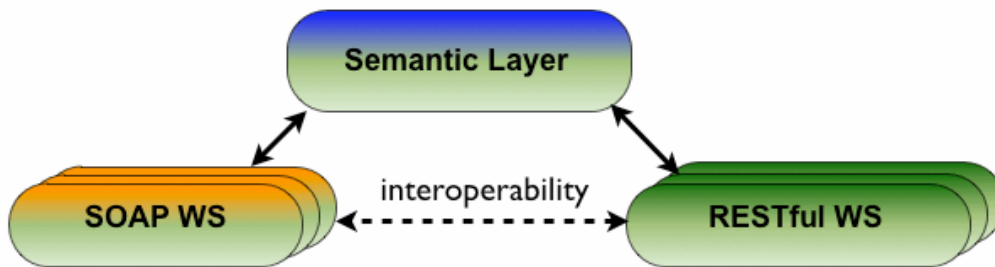


Figure 1-6 (a) Semantic Web Services Interoperability

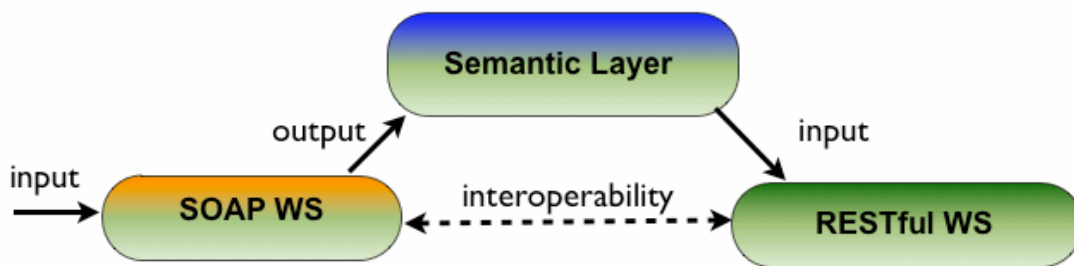


Figure 1-6 (b) SOAP WS output mapped to RESTful WS input

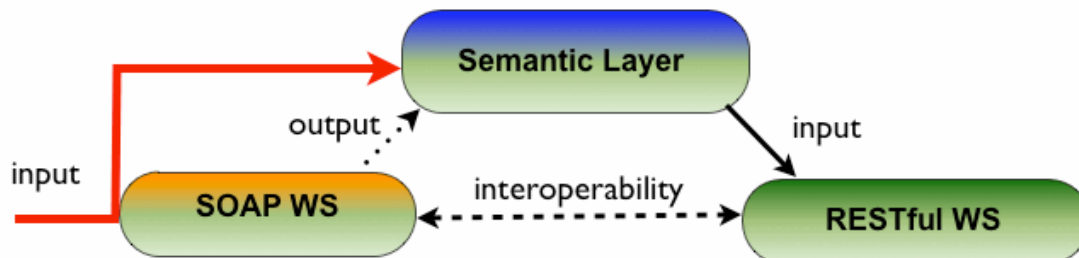


Figure 1-6 (c) SOAP WS input mapped to RESTful WS input

SWEET contribution

The research conducted as part of the Semantic Web Services element of this thesis, resulted in the use of the SWEET [13] editing tool. This tool was used to add semantic markup to the RESTful Web Service descriptions. A core issue was discovered and reported to the developer. The issue was resolved on the 27th January, 2011.

1.4 Thesis Overview

This thesis consists of three sections: the background to the contribution; the contribution and the conclusions. The chapter structure of this thesis is visually represented in Fig. 1-7. The background section contains the literature review of the specific areas of motivation and contribution. The second section contains two chapters which detail the contribution of this research and the last section relates to the conclusions of the research.

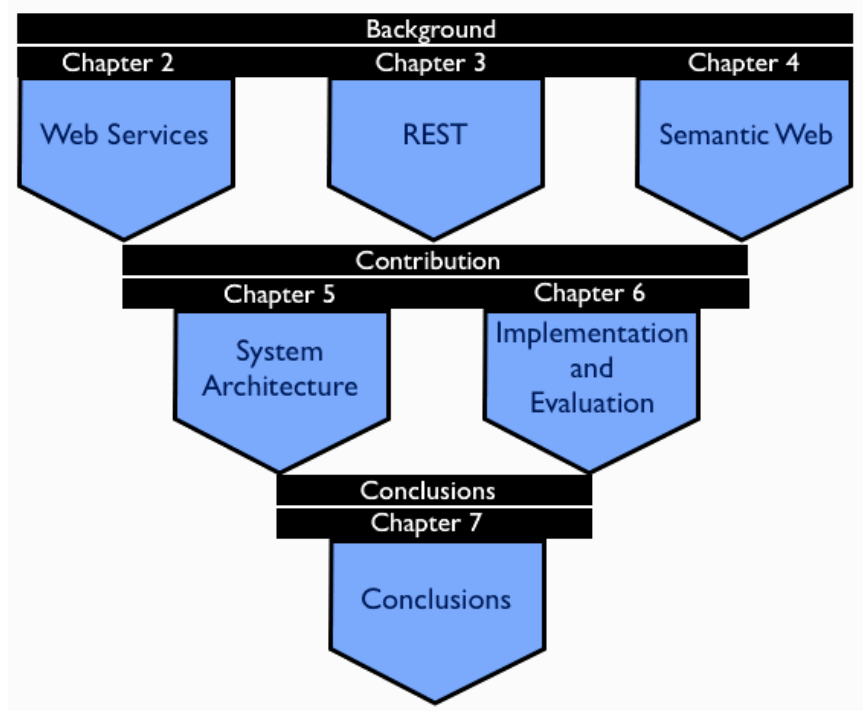


Figure 1-7 Thesis Road Map

Chapter 1: Introduction

This chapter introduces the domain and motivations behind this research. The principal research contributions are identified and briefly discussed. A high level overview of the research is presented as well as a road map of the thesis.

Chapter 2: Web Services

This chapter introduces the reader to the domain of Web Services. As “*Web Services are the de facto implementation of the Service Oriented Architecture (SOA) paradigm*” [121], an overview of Service Oriented Architecture (SOA) is given. The two popular XML Web Services approaches, namely

SOAP and Plain Old XML (POX) are discussed. The WS-* stack, to which SOAP belongs, is detailed and how the WS-* stack provides Quality of Service (QoS) is covered.

Chapter 3: Representational State Transfer (REST)

This chapter focuses on the architectural style called REST. As REST is in itself abstract, REST's most popular incarnation, the Web and especially, HTTP is discussed. How HTTP implements QoS is also detailed. Criteria for deciding whether an architecture is RESTful or not is outlined and finally, some common REST anti-patterns are detailed.

Chapter 4: Semantic Web

One of the techniques in the research offering presented in this thesis involves adding intelligence and automation to a software artefact created as part of this research. This chapter focuses on introducing the core technologies of the Semantic Web. This chapter also discusses Semantic Web Services from both the SOAP/POX WS perspective and the RESTful WS perspective. The technologies presented in this chapter enabled an intelligent and automated artefact to be created.

Chapter 5: StoRHm (SOAP to RESTful HTTP mapping)

This chapter outlines the architecture and design of StoRHm: the core contribution of this thesis. The constituent elements of StoRHm's two versions and the design decisions that guided them are discussed. The gradual evolution towards a semantically enabled and automated architecture is outlined.

Chapter 6: Implementation and Evaluation

This chapter outlines the implementation instances of the architecture outlined in Chapter 5. Test cases are outlined and examples of the on-the-wire runtime messages are presented. The architecture is then evaluated based on criteria such as message visibility and performance.

Chapter 7: Conclusions

This chapter summarises the thesis, focusing on the results and benefits of the architecture. Some interesting future directions are also discussed.

Chapter 2

Web Services

2.1 Introduction

In this chapter, an overview is given of Service Oriented Architecture (SOA) and one of its most popular implementation paradigms: Web Services. The core technologies enabling Web Services, namely SOAP and WSDL are discussed in detail. How Web Services address advanced enterprise requirements i.e. Quality of Service is outlined. Plain Old XML (POX), a more lightweight approach to Web Services is detailed. Lastly, HTTP tunnelling and Web Services interoperability is discussed.

2.1.1 Background

As outlined in the Introduction, Web Services evolved from a previous distributed computing paradigm, CORBA. CORBA is a specification maintained by the OMG (Object Management Group) that enables software components written in different computer languages, residing on different

computers using different operating systems to interoperate. The first version of CORBA (v1.0) was released in late 1991 and the latest version is v3.1, released in 2008. CORBA interfaces were specified using Interface Definition Language (IDL) for which language mappings were available to generate client-side stubs and server-side skeleton code. Object Request Brokers (ORBs) exist on both the client and server side to handle the communication between the application and the remote objects. CORBA's interoperability protocol over TCP/IP, namely Internet Inter-ORB Protocol (IIOP) was released in 1996. CORBA, a binary protocol, had issues with client-side firewalls when the Internet exploded.

XML, a text-based W3C (World Wide Web Consortium) standard, was defined in 1998. XML is a meta-language i.e. when defining an XML vocabulary, one can customise the tag names used. Customised Document Type Definitions (DTDs) or XML Schemas specify the structure and elements allowed in XML documents. The DTD/Schema enables XML instance documents to be validated. Despite its verbosity, XML's simplicity and flexibility has made it extremely popular as a data interchange format. XML has parsers for processing documents and many XML-based languages have been developed e.g. XHTML (eXtensible HyperText Markup Language), SOAP (no longer an acronym), RSS (Rich Site Summary) and Atom.

HTTP, a text-based application protocol for transferring hypertext documents was initially defined in 1991 (v0.9). The latest version is v1.1 [51]. The Internet Engineering Task Force (IETF) and W3C control the HTTP standard. What initially started as a simple means of communicating ideas within a research group led by Tim Berners-Lee has become the foundation of the modern Web. The explosion of the Web ensured that the firewall-friendly HTTP protocol is now ubiquitous.

The technologies just outlined have influenced Web Services in different ways:

- HTTP is firewall friendly (CORBA was not)
- HTTP is also pervasive
- XML was the *de facto* standard for data interchange. In fact, such is the role that XML plays in Web Services that Nordbotten states in [146] that “*the suitability of Web Services for integrating heterogeneous systems is largely facilitated through its extensive use of Extensible Markup Language (XML)*”.
- XML is also text-based

Thus, an XML-based distributed systems protocol that would leverage HTTP (by default) heralded the arrival of Web Services. The W3C first used the term “Web Services” in 2001.

Since its inception in 2001, Web Services have inspired many books including [15][16] and [18]. In [16], the authors describe Web Services as “*the next generation of distributing computing*”. A lot of academic research has also been carried out in the field of Web Services, for example:

- Web Service adapters are the subject of research in [116] and [118] where the authors propose a framework for developing Web Service adapters. The authors concern themselves with signature mismatches between functionally similar services e.g. different operation names and/or number of arguments and also business protocol mismatches e.g. different order and/or number of messages. The authors define template patterns requiring user provided XSLT (Extensible Stylesheet Language Transformations) or XQuery transformations to adapt the messages. The work done in [116] and [118] is specifically Web Services oriented and consequently differs from the research conducted as part of this thesis in the following areas:
 - RESTful WS are not integrated and as a result, a key research objective, namely, “POST tunnelling” is not addressed. In addition, POX services are not catered for.
 - The authors in [116] and [118] do not leverage the Semantic Web to automate the matching i.e. the user must provide this information manually.
- Web Services *raison d’etre* is integration. This is demonstrated by the work done in [110] where the authors implement a Web based student appeals system involving Web Services across both Java EE and .NET platforms. In [110], the authors explain in detail the SOAP invocation process e.g. marshalling to/from XML and encapsulating to/from HTTP.
- A simple atomic Web service is defined as a service that does not invoke other Web services to fulfil the consumer request; whereas a composite Web service is an aggregation of simples and/or composite Web Services [151]. As “*Web Services are autonomous units of code, independently developed and evolved*” [153], the efficient and intelligent composition of Web Services has been the subject of research attention over the years. For example, in [151], a framework that utilises the Semantic Web for the automatic composition of Web Services is outlined. The framework involves the user specifying (at a high level) their requirements (e.g. service, operations, parameters). The framework then semantically matches these requirements against predefined Web Services stored in a registry. In [153], an alternative approach to composition is outlined. The authors in [153] detail a token based approach whereby important tokens are extracted from the WSDL description and free-form Web Service description (metadata in a UDDI (Universal Description Discovery and Integration) repository). These tokens are used to calculate proximity values against other Web Service descriptions in order to suggest possible compositions. It is an exploratory approach i.e. the

framework matches and ranks rather than automatically composes. Both [151] and [153] are traditional top-down goal-centric approaches. A bottom-up approach in which a Web Service mining framework that presents potentially interesting and useful compositions to the user, is outlined in [65]. In the framework, the user initially specifies a general topic e.g. medicine. This scoping input determines (and limits) the search space in the registry. The resultant Web Services are linked together into compositions. This is achieved via semantic ontologies e.g. operations pre- and post-conditions must hold. The proposed new compositions are then presented to the user for evaluation. In [140], a framework that searches WSDL repositories based on an existing user-specified WSDL is presented. In their framework, proximity is based on keyword matching in order to rank proximity so that the user can select the WSDL to compose with.

- The Web Services standards of SOAP, WSDL and UDDI enabled “*automatic and dynamic interoperability between systems*” [133], However, these specifications are not considered sufficient given the challenges of the enterprise, especially in the area of security and reliability [113]. Thus, Web Services standards have evolved over time from a basic suite enabling interoperability to a sophisticated stack enabling many advanced features required in the enterprise. For example, security in Web Services has evolved from reliance on HTTPS to a comprehensive suite of standards, encapsulated in WS-Security. HTTPS is viewed as insufficient in a Web Services scenario because, while HTTPS secures a direct connection between one machine and another, it offers no help in the Web Services scenario where a message may travel over several connections e.g. where messages originate deep inside one enterprise and travel deep inside another (travelling over SOAP intermediaries along the way) [113][133]. If a SOAP intermediary inserts/removes a SOAP header, HTTPS breaks [146]. WS-Security relates to a family of specifications used to secure Web Services messages [146]:
 - WS-Security specifies how to apply XML Encryption and XML Signature to SOAP messages, effectively providing confidentiality and integrity to SOAP messages [147]. In addition, WS-Security also enables the inclusion of security tokens e.g. X.509 certificates into SOAP messages, thereby enabling authentication.
 - WS-Security has no concept of a conversation/session i.e. it is concerned with a single request/response exchange. If a sender is sending multiple messages to the same recipient then sending the same security token (e.g. a public key on an X.509 certificate) repeatedly is a waste of bandwidth and could result in multiple validations on the recipient side. In these situations, WS-SecureConversation is a framework that can be used for establishing and maintaining the context required for a conversation.

In the given scenario, full key exchange and validation is only required when establishing the security context and not for all the messages in the conversation.

- WS-SecurityPolicy enables a service provider to assert its security requirements and capabilities (in a WSDL file). Examples would be security tokens required to successfully invoke a service (e.g. X.509 certificates); cryptographic algorithms supported and what parts (or all) of the SOAP message needs integrity and/or confidentiality protection via digital signature and encryption respectively.
- WS-Security provides for including security tokens in SOAP messages and WS-SecurityPolicy provides for asserting what security policies are required. If a consumer does not have access to the security token required by a service, WS-Trust defines a security token service (as a Web Service) from which security tokens can be requested (by service consumers) and subsequently validated (by service providers) [147].

In [149], the authors examine WS-Security and identify the following issues:

- The “*alphabet soup of standards can be confusing to Web Services developers who don’t know which standards to follow*” [149]
 - All of the standards relate to network security and none address protecting the data while it is being stored e.g. in a server side database
 - “SQL injection” attacks are not addressed by the standards. SQL injection is a malicious SQL command such as “*delete *from orders*” inserted as XML into a Web Service request which can result in data being deleted on the server. To address this type of attack, the authors suggest using “XML firewalls”. XML firewalls operate much like traditional firewalls (which base their decisions on network address information), except that they can also filter XML traffic. Thus, data that looks like SQL can be filtered.
- An important challenge in SOA is to find adequate services to solve a particular task [131]. In Web Services architecture, Web Service descriptions are published in a registry to be later *discovered* by potential Web Service consumers. UDDI is the official registry standard. However, “*service discovery is too complex*” [132]. This is attributed to UDDI’s limited category-based keyword-browsing search facility [132] and UDDI’s complex data model/API [38]. Consequently, research has been conducted to improve the discovery of Web Services. In [132], a two-step approach that acts as a front-end to UDDI is outlined. The framework utilises linear algebra techniques for matching similar documents – each WSDL document is

mapped to a vector of elements where each element represents the importance of a word in the document. As a result, similar documents will have similar vectors. Initially the category of the query is calculated so that the search space is reduced to a subspace. Subsequently, the query is compared against service descriptions in the subspace. Service discovery is also pertinent to the work in [132] where an architecture that enables the efficient searching for services running on embedded devices is outlined. Once devices power up they register themselves on the network and advertise their services in a process called “network discovery”. The services metadata advertised during network discovery is stored in a repository. Due to the constraints on the devices, the services keywords advertised are limited. The framework requires the user to enter specific keywords that act as search criteria. The keywords are automatically augmented/enhanced with other keywords from an analysis of searching existing repositories such as web encyclopaedias (e.g. Wikipedia) and search engines (e.g. Google and Yahoo!). The services repository is searched with the augmented keyword list and all services that potentially support the functionality the user is looking for are returned [131].

2.1.2 Definition

Web Services are defined by the W3C as “*a software system designed to support interoperable machine to machine interaction over a network*” [14]. Web Services are a suite of XML based technologies that are used in the area of application integration in a heterogeneous distributed environment. A Web service is a software application (i.e. procedure, method or object) with a stable, published, programming interface that can be invoked by clients [117]. The Web services model borrows from the success of the Web. Instead of a simple markup language (HyperText Markup Language (HTML)) over a lightweight transport (HTTP) as we have in the Web; we have an open, portable, platform-neutral language, namely XML, over HTTP (or any other supported transport). All that is required for Web services is that they can understand an XML-formatted request over a supported communications transport (e.g. HTTP) and return an XML-formatted reply (if required) [15]. Web services are invoked by other programs [117] and can be invoked by small mobile devices e.g. mobile phones [129], personal digital assistants (PDAs) [111], PC-based programs, Web browsers and mainframe systems.

Web Services are “*based on the notion that pieces of software applications can be developed and then published to a registry where they can be dynamically discovered and consumed by other client applications over different transport protocols irrespective of the language used to develop those*

applications on which they are implemented” [108]. Web services are attractive because existing legacy systems can be wrapped as Web services and made available for integration with other applications, even if these applications are running on different platforms and written in different programming languages. The complexity of these legacy systems is hidden behind the Web services interface. Now, instead of integrating one system with another, we can integrate one system with many others i.e. once we expose the system in Web services, many systems can integrate with it. This ability to wrap existing software has the benefit of promoting software reuse [16]. In effect, Web services act as entry points to the local information systems [117].

Web services’ are a popular approach to implementing an SOA. According to [132]: “*The software industry has broadly adopted SOA by using Web Service technologies*”. Consequently, an overview of SOA is necessary. A discussion on Web Services architecture and its constituent core technologies then follows.

2.2 Service Oriented Architecture (SOA) [15] [16]

SOA is an architectural approach to building application systems whereby the focus is on a loosely coupled set of services which can be dynamically discovered and invoked. In an SOA, software services are exposed over a network via well-defined interfaces. This focus on the service interface abstracts the service requestor from the service implementation thereby promoting loose coupling. As applications are integrated at the interface level (and not the implementation level), SOA increases flexibility as it allows applications to integrate with any implementation of the contract. This allows the consumer to pick and choose the service it wants to invoke based on other criteria e.g. security or message reliability.

The SOA approach to building IT systems is to align IT services with business services. The ultimate goal is to create a repository of IT services that can be orchestrated/composed into business processes quickly and easily using tools. This enables enterprises to react more quickly if market conditions change e.g. a business merger.

In an SOA, existing applications can be refactored and wrapped as services, to be consumed by existing or new applications. Thus previous IT investments hidden away in legacy silos can be exposed as services and their logic reused and integrated into new applications. In the SOA approach, the designer is not building a software service for one purpose only; rather they are

building a service that can potentially be used in many different business contexts. Enterprises no longer want their business functionality buried deep in a monolithic application which is difficult to integrate with. They want to abstract the business logic out into services so that these services can be invoked and re-used from a variety of locations and platforms.

The concept of SOA is not new. However, while the idea of separating the interface from its implementation to create a software service definition has been well proven in CORBA, the ability to completely separate a service description (into a text file) from its execution environment is new. This allows us to mix and match services from different execution environments e.g. CORBA, Java EE, .NET etc... Previous SOA implementations were based on a single execution environment. With SOA, the implementation environments of the services don't matter; it's the service that's important.

2.2.1 SOA Roles

Any SOA contains three roles: service requestor, service provider and service registry:

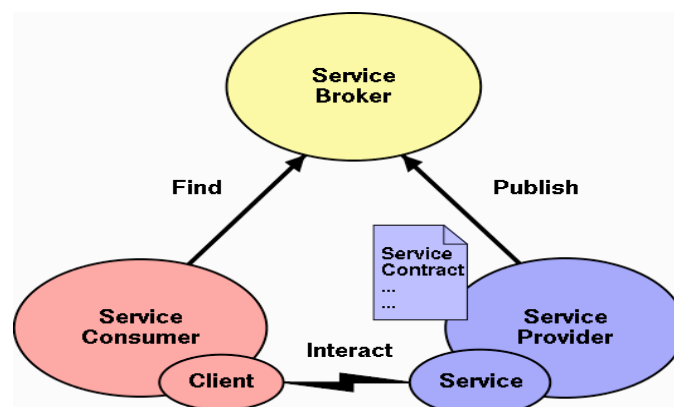


Figure 2-1 SOA Roles [17]

Service provider

A service provider creates the service description (contract). The service description is very important in an SOA because it describes everything the service requestor needs to know in order to invoke the service. The service provider deploys the service, makes it accessible over the network, *publishes* the service description in a registry (broker) and invokes the service when it receives service requests.

Service requestor

A service requestor must *find* the service description published in a service registry and then *bind* (execute or interact) the service based on the description metadata.

Service registry

The service registry is responsible for advertising service descriptions published to it by service providers and for allowing service requestors to search for services within the registry.

Service description

The key to SOA is the service description. The service description is published in the service registry by the service provider; the service description is what the service requestor “finds” and it is the service description that tells the service requestor everything it needs to know in order to bind to the service.

2.3 Web Services Architecture

“No single technological advancement has been more successful in manifesting a service-oriented architecture (SOA) than Web Services” [147]. As stated in the introduction, Web Services are a suite of XML based integration technologies used in implementing an SOA. The major advantages of implementing a SOA with Web Services are:

- the use of HTTP
 - HTTP is the default (and most common) transport and due to the success of the Web is ubiquitous
 - HTTP is firewall friendly
- the use of XML
 - simplicity – service descriptions and messages are plain text files
 - platform-neutrality – XML is by its nature both platform and language neutral thereby increasing interoperability; it is the lingua franca for information exchange [114] and platform independence [113].

Figure 2-2 illustrates the basic Web services architecture. In this diagram, the Web service producer *publishes* the service description (an XML file called a WSDL description) in a registry. The Web

service consumer accesses this registry, *finds* the services details via the published description and *binds* to the service (executes it) by sending a SOAP message to the provider [147].

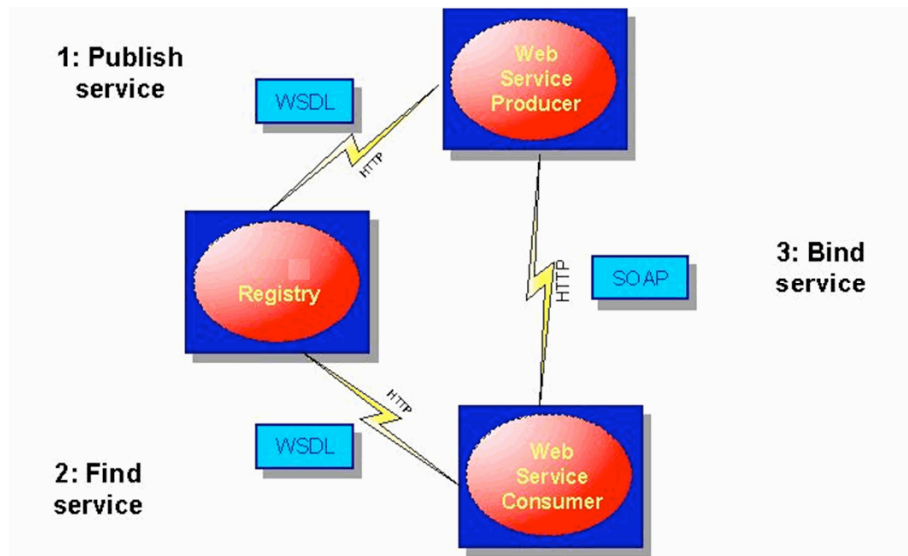


Figure 2-2 Basic Web services architecture

Web services can be considered conceptually as a layered set of technologies. There are many specifications and Figure 2-3 shows the more important ones. The stack is known as the “WS-*” interoperability stack due to the fact that the vast majority of the standards begin with the “WS-” prefix. As the Web Service framework is modular, *“you can just use the parts of the stack you need”* [113].

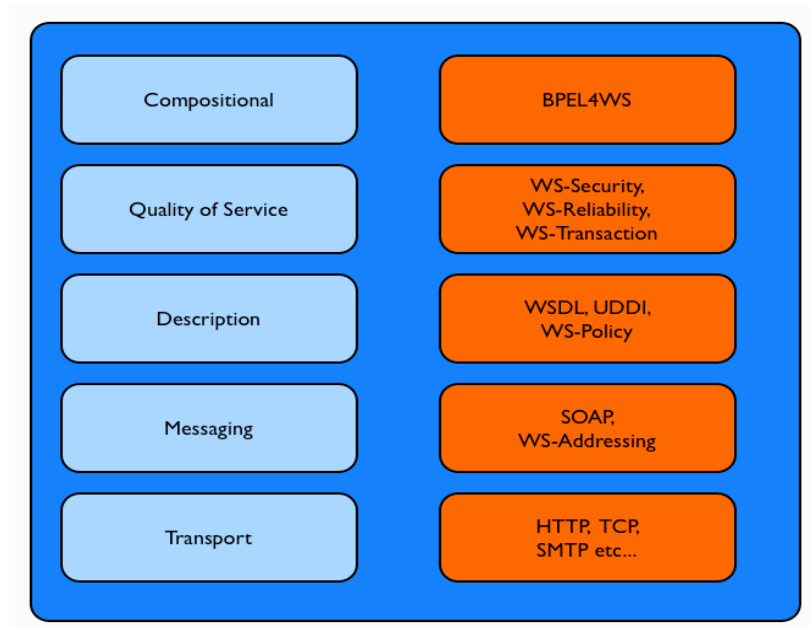


Figure 2-3 WS-* Interoperability Stack [16]

2.3.1 WS-* Interoperability Stack

The WS-* stack is a suite of XML specifications supported and maintained by various standards bodies (principally the W3C and OASIS (Organization for the Advancement of Structured Information Standards)). The specifications are organised into a stack conceptual model to help understand at which layer the specifications reside.

Transport layer

Web services are transport neutral i.e. a Web service message can be transported over HTTP, TCP, SMTP etc...The default transport is HTTP.

Messaging layer

At the messaging layer we have the core Web services technologies of XML, SOAP and WS-Addressing. XML is used for formatting the messages between producer and consumer (SOAP) and also for describing the services themselves (WSDL). SOAP was one of the original Web services standards. It defines an enveloping mechanism for Web services messages [18]. WS-Addressing standardizes the concept of response addresses i.e. where the response of the Web service should go. It is used in asynchronous messaging as it enables the response message to be addressed to a different end-point from the original request. The consumer can poll this end-point for the response at a later stage.

Description layer

A service description is additional descriptive meta-data. It helps SOA systems such as Web services achieve loose coupling and dynamic binding. With WSDL, programmers define the functional characteristics of their Web services e.g. what operations the service provides and what input/output messages are associated with those operations. *“It is the XML-based file that any developer needs in order to develop a client application to consume the Web Service”* [108]. SOAP and WSDL enable application developers to create Web Service wrappers around their applications that expose the applications functionality through a standardised interface definition language [137].

WS-Policy is an approach to describing non-functional characteristics of a Web service. For example, how does a consumer know that a particular Web Service supports message security and/or reliability? These non-functional characteristics of a Web service can be described using WS-Policy that allows Web service providers advertise the properties and capabilities of their Web service.

Depending on the policy, the requestor may have to provide extra information to validly invoke the Web service [18].

Web services are published in registries. UDDI is the most widely known specification for Web service registries. A complex specification, its adoption has been poor.

QoS layer

This layer contains specifications that deal with the quality of service of the Web services solution. At this layer, we are dealing with the capabilities and requirements of Web services with regard to reliable messaging, security and transactions.

WS-ReliableMessaging is a standard that addresses the requirement for reliable messaging. It is a protocol for exchanging SOAP messages that supports guaranteed delivery, guaranteed message ordering and non-duplication of messages [15].

WS-Security is a family of related specifications that address the need for Web service interactions to be conducted in a secure fashion.

WS-Transactions is a set of specifications addressing how Web services enable transactions.

Compositional layer

At this layer we have the WS-BPEL (Web Services Business Process Execution Language) specification. WS-BPEL was formerly known as BPEL4WS (Business Process Execution Language for Web Services). WS-BPEL is a standard that addresses how Web services can be composed or orchestrated into higher level business process workflows. *“WS-PEL is essentially a layer on top of WSDL, with WSDL defining the specific operations and WS-PEL defining how the operations can be sequenced”* [133].

How the stack works

All these WS-* standards can be intimidating but one must remember that Web services address a complicated problem: that of loosely coupled, dynamically configured, heterogeneous distributed computing. Rather than deliver one massive specification, the community is delivering a series of smaller, domain-specific specifications e.g. security, reliable messaging or transactions. The individual specifications can be composed into an overall Web services system because each Web services specification is designed to integrate with the others. For example, rather than WS-

ReliableMessaging defining its own security standard it will rely on WS-Security. One can therefore pick and choose the specifications that relate to your solution e.g. if security is not required then there is no need to use WS-Security.

WS-I

The Web Services Interoperability (WS-I) organization is an open industry organisation chartered with establishing best practices for Web Services interoperability. The WS-I was setup to help designers determine how to combine the WS-* specifications. The WS-I is standardizing combinations of specifications to increase interoperability between them [108].

2.4 SOAP

SOAP is a messaging protocol that defines the rules by which a client interacts with a Web Service [108]. The SOAP specification is maintained by the W3C and the latest recommendation is SOAP v1.2 [24]. SOAP is simple, flexible and extensible; as it is XML based, SOAP is programming-language, platform and hardware neutral. Figure 2-4 shows how client and server applications communicate in a SOAP over HTTP environment. The client method request is serialised as SOAP (i.e. native language formats to SOAP), wrapped in HTTP and directed to the Web Service. The Web Service server decodes (i.e. unwraps) the HTTP message, deserialises the SOAP message contained therein and invokes the Web Service. The Web Service response is returned via the same process: the response is serialised as SOAP, encoded in HTTP and returned to the client. The client decodes the HTTP message and deserialises the Web Service response from SOAP back to the native programming language types [110][114].

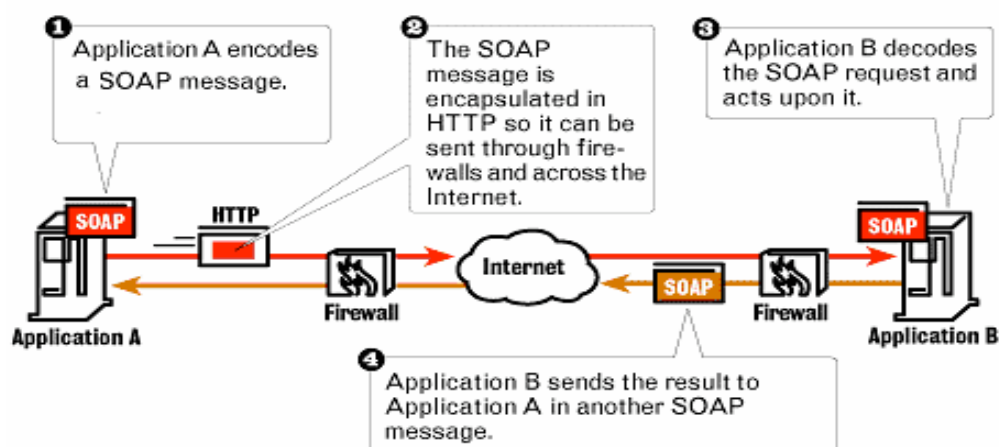


Figure 2-4 SOAP message path

SOAP provides the following [16]:

- *The SOAP Messaging Model* - this is a mechanism for defining the unit of communication for SOAP messages. All information in a SOAP message is encapsulated within a SOAP *envelope*. The envelope has a *body* and any number of optional *header* sections.
- *A processing model* - this defines how SOAP messages are handled in software e.g. SOAP engines. It is particularly important when headers/extensions are involved.
- *A mechanism for error handling* - with SOAP *faults*, one can identify the source and cause of an error
- *An extensibility model* - SOAP *headers* are used to implement extensions on top of SOAP. For example, QoS features are implemented using SOAP headers.
- *A protocol binding framework* - bindings allow the transmission of SOAP messages over arbitrary underlying transports. The default binding is to the ubiquitous HTTP transport.

Each of these topics is now examined in greater detail.

2.4.1 SOAP Messaging Model

A SOAP message is an ordinary XML document with a simple structure: “*an XML element with two child elements, one of which contains the header and the other the body*” [113]. The header contains information not visible to the end-host applications e.g. QoS; the body contains the actual payload encoded in XML format [114].

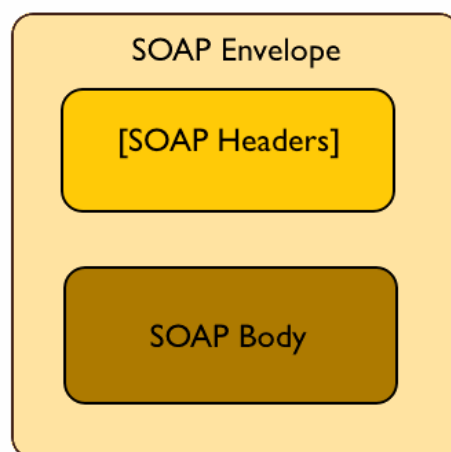


Figure 2-5 SOAP message structure

SOAP messaging defines how SOAP messages are structured and the rules processors must abide by when producing and consuming them. Listing 2-1 shows a sample SOAP message segment.

```
<?xml version="1.0" ?>
  <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
      <ns2:getPhoneNumber xmlns:ns2="http://PhoneDirServerPkg/">
        <firstName>Sean</firstName>
        <surname>Kennedy</surname>
      </ns2:getPhoneNumber>
    </S:Body>
  </S:Envelope>
```

Listing 2-1 **Sample SOAP request message**

S:Envelope is the root of the document and is defined in the *http://schemas.xmlsoap.org/soap/envelope/* namespace. The SOAP envelope contains optional headers and a mandatory body section. The headers allow vertical extensibility (discussed later). The *S:Body* surrounds the Web service to be executed and the parameters to be passed in. In Listing 2-1, we are invoking a Web service called *getPhoneNumber* which, when passed a first name and a surname, returns that person's phone number.

RPC-style versus Document-style

This discussion is covered in more detail in section 2.5.4 but a brief overview is necessary at this point so as to introduce the “wrapped document-literal” pattern. Two styles of messages are supported: *document* and *rpc* (Remote Procedure Call). These can be combined with a style of “*encoding*” or “*literal*” in the WSDL file (see section 2.5). However, the WS-I [19], state in their Basic Profile v1.1 that “*the Profile prefers the use of literal, non-encoded XML*” [20]. In [18], Papazoglou states that “*The RPC/Encoded and Document/Encoded modes are explicitly prohibited [in the WS-I Basic Profile]*”. Consequently, a discussion follows on the “*literal*” style only.

Listing 2-2 outlines a SOAP message using RPC/literal style. The *rpc-literal* style specifies that the *<soap:body>* contains an element with the name of the Web service operation being invoked. This element contains the parameters to be passed to the Web Service operation as child elements. With this style, dispatching is easy as the Web service name *myMethod* is inside the SOAP body.

However, validation is difficult as not every element in the SOAP body is defined in an XML schema – `myMethod` is a WSDL definition [21].

```
<soap:Envelope>
  <soap:Body>
    <myMethod>
      <x>3</x>
      <y>4</y>
    </myMethod>
  </soap:Body>
</soap:Envelope>
```

Listing 2-2 RPC/Literal SOAP message for *myMethod* [21]

Listing 2-3 outlines a SOAP message using document/literal style. With this style, validation is possible as all elements inside the SOAP body are defined in a schema i.e. XSD is used to literally describe the transmitted XML [113]. However, dispatching is difficult as the Web service operation to be invoked i.e. `myMethod` is no longer inside the SOAP body [21]. This issue has led to the style known as “wrapped document-literal”.

```
<soap:Envelope>
  <soap:Body>
    <xElement>3</xElement>
    <yElement>4</yElement>
  </soap:Body>
</soap:Envelope>
```

Listing 2-3 Document/Literal SOAP message for *myMethod* [21]

wrapped Document-literal

The style in Listing 2-4 is known as a “wrapped document literal”. “Wrapped” means that the child element of the `<soap:Body>` is deliberately given the same name as the Web Service operation to be invoked. This is achieved by wrapping the input parameters with an XML complex element which has the same name as the operation to be executed in the Web Service [22]. The wrapped document literal style was introduced by Microsoft in order to get the operation name back into the `<soap:Body>` when using the document-literal pattern outlined above. With this pattern, both dispatching and validation are possible. It has now become a de facto best practice [23]. Note that the sample SOAP message in Listing 2-1 follows the wrapped Document/Literal style.

```

<soap:Envelope>
  <soap:Body>
    <myMethod>
      <x>3</x>
      <y>4</y>
    </myMethod>
  </soap:Body>
</soap:Envelope>

```

Listing 2-4 **Wrapped Document/Literal SOAP message for *myMethod* [21]**

Listing 2-5 below is the SOAP response to the SOAP request from Listing 2-1.

```

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:getPhoneNumberResponse xmlns:ns2="http://PhoneDirServerPkg/">
      <return>
        <phoneNumbers>
          <phoneNumber>
            "0909745087"
          </phoneNumber>
        </phoneNumbers>
      </return>
    </ns2:getPhoneNumberResponse>
  </S:Body>
</S:Envelope>

```

Listing 2-5 **SOAP Response**

2.4.2 **SOAP Headers (vertical extensibility)**

The optional header section is how SOAP achieves message extensibility. The headers (if present) appear before the body section. Because the headers sit on top of the message, it is termed “vertical extensibility”. The headers are used to achieve QoS. SOAP QoS is discussed in more detail in Section 2.7.

2.4.3 **SOAP Intermediaries (horizontal extensibility)**

Whereas vertical extensibility is about introducing new information into a SOAP message, horizontal extensibility is about targeting different parts of the same SOAP message to different recipients. This leads us to SOAP intermediaries. SOAP intermediaries are applications that can process a SOAP

header specifically targeted at it, as the SOAP message travels from source to destination. Intermediaries are useful in several situations [16]:

- a) Securing message exchange through untrustworthy domains - this can be done by sending the message first to an intermediary that would first encrypt and then digitally sign the message. On the receiving end an intermediary would do the opposite - check the digital signature and if ok, decrypt the message.
- b) Nonrepudiation - used to make a record of a transaction (so that the transaction cannot be later repudiated or denied by either party). Instead of sending the message directly to the receiver, the sender sends it to the intermediary who makes a persistent copy of the request and then sends it on to the receiver. The response comes back via the intermediary as well and both parties are given a token to reference the transaction.
- c) Message tracing facilities - this allows a message recipient to trace the message route, complete with timings of arrivals and departures to and from intermediaries. This information is very useful for measuring quality of service and identifying bottlenecks.

All header elements can have the optional *soapenv:role* attribute. The value of this attribute is a role URI that identifies the intermediary that should handle that header. The URI can be a specific node or a class of nodes e.g. “any cache manager along the message path”. As such, nodes can play many roles and if a message arrives at a node for one (or more) of the roles the node plays, the node processes the relevant headers, removes the headers from the message and then forwards on the message (assuming the node is not the ultimate receiver). Headers that are not targeted at the intermediary are passed on untouched.

There are some specific roles i.e. URIs :

- <http://www.w3.org/2003/05/soap-envelope/role/next> - this role targets all nodes in the message path i.e. all nodes must recognise this role; it is useful where hop-by-hop processing is required e.g. tracing a message.
- <http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver> - this role targets the final recipient (the node which processes the SOAP body); this is the same as not setting the role attribute on the header i.e. the final recipient processes headers which have no role set
- <http://www.w3.org/2003/05/soap-envelope/role/none> - no node plays this role; however this header can be inspected and as a result, it is a way of passing data along the full message path. For example, if one updated a schema but did not want older systems to fail validation, one could put the new information into a header and mark it with the *none* role. Thus, newer

systems (which have the newer schema) will be able to retrieve the new data without impact to older systems.

“mustUnderstand” attribute

A SOAP header targeted at a node does not necessarily have to be processed by the node. This is because the SOAP 1.2 specification states that a SOAP node *may* process non-mandatory headers targeted at it [24]. To ensure a header is processed by the targeted node, one uses the “*mustUnderstand*” attribute. If a message header arrives with the correct role and the *mustUnderstand* attribute is set, then the node must process the header. If a node is unable to process a mandatory header block it must generate a SOAP fault (error) and stop processing.

“relay” attribute

The relay attribute is used to ensure that a header is forwarded on to the next node in the message path i.e. even if a node processes the header, it must not remove the header. This would be useful in hop-by-hop processing scenarios e.g. message tracing.

2.4.4 SOAP Processing model

The processing model outlines what are the steps to perform when a node receives a message [16]:

- a) Identify the mandatory headers that are targeted at the node e.g. role set to “*../next*” and *mustUnderstand* set to *true*
- b) If the node fails to understand any of the mandatory headers identified then generate a SOAP fault and stop processing.
- c) Process all the mandatory headers identified and if the node is the ultimate receiver, process the SOAP body. A SOAP node *may* also choose to process non-mandatory SOAP headers targeted at it.
- d) If an intermediary (i.e. not the final recipient), remove the headers that have been processed (except where the *relay* attribute is *true*) and pass on the message.

2.4.5 SOAP faults (errors)

To signify that an error has occurred, a SOAP fault is generated. This is much like an exception in Java. A SOAP fault is a normal SOAP message except that you have *Fault* as the main element inside the SOAP *Body* element. Here is an example:

```

<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:st="http://www.skatestown.com/ws">
  <soap:Body>
    <soap:Fault>
      <soap:Code>
        <soap:Value>soap:Sender</soap:Value>
        <soap:Subcode>
          <soap:Value>st:InvalidPurchaseOrder</soap:Value>
        </soap:Subcode>
      </soap:Code>
      <soap:Reason>
        <soap:Text>Purchase Order failed validation</soap:Text>
      </soap:Reason>
      <soap:Detail>
        <st:LineNumber>9</st:LineNumber>
        <st:ColumnNumber>24</st:ColumnNumber>
      </soap:Detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>

```

Listing 2-6 SOAP Fault Message [16]

In SOAP 1.2, a SOAP *Fault* element, contains a *Code* element, followed by a *Reason* element followed by a *Detail* element. The *Code* element outlines the general issue with the following possibilities:

- Sender – incorrect or missing data in the request message from the client. In Listing 2-6, the sender has sent an incomplete XML document i.e. it is the senders responsibility to correct the issue.
- Receiver - something went wrong at the receiving end while processing the message e.g. a database was down. If the original message is resent, it may succeed.
- mustUnderstand - a mandatory header was not understood by the targeted node
- versionMismatch – there is a mis-match between the SOAP message and the SOAP server

The *Subcode* element (sub-element of Code) allows us to provide further detail. In this case, it is an invalid purchase order.

The *Reason* element contains a human-readable description of the error (which could be displayed to the user or logged).

The *Detail* element contents depends on the SOAP server implementation i.e. the server can insert any information necessary to help the client further understand the cause of the fault. In Listing 2-6, the contents reflect the line and column numbers of where validation had an issue with the request document.

2.4.6 Transport binding

A *protocol binding* is a specification, named with a URI, of how to serialise SOAP messages over a particular underlying protocol. SOAP is “protocol agnostic”, which means SOAP can bind to any

transport. However, HTTP, identified by the URI <http://www.w3.org/2003/05/soap/bindings/HTTP/>, is the default transport and is the only binding supported by the WS_I Basic Profile. If you are sending SOAP messages over HTTP then the SOAP HTTP binding describes how you take a SOAP message from one node and serialize it across a HTTP connection to another node. Bindings serve to pass messages between adjacent nodes along the message path. If, for example, one wanted to send a message securely, one could do it either by using a protocol binding or SOAP headers. For example, assuming that HTTP is available (and HTTPS is viewed as secure enough), one could use the SOAP-HTTPS binding to serialize the SOAP messages. Alternatively, if HTTP is not available (or HTTPS is not viewed as being secure enough), one could use a SOAP header e.g. WS-Security to encrypt/sign the SOAP message (as this will work over *any* binding).

2.4.7 Message Exchange Patterns (MEPs)

An MEP is a feature that specifies how many messages move around in an interaction, where they originate from and where they end up. SOAP 1.2 [24] defines two standard MEPs:

- Request-Response and
- SOAP Response

Figure 2-6 outlines the Request-Response MEP. The Request-Response MEP is straightforward: the requesting node sends a SOAP message to a responding node. The responding node replies with a SOAP message of its own to the requesting node. This response could be a fault.

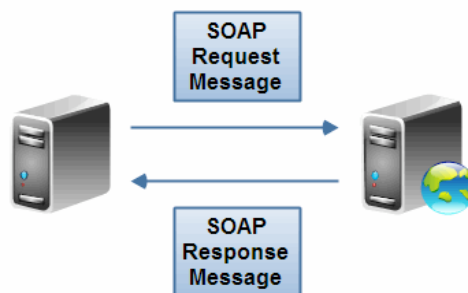


Figure 2-6 Request-Response MEP

Figure 2-7 outlines the SOAP Response MEP. The SOAP Response MEP is similar to the SOAP Request-Response MEP except that the request message is *not* a SOAP message [24]. This means that the receiving node does not have to implement the SOAP processing model (examining headers etc...). For example, it could be a HTTP server that responds with SOAP messages stored on a file system. The SOAP Response MEP was introduced in SOAP 1.2[24] so as to support REST-style

interactions with SOAP; for example, the request could be a simple HTTP GET with the HTTP server free to respond with any SOAP message it chooses.

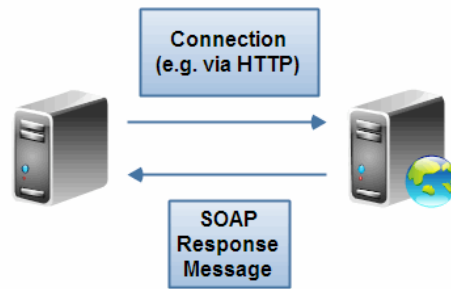


Figure 2-7 SOAP Response MEP

2.4.8 SOAP 1.2

SOAP v1.2 became a W3C recommendation in April 2007. The principal reason for discussing it here is that certain features were introduced to address the issues that are at the core of this thesis. This is echoed in [137]: “*the incorporation of REST ideas into the latest SOAP*”. Note that a “*feature*” is abstract. It is given a name (a URI) and a specification stating what it does. It is generally convenient to provide the specification at the URI naming the feature i.e. type in the features URI into a browser and what is returned is a description of what the feature does. Features are implemented at the SOAP layer by SOAP headers (modules) or below the SOAP layer by SOAP bindings. The relevant features are as follows:

Message Exchange Pattern (MEP) feature

The MEPs in section 2.4.7 were introduced in SOAP 1.2. The important one from a Web-friendly perspective is the SOAP Response MEP (<http://www.w3.org/2003/05/soap/mep/soap-response/>). This MEP must use the abstract “GET” WebMethod feature which the HTTP binding translates into an HTTP GET message [24]. The other MEP is the SOAP Request-Response MEP (<http://www.w3.org/2003/05/soap/mep/request-response/>). This MEP must use the abstract “POST” WebMethod feature which the HTTP binding translates into an HTTP POST message i.e. your normal SOAP request. This is how SOAP 1.1 operated.

WebMethod feature

The WebMethod feature (named <http://www.w3.org/2003/05/soap/features/web-method/>) allows us to specify GET or POST as the HTTP verb to be used (depending on the MEP used).

SOAP HTTP Binding

The binding (named *<http://www.w3.org/2003/05/soap/bindings/HTTP/>*) outlines how the above MEPs and WebMethod features are to be implemented in HTTP. Note that, in section 7.1.2 of the SOAP 1.2 specification [24] it states that “*This binding is not intended to fully exploit the features of HTTP... Therefore, this HTTP binding for SOAP does not specify the use and/or meaning of all possible HTTP methods, header fields and status responses*”. As outlined in section 7.4 of the specification [24], the value of the WebMethod feature depends on the MEP in use:

MEP	WebMethod
" http://www.w3.org/2003/05/soap/mep/request-response/ "	“POST”
" http://www.w3.org/2003/05/soap/mep/soap-response/ "	“GET”

Table 2-1 SOAP 1.2 MEPs [24]

The SOAP 1.2 specification attempts to address the criticisms that SOAP 1.1 was not “Web-friendly”. One of the principle concerns was that all SOAP 1.1 messages were issued with the same HTTP verb i.e. POST. This disabled Web intermediaries such as caches and proxies and the valuable work they do because the HTTP verb GET was not used. This changed in SOAP 1.2 with the introduction of an updated HTTP binding that covered the features outlined above. However, “*very few people use SOAP 1.2; of those who do, even fewer use the WebMethod feature*” [25]. Tilkov reiterates this in his Devovx presentation [26]. I believe this is because of the impact involved in such a migration. Firstly, all the clients must change to use the new MEP and WebMethod features of SOAP 1.2. Not only that, but a Web Server must be setup (to co-exist with the SOAP server) in order to respond to client HTTP GET requests to the new URI’s. Specifically, as shown in Figure 2-6, the MEP of direct interest relates to the SOAP Request-Response MEP i.e. where the client issues a SOAP request message using POST.

2.5 Web Services Description Language (WSDL)

WSDL is the Interface Definition Language (IDL) for Web services. WSDL describes the service description (both functional and non-functional). The service description is key in an SOA in that the provider publishes it, the consumer finds it and then binds to the Web service based on the service description’s contents.

2.5.1 Service Description

The service description is a key component in an SOA in that it tells a service requestor everything it needs to know in order to invoke the Web service. A Web service has two major components: its functional and non-functional descriptions. The functional description details the syntax of the application-specific message, the transport to be used and where the service resides i.e. where to send the request messages. The non-functional description describes details that are secondary to the message and are specified in SOAP headers. The non-functional description enables a provider to specify the properties and capabilities of the Web service. For example, are there any extra requirements (e.g. Security headers) that requestors need to provide. For most of the remainder of this section the aspects of WSDL relating to the functional description are covered, followed by a section on WS-Policy covering the non-functional aspects.

2.5.2 WSDL 1.1/2.0

According to [27], WSDL 1.1 “*has become a de facto standard for the description of Web services*” and that “*WSDL 1.1 remains the predominant technology for describing Web services*”. The differences between WSDL 1.1 and 2.0 are discussed in section 2.5.7.

2.5.3 WSDL 1.1 Structure

A WSDL description has three essential properties [15]:

- the service signature - the operations (methods) the service provides and the data passed (arguments and returns)
- how to access the service - the transport protocols to be used e.g. SOAP over HTTP
- where the service is located - the protocol specific network address of the endpoint hosting the Web service e.g. a URL (Uniform Resource Locator).

Figure 2-8 gives a diagrammatic overview. The constituent elements are then explained in detail. Note that there is a clear separation of concerns into abstract (light blue portion) and concrete descriptions (beige portion) [133]. This enables the same Web service to be offered via different mechanisms e.g. the Web service could be offered over HTTP (via a URL) and also offered over SMTP (via an email address).

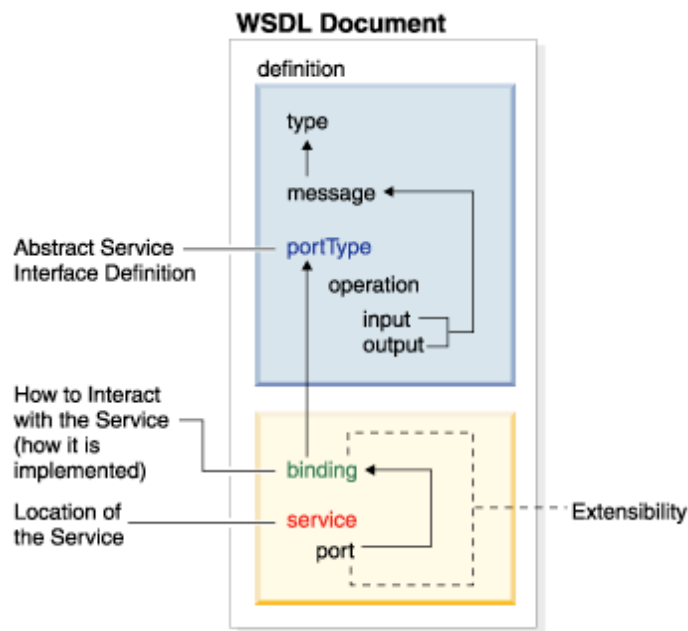


Figure 2-8 Overview of WSDL 1.1 file structure [28]

The WSDL file is often broken up into two separate files so that the service interface definition is separate from the service implementation definition:

- service interface definition (abstract description)
 - the *types*, *message*, and *portType* elements (the “what”)
- service implementation definition (concrete description)
 - *binding* elements (the “how”)
 - the *port* and *service* elements (the “where”)

The service interface can then be published on a well-known website and any organization interested in implementing that Web service is free to do so using their own preferred implementation.

***definitions* element**

The Definitions element is the root of a WSDL document. It defines the namespaces to be used and their associated prefixes.

***portType* element**

This is the element that *abstractly* defines the interface of the Web service. It is key to understanding what the Web service does because all the other WSDL definitions exist to support the *PortType*. It is similar to a Java interface in that it specifies what operations (methods in a Java interface) are available. A *portType* is a collection of operations that are supported by an end point [113]. Even though a WSDL document can contain more than one *PortType*, the convention is to have only one

[16]. This granularity of having one WSDL document per Web service interface definition promotes reuse i.e. one can import the WSDL interface definition into a second WSDL document which implements the interface differently.

operation element

An *operation* in WSDL is equivalent to a method signature in Java. A *PortType* can have many *operation* elements i.e. the Web Service can support many operations. Each *operation* defines a message exchange pattern via its *input*, *output* and optional *fault* messages [113]. These messages define the input to and output from the Web service *operation*.

message element

Messages are aggregations of *part* elements. The *messages* defined are referred to from the *operation* section.

part element

The *part* element links the *message* element and the *types* section (the native XML schema section).

types element

The default type system in WSDL is XML Schema. To increase interoperability between Web services, one should only use types defined using XML Schema. The *types* section is essentially a place where user-defined XML types and elements can be specified for later use from *parts* elements. A types section contains a schema section where one defines the application-specific types required. Alternatively, one can import externally defined XML Schemas.

At this point we know the abstract or reusable portion of the WSDL definition i.e. the *portType*, *message* and *types* elements. However, we do not know how to format the message (RPC or document style), what transport protocol to use or the end-point where the Web Service resides. The concrete description section of WSDL (beige section of Fig. 2-8) answers all these questions.

binding element

The *binding* element tells a requestor what transport protocol to use and how the application inserts the payload into the body of the SOAP message e.g. document-literal [114]. For example, if a message is to be sent using SOAP over HTTP, the binding describes how the message parts are mapped to elements in the SOAP <Body>. The *binding* element structure is very similar to the *portType* element – this is no coincidence as the *binding* is the implementation of the *portType* [18].

A *portType* can be associated with many *binding* elements. For interoperability reasons, each *portType* should have one SOAP/HTTP binding defined because many clients can participate in a protocol defined by SOAP over HTTP. The *type* attribute (of the *binding* element) identifies which *portType* a binding relates to.

If the transport used is SOAP then the first sub-element of the WSDL *binding* element will be a SOAP *binding* element from the SOAP namespace (see line 41 in Listing 2-7). The SOAP *binding* element has two attributes: the *style* attribute specifies whether the operations are document-style (“document”) or RPC-style (“rpc”) and the *transport* attribute, which specifies via a URI how the SOAP messages will be transported e.g. for SOAP over HTTP the URI is (“<http://schemas.xmlsoap.org/soap/http>”).

The *operation(s)* are the next sub-elements under *binding*. The *operation* attribute *name* ties the implementation *operation* element (in *binding*) to the abstract *operation* element (in *portType*). The *operation* element has *input*, *output* and optional *fault* sub-elements. The *input/output/fault* sub-elements have a *soap:body* child element (from the SOAP namespace, assuming *soap* prefix) with a *use* attribute. This *use* attribute is generally set to “literal” (as it is the recommendation of the WS-I Basic Profile [20]). This means that the content of the SOAP body will literally be the XML schema types/elements referred to in the WSDL.

The first sub-element of (the WSDL) *operation* is a *soap:operation* with a *soapAction* attribute. In SOAP 1.1, all XML documents (including SOAP documents) had the same media type i.e. there was no way to differentiate SOAP traffic from other XML documents going over HTTP connections. The Request URI was that of the SOAP server and thus the server had to parse the SOAP message to figure out what operation to invoke [5][136]. To address these issues, an HTTP header named *SOAPAction* was introduced. Its value was a URI and it was used “to indicate the intent of the SOAP HTTP request” [29]. This enabled firewalls to identify and filter SOAP messages and enabled servers in routing the message. Note however, that the *SOAPAction* header is discouraged by the WS-I Basic Profile [101].

SOAP 1.2 uses the *application/soap+xml* [30] media type. This new media type has an optional parameter called *action* that can be used as SOAP 1.1 used the HTTP header *soapAction* [30] i.e. set it to the URI of the relevant operation to be called by the Web service. Thus, in SOAP 1.2, the issue of identifying the intent of SOAP messages using the HTTP header *soapAction* is no longer needed. Listing 2-7 was constructed using Netbeans v6.8 where the SOAP default was 1.1. In Listing 2-7, the

soapAction attribute is “” which implies that the SOAP Body element will have to be parsed for dispatching purposes.

service element

The service element takes the *binding* element(s) defined previously and ties them to one or more *port* elements. A *port* element, which is a sub-element of *service*, is a single endpoint defined as a combination of a *binding* and a network address. A *port* element defines the endpoint at which the operations of a specific *portType*, using a particular transport protocol, can be invoked [18]. For example, the endpoint is a URL when HTTP is the transport protocol used. The *port* refers to the *binding* which in turn refers to the *portType*. Thus the abstract interface in the *portType* is tied to an endpoint in the *port* via the *binding*. A *service* can contain a set of related *ports* i.e. the interface is implemented by two or more bindings (e.g. SOAP/HTTP and SOAP/SMTP). Fig. 2-9 below details this.

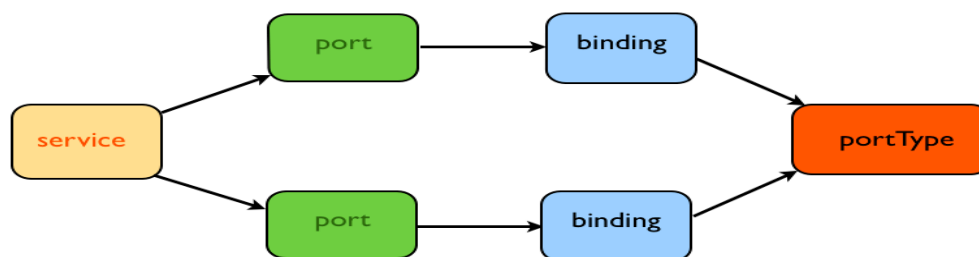


Figure 2-9 service relationship to *portType* in WSDL [18]

An example

Listing 2-9 details an example service called `PhoneDirectoryService` (line 57). Note that the default namespace is the WSDL namespace i.e. “`http://schemas.xmlsoap.org/wsdl/`”. The service contains one port element that refers to the `PhoneDirectoryPortBinding`. The port’s `soap:address` gives us the endpoint location of where to send our messages. The binding `PhoneDirectoryPortBinding` (line 40) binds to the portType `PhoneDirectory` (line 33). `PhoneDirectoryPortBinding` states that the messaging/transport protocol is SOAP/HTTP (line 42) and that the style of messaging is document-style (line 42). One operation is defined in the binding, namely `getPhoneNumber`. This operation defines both an input and output message and does not use the `soapAction` parameter. These messages are literal messages i.e. their contents are XML elements. The portType `PhoneDirectory`, referred to from the binding, defines an abstract operation namely `getPhoneNumber`, with an input and output message (in that order). These messages (`getPhoneNumber` and `getPhoneNumberResponse`) are defined using part elements. The

parts are defined in the XML schema types section - `getPhoneNumber` (the input) is defined as consisting of two XML strings and `getPhoneNumberResponse` (the output) is also an XML string.

```

1 <definitions
2   targetNamespace="http://PhoneDirServerPkg/" name="PhoneDirectoryService"
3   xmlns:tns="http://PhoneDirServerPkg/"
4   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
5   xmlns="http://schemas.xmlsoap.org/wsdl/"
6
7   <types>
8     <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
9       <xs:element name="getPhoneNumber" type="tns:getPhoneNumber"/>
10      <xs:element name="getPhoneNumberResponse"
11        type="tns:getPhoneNumberResponse"/>
12      <xs:complexType name="getPhoneNumber">
13        <xs:sequence>
14          <xs:element name="firstName" type="xs:string" />
15          <xs:element name="surname" type="xs:string" />
16        </xs:sequence>
17      </xs:complexType>
18      <xs:complexType name="getPhoneNumberResponse">
19        <xs:sequence>
20          <xs:element name="return" type="xs:string" />
21        </xs:sequence>
22      </xs:complexType>
23    </xs:schema>
24  </types>
25
26  <message name="getPhoneNumber">
27    <part name="parameters" element="tns:getPhoneNumber"/>
28  </message>
29  <message name="getPhoneNumberResponse">
30    <part name="parameters" element="tns:getPhoneNumberResponse"/>
31  </message>
32
33  <portType name="PhoneDirectory">
34    <operation name="getPhoneNumber">
35      <input message="tns:getPhoneNumber"/>
36      <output message="tns:getPhoneNumberResponse"/>
37    </operation>
38  </portType>
39
40  <binding name="PhoneDirectoryPortBinding" type="tns:PhoneDirectory">
41    <soap:binding
42      transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
43    <operation name="getPhoneNumber">
44      <soap:operation
45        soapAction="" />
46      <input>
47        <soap:body use="literal"/>
48      </input>
49      <output>
50        <soap:body use="literal"/>
51      </output>
52    </operation>
53  </binding>
54
55  <service name="PhoneDirectoryService">
56    <port name="PhoneDirectoryPort" binding="tns:PhoneDirectoryPortBinding">
57      <soap:address location="http://example.com:8080/PhoneDirectoryService"/>
58    </port>
59  </service>
60 </definitions>

```

Listing 2-7 Sample WSDL 1.1 file

2.5.4 RPC vs. Document style

A SOAP message's payload (i.e. contents) is structured depending on whether one is using RPC-style or document-style messaging. The style of messaging used is specified in the WSDL file.

RPC-style

When using RPC-style messaging, clients issue their request as a method call with a set of parameters, which returns a response containing a return value. In order to facilitate this paradigm, RPC style supports the automatic serialisation/deserialisation of programming constructs such as methods, parameters and responses to and from XML when transmitting messages over-the-wire. The client and Web service are tightly coupled and communicate synchronously [18].

RPC-style messages have a definite structure: the SOAP *<Body>* element's first child represents the operation to be performed i.e. this element will have the same name as the *operation* in the *portType*. This element's children represent the parameters of the method call. For example, in Listing 2-2, the method called is *myMethod* and the parameters passed are *x* and *y* with values 3 and 4 respectively. To specify that one wants RPC-style, set the *style* attribute to "*rpc*" in the *binding* section of the WSDL file (line 42 of Listing 2-7).

Document-style

Document-style (or message-style) Web services exchange arbitrary XML document fragments. The focus is on a client sending an entire document e.g. a purchase order, rather than a discrete set of parameters. The Web service processes the document and may or may not return a response, leading to an asynchronous communication model (as the client is free to continue processing without waiting for a response). The response, if any, may appear at any time later [18].

There is no serialisation/deserialisation to/from XML as in RPC; instead there is an assumption that the contents of the SOAP *<Body>* is well formed XML containing arbitrary application data. The SOAP runtime accepts the SOAP *<Body>* element as is and hands it over to the target application unchanged. To specify that one wants Document-style, set the *style* attribute to "*document*" in the *binding* section (line 42 of Listing 2-7). Note that Listing 2-7 defines a document-literal SOAP message i.e. the messages defined "*will be included in the message payload 'as is', with no further encoding performed*" [114].

2.5.5 WSDL Message exchange patterns (MEPs)

In section 2.5.4, we discussed the payload of a SOAP message as either an XML document fragment (document/literal) or an XML representation of a method, its parameters and return values (RPC-literal). In this section we are concerned with the flow of messages (not their contents). WSDL supports four patterns for exchanging messages. The message exchange pattern (MEP) for an operation is defined by the existence and ordering of the messages. The two most popular message exchange patterns are: *one-way* and *request/response*.

One-way

In this pattern, a service receives a message but does not send back a response. This is thought of as an asynchronous model as the client does not have to block for a reply. An *operation* which defines an *input* message but no *output* message is defining a *one-way* MEP.

```
<portType name="PhoneDirectory">
  <operation name="updatePhoneNumber">
    <input message="tns:somePhoneNumber"/>
  </operation>
</portType>
```

Listing 2-8 One-way MEP

Request-response

In this common pattern, a request message is sent to a particular service, the service then fulfils the request and returns a response accordingly [107]. This is a synchronous model as the client blocks waiting for a response. An *operation* which defines an *input* message followed by an *output* message is defining a *request-response* MEP.

```
<portType name="PhoneDirectory">
  <operation name="getPhoneNumber">
    <input message="tns:aPhoneNumber"/>
    <output message="tns:aPhoneNumberResponse"/>
  </operation>
</portType>
```

Listing 2-9 Request-response MEP

Note that while both RPC-style and Document-style messaging can be used with both one-way and request/response MEPs, RPC-style messaging is commonly used with request/response messaging [18]. Note also, that Listing 2-7 defines the abstract request/response MEP (lines 35,36) and that HTTP is the transport to be used (line 42). Thus, the abstract request/response MEP defined is implemented by HTTP's native request/response model.

Notification

In this pattern, a service sends a message to the client but does not expect a response. This is a push model where a service needs to notify subscribed clients of events as and when they occur. An *operation* which defines an *output* message but no *input* message is defining a *notification* MEP.

```
<portType name="somePortType">
  <operation name="someOperation">
    <output message="tns:SomeEvent"/>
  </operation>
</portType>
```

Listing 2-10 Notification MEP

Solicit/response

This pattern is very similar to the Notification pattern except that a response is expected from the client. It is the exact opposite to the request-response model in that it is the server that is initiating the operation. An *operation* which first defines an *output* message followed by an *input* message is defining a *solicit/response* MEP.

```
<portType name="somePortType">
  <operation name="someOperation">
    <output message="tns:SomeEvent"/>
    <input message="tns:SomeResponse"/>
  </operation>
</portType>
```

Listing 2-11 Solicit-response MEP

2.5.6 Non-functional (WS-Policy)

The non-functional aspects of Web services are detailed by WS-Policy. This allows service providers to advertise the properties and capabilities of their Web service and enables consumers to check that

a Web service provides the facilities it requires. For example, providers can state that their Web service operates in a secure environment enabling consumers requiring secure information exchange to realise that the provider meets their requirements. When advertising a WSDL file that supports QoS, Web service providers specify the extra information required for the successful invocation of the Web service, such as security headers.

Assertions

Policy assertions are the building blocks of policies. Assertions are context-specific i.e. they relate to security, reliable messaging etc... One chooses the relevant assertions and then combine them into a policy. WS-Policy defines certain XML elements to facilitate these capabilities [18]:

`<wsp:Policy>` is the overall container element that defines a policy expression

`<wsp:ExactlyOne>` and `<wsp:All>` are the two most frequently used policy operators. They enable us to combine assertions in different ways:

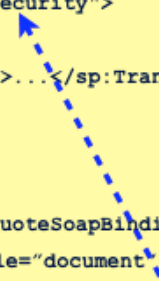
`<wsp:ExactlyOne>` means that a message must conform to exactly one of the assertions that are children of this element

`<wsp:All>` means that a message must conform to all of the assertions that are children of this element

WS-PolicyAttachment

WS-PolicyAttachment is a specification used when associating a policy with a Web service. Policy expressions can be defined internally (as in Listing 2-12) or externally and referenced from the WSDL document. The `wsp:PolicyReference` is used within a WSDL to refer to a policy.

Listing 2-12 details a simple WS-Policy example. The `wsu` prefix relates to OASIS's Web Services Security Utility specification. This specification contains an `Id` element, which is used to identify the policy and to associate the policy with a `binding` element. The policy expression `wsp:Policy` in Listing 2-12 contains one policy assertion. This assertion `sp:TransportBinding` (where the `sp` prefix relates to the Web Services Security Policy XML namespace), states that transport layer security (HTTPS) must be used when invoking the operation `GetShipmentPrice`. The `PolicyReference` element in the `binding` has a `URI` attribute which associates the binding with the identified policy.



```

<wsp:Policy wsu:Id="TLS_Security">
  <wsp:All>
    <sp:TransportBinding>...</sp:TransportBinding>
  </wsp:All>
</wsp:Policy>
...
<wsdl:binding name="ShipQuoteSoapBinding" type="ship:Quote">
  <soap:binding style="document" transport="...soap/http" />
  <wsp:PolicyReference URI="#TLS_Security" />
    <wsdl:operation name="GetShipmentPrice">
      ...
    </wsdl:operation>
</wsdl:binding>

```

Listing 2-12 WS-Policy example [31]

2.5.7 WSDL 2.0 differences

WSDL 2.0 [32] became an official W3C recommendation in June 2007 (WSDL 1.1 is a W3C Note). While initially it was intended to be a minor upgrade to WSDL 1.1 [33] the changes were considered significant enough to call the new version 2.0. The primary differences are in the areas of:

- message and part elements removed
- portType and port elements renamed to interface and endpoint respectively
- the method of expressing MEPs and operation styles (RPC/Document) has been changed
- SOAP 1.2 binding support
- the HTTP binding supports GET, PUT, POST and DELETE (previously only GET and POST was supported). Note: WSDL 2.0 “*was designed with REST Web services in mind*” [111].

Fig. 2-10 shows the differences. Note that the tool used for Web services development as part of this thesis was Netbeans v6.8 [34]. This supported SOAP 1.1 (default) and WSDL 1.1. Note also that the WS-I Basic Profile v1.1 [20] uses WSDL 1.1 and not WSDL 2.0.

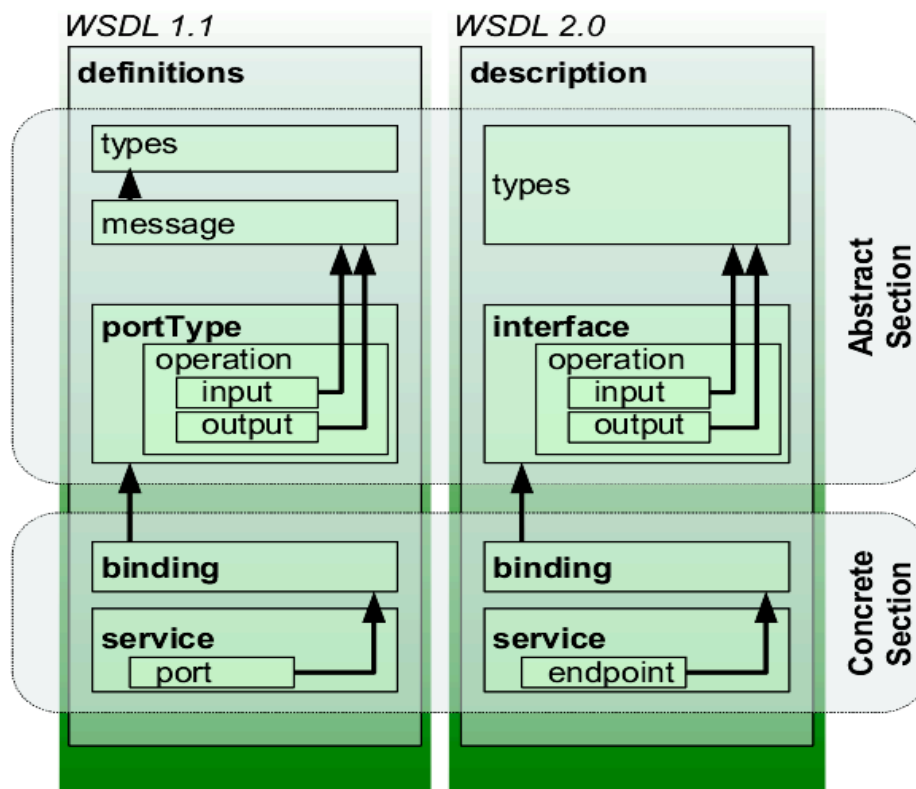


Figure 2-10 WSDL 1.1 versus WSDL 2.0 [35]

Differences [16]

- *message* and *part* elements removed
 - in WSDL 1.1 these elements were referred to by the *input*, *output* and *fault* elements from the *portType* section; now the *input*, *output* and *fault* elements refer to the XML schema elements directly
 - *portType* element renamed to *interface*
 - introduced to align the name of the element with its semantics
 - *operation* element (*interface* section):
 - *pattern* attribute - WSDL 2.0 introduces eight MEPs, all of which are identified by URI. Whereas in WSDL 1.1 the ordering of the input and output messages defined the MEP, WSDL 2.0 uses the *pattern* attribute. The most popular are listed below (note that these are the only ones discussed in the Adjuncts section of the specification [36])
- :
- In-Only: identified with the URI “*http://www.w3.org/ns/wsd/in-only*” means that there is exactly one input message only.
 - Robust In-Only: identified with the URI “*http://www.w3.org/ns/wsd/robust-in-only*” means that there is exactly one input message with the possibility of faults being generated and returned [37])

- In-Out. identified with the URI “*http://www.w3.org/ns/wsd/in-out*” means that there is exactly one input message followed by exactly one output message.
- *binding* element
 - *interface* attribute to refer to the *interface* element that this binding is associated with
 - *type* attribute : a URI that identifies the binding in question; for example, to bind to SOAP, the value of this URI is *http://www.w3.org/ns/wsd/soap* (to bind to HTTP, this attribute would have a value of *http://www.w3.org/ns/wsd/http*).
 - *wsoap:protocol* attribute (where *wsoap* is a prefix for the *http://www.w3.org/ns/wsd/soap* namespace): the value of this attribute determines which protocol SOAP uses as its underlying protocol e.g. to use SOAP over HTTP this attribute would be set to “*.../bindings/HTTP*”.
 - *wsoap:version* attribute: this attribute can be used to specify the version of SOAP used.
 - *wsoap:mepDefault* attribute: this attribute can be used to specify the SOAP MEP for all operations inside this binding e.g. for SOAP Request-Response MEP this attribute will have a value of *http://www.w3.org/2003/05/soap/mep/request-response* (alternatively, this value can be assigned to the *wsoap:mep* attribute on each *operation* in the binding as in Listing 2-13).
- *service* element
 - *interface* attribute
 - links the *service* with the *interface*
 - *port* element renamed to *endpoint*
 - introduced to align the name of the element with its semantics

Listing 2-13 is a WSDL 2.0 version of the WSDL 1.1 file (detailed in Listing 2-7).

```

1 <description xmlns="http://www.w3.org/ns/wsdl" xmlns:tns="http://PhoneDirServerPkg/"
2       targetNamespace="http://PhoneDirServerPkg/">
3   <types>
4       <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5           xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
6           xmlns="http://schemas.xmlsoap.org/wsdl/">
7           <xs:element name="getPhoneNumber" type="tns:getPhoneNumber"/>
8           <xs:element name="getPhoneNumberResponse" type="tns:getPhoneNumberResponse"/>
9           <xs:complexType name="getPhoneNumber">
10              <xs:sequence>
11                  <xs:element name="firstName" type="xs:string" />
12                  <xs:element name="surname" type="xs:string" />
13              </xs:sequence>
14          </xs:complexType>
15          <xs:complexType name="getPhoneNumberResponse">
16              <xs:sequence>
17                  <xs:element name="return" type="xs:string" />
18              </xs:sequence>
19          </xs:complexType>
20      </xsd:schema>
21  </types>
22
23  <interface name="PhoneDirectory">
24      <operation name="getPhoneNumber" pattern="http://www.w3.org/ns/wsdl/in-out">
25
26          <input element="tns:getPhoneNumber"/>
27          <output element="tns:getPhoneNumberResponse"/>
28      </operation>
29  </interface>
30
31  <binding xmlns:wsoap="http://www.w3.org/ns/wsdl/soap" name="PhoneDirectoryBinding"
32      interface="tns:PhoneDirectory" type="http://www.w3.org/ns/wsdl/soap"
33      wsoap:version="1.1"
34      wsoap:protocol="http://www.w3.org/2006/01/soap11/bindings/HTTP/">
35      <operation ref="tns:getPhoneNumber"
36          wsoap:mep="http://www.w3.org/2003/05/soap/mep/request-response"/>
37  </binding>
38  <service name="PhoneDirectoryService" interface="tns:PhoneDirectory">
39      <endpoint name="PhoneDirectory" binding="tns:PhoneDirectoryBinding"
40          address="http://example.com:8080/PhoneDirectoryService"/>
41  </service>
42 </description>

```

Listing 2-13 **Sample WSDL 2.0 file**

The example in Listing 2-13 shows a Web service `PhoneDirectoryService` that refers to an interface `PhoneDirectory`. The Web service is hosted at the URL

`http://example.com:8080/PhoneDirectoryService` (line 40). The interface `PhoneDirectory` (line 23) has one operation `getPhoneNumber` (line 24) which has an input message `getPhoneNumber` (line 26) and an output message `getPhoneNumberResponse` (line 27). Both of the messages refer to XML schema types defined in the *types* section (lines 9 and 15 respectively). The abstract WSDL MEP for `getPhoneNumber` is “in-out” (line 24). Also, document-literal is the style of message passing as the *style* attribute (line 25) is set to “*.../style/iri*”. The *binding* (lines 31-37) that make the abstract *interface* concrete specifies that the *interface* is bound to SOAP (line 32) over HTTP (line 34) and that the SOAP MEP that will be used to implement the abstract WSDL MEP is the SOAP “*Request-Response*” MEP (line 36).

2.6 Universal Description, Discovery and Integration (UDDI)

Before a service can be used, one must discover the service description and the service location. The service provider publishes the service description in a service registry; the service requestor finds the service description in the registry and uses the information in the description to bind to (invoke) the service. This decouples the service provider and service consumer and provides location transparency. UDDI, an OASIS standard, performs the role of service broker in Fig. 2-1 and registry in Fig. 2-2. UDDI started out as one of the core specifications in SOA and the UDDI Business Registry (UBR) was intended as a global yellow pages, where enterprises could advertise their services. In reality, however, UDDI is complex [131], difficult to use [132] and has proven to be unpopular as “*very little serious real world Web Services have used this complex system*” [108]. The main reasons for this were that from a business perspective, enterprises were uncomfortable with competitors viewing their services and technically, UDDI is viewed as having a complex data model and API [38]. In addition, UDDI contains many obsolete entries [152][153]. This has led to poor adoption and in fact, Microsoft, IBM and SAP closed their UBR in 2007 [108]. Zimmermann (a WS-* advocate) concedes that UDDI is “*not a mainstream technology*” [38] and “*has failed to reach widespread acceptance in the industry*” [101]. Tilkov goes further, stating that “*UDDI is dead...The standard isn't being extended, there is no working group, there are no new initiatives*” [39]. The fact that SOAP and WSDL are the core technologies in Web Services is confirmed in [134] and [137]. This is echoed by Kopecky *et al* : “*The major technologies for Web Services are SOAP and WSDL*” [109].

Given UDDI's issues, enterprises have used alternative methods to inform potential consumers of the WSDL file details. Enterprises can publish the WSDL file details much more simply via methods

such as a web portal supported by databases acting as service repositories [101]. In any event, UDDI is not relevant to this thesis, which is concerned with mapping the SOAP message, with the assistance of Semantic Web technologies, to RESTful HTTP format.

2.7 Quality of Service (QoS) in Web Services

QoS specifications refer to the extended features required by mission-critical applications e.g. security, reliable messaging and transactions [15]. QoS is implemented in a SOAP environment via SOAP headers. Feedback from a paper submitted to the Distributed Objects, Middleware and Applications conference (DOA 2009) directed this thesis towards Security and Reliable Messaging and consequently, they are the two QoS features discussed.

SOAP and RESTful HTTP implement these features very differently. The concern of this thesis is to realise when they are required and address them in RESTful HTTP. As a result a brief overview of these QoS features in WS-* follows.

2.7.1 Reliable Messaging

There are three aspects to message reliability [18]:

- both the sender/receiver must be certain whether or not a message was actually sent/received
- a message is sent once and only once
- received messages are in the same order that they were sent

To address this, WS-* uses the WS-ReliableMessaging standard defined by IBM, Microsoft, BEA and Tibco.

WS-ReliableMessaging (WS-RM)

This is a protocol that ensures that lost or duplicate SOAP messages are detected and received messages are processed in the order in which they were sent [18]. To achieve this, WS-RM implements client and server side datastores coupled with SOAP header correlation identifiers.

The WS-RM model is outlined in Figure 2-11:

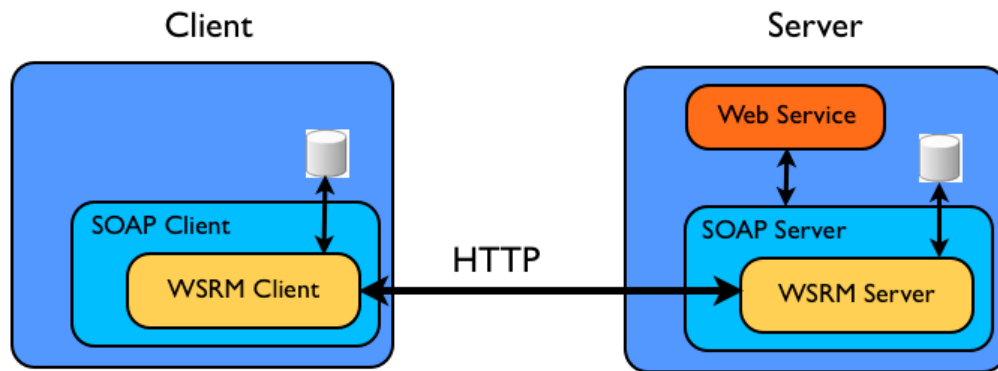


Figure 2-11 WS-RM Design [16]

WS-RM is developed around three key components [18]:

- *Sequences* – message exchanges are modelled as sequences even if there is only one message to be transmitted
- *Message numbers* – individual messages in a sequence are identified by an ascending message number; this enables missing or duplicate messages to be detected and simplifies acknowledgement generation
- *Acknowledgements* – an acknowledgement indicates that a message was successfully transferred to its destination

Listings 2-14 and 2-15 are examples of WS-RM client and server response messages respectively. Listing 2-14 demonstrates the *Sequence* element with its mandatory elements *Identifier* and *MessageNumber*. The *Identifier* element is the unique URI-style identifier for this sequence. The *MessageNumber* element numbers this message within the sequence thereby enabling the WS-RM server to order the messages in a sequence and identify the messages it has/has not received. The *LastMessage* element is sent with the last message in the sequence. Only when the server-side WS-RM receives this flag and has received all the messages in the sequence, will it execute the Web service.

```

<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm">
  <soap:Header>
    <wsrm:Sequence>
      <wsrm:Identifier>http://example.com/abc</wsrm:Identifier>
      <wsrm:MessageNumber>4</wsrm:MessageNumber>
      <wsrm:LastMessage />
    </wsrm:Sequence>
  </soap:Header>
  <soap:Body>
    <GetOrder xmlns="http://example.com/orderservice">
      ...
    </GetOrder>
  </soap:Body>
</soap:Envelope>

```

Listing 2-14 Sample WSRM client message [18]

Listing 2-15 is a sample WS-RM server response that demonstrates the *SequenceAcknowledgement* element. The example in Listing 2-15 states, using the *AcknowledgementRange* and *Nack* elements, that while messages 1, 2 and 4 have been successfully received, message number 3 has not been received. This response can be used by the client to re-send message number 3.

```

<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm">
  <soap:Header>
    <wsrm:SequenceAcknowledgement>
      <wsrm:Identifier>http://example.com/abc</wsrm:Identifier>
      <wsrm:AcknowledgementRange Lower="1" Upper="2" />
      <wsrm:AcknowledgementRange Lower="4" Upper="4" />
      <wsrm:Nack>3</wsrm:Nack>
    </wsrm:SequenceAcknowledgement>
  </soap:Header>
  <soap:Body>
    ...
  </soap:Body>
</soap:Envelope>

```

Listing 2-15 Sample WSRM server response [18]

2.7.2 Security

Sending data over an insecure network such as the Internet poses security risks such as messages being stolen, lost or modified. The following security requirements address this:

- confidentiality ensures that sensitive data such as credit card numbers are not stolen
- integrity ensures that a message has not been modified in transit
- authentication ensures that access is only given to those who can prove their identity
- nonrepudiation means that a sender cannot deny having sent a message
- authorisation means that access is only given to what one has access to

The first four requirements affect the message content i.e. headers related to these requirements are included in the SOAP message. The last requirement, authorisation, is often implemented by a Web server or an application server using for example, access control lists. These access control lists are userid or group specific i.e. your userid or the group to which you belong, determine your level of access, if any, to protected resources.

Conventional security measures such as firewalls provide a secure layer between the private, trusted network and the external, untrusted network. However, firewalls operate at the network layer and consequently are unable to provide the application layer security that Web services require. In any event, Web services are designed to go through conventional security measures such as firewalls, by using port 80. In addition, Web services are standardised, with SOAP providing the structure of a message and WSDL providing the endpoint location, the interface and the supported protocol. This is significant information to a potential intruder [18]. A general discussion on security topics follows.

Symmetric encryption

Symmetric encryption uses a single key that both the sender and receiver possess. The sender encrypts the plaintext message with the key to create an encrypted message called ciphertext. The receiver decrypts the ciphertext with the same (i.e. symmetric) key to obtain the plaintext message. Symmetric encryption is fast and easy to implement. Its weakness is that every pair of users' needs a unique key (otherwise everyone can read all messages), resulting in a large number of keys.

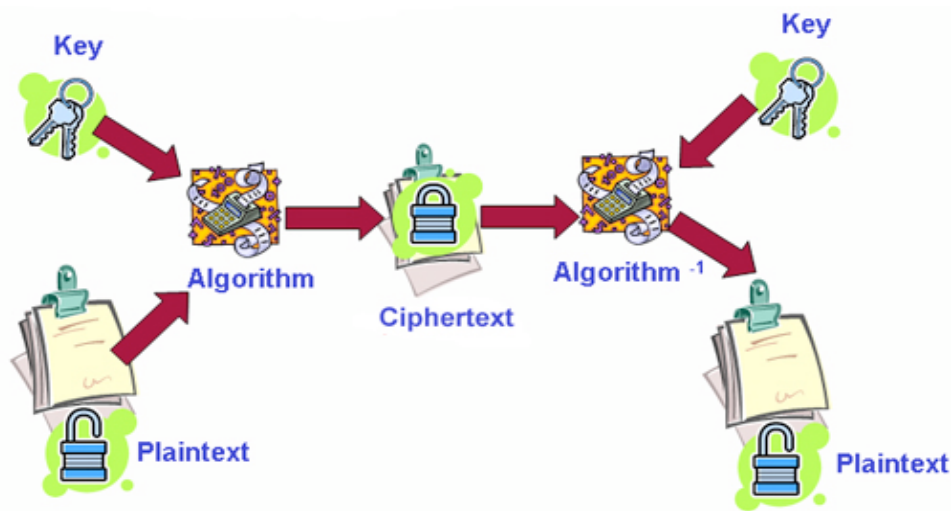


Figure 2-12 Symmetric key encryption

Asymmetric encryption

Asymmetric encryption uses two separate keys, one for encryption and the other for decryption. The key pair are created at the same time and share a mathematical property that if a message is encrypted with one key, only the other key can decrypt the message i.e. the original key cannot decrypt it [18]. A recipient publishes its public key but keeps its private key private. The sender encrypts a message with the recipient's public key and the recipient can decrypt the message with its private key. Asymmetric encryption suffers from performance [18] and cannot guarantee the sender's authenticity, as anybody can use the recipient's public key.

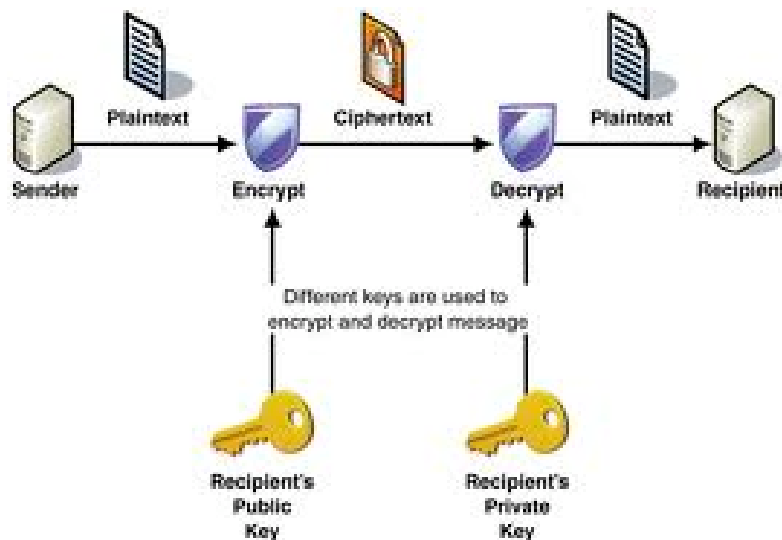


Figure 2-13 Asymmetric key encryption

Digital certificates/signatures

To guarantee a sender's authenticity (identity), digital certificates can be used. A digital certificate is a document that binds an enterprise to its public key. They are issued by third parties known as Certificate Authorities (CA), in an infrastructure known as Public Key Infrastructure (PKI). The CA (Certificate Authority) guarantees that the public key listed in the certificate belongs to the party identified in the certificate [18].

To address the performance concerns of asymmetric encryption, digital signatures can be used. A digital signature is not involved in data encryption but is used for ensuring message integrity and non-repudiation [18]. To create a digital signature a sender encrypts a "digest" of the original message with its private key. A message digest is a code, which is usually much smaller than the original message but nonetheless unique to it. Fig. 2-14 outlines how this works. Firstly, the sender calculates the message digest using a hashing (digest) algorithm. The message digest is encrypted with the sender's private key giving the digital signature. The plaintext message, along with the digital signature is sent to the receiver. The receiver, using the plaintext message and the same hashing algorithm re-calculates the message digest. The receiver also decrypts the digital signature using the sender's public key (which is on the digital certificate that the sender attached to the message) and then compares the two message digests. If they are different then the message has been altered otherwise the integrity of the message has been proven [18].

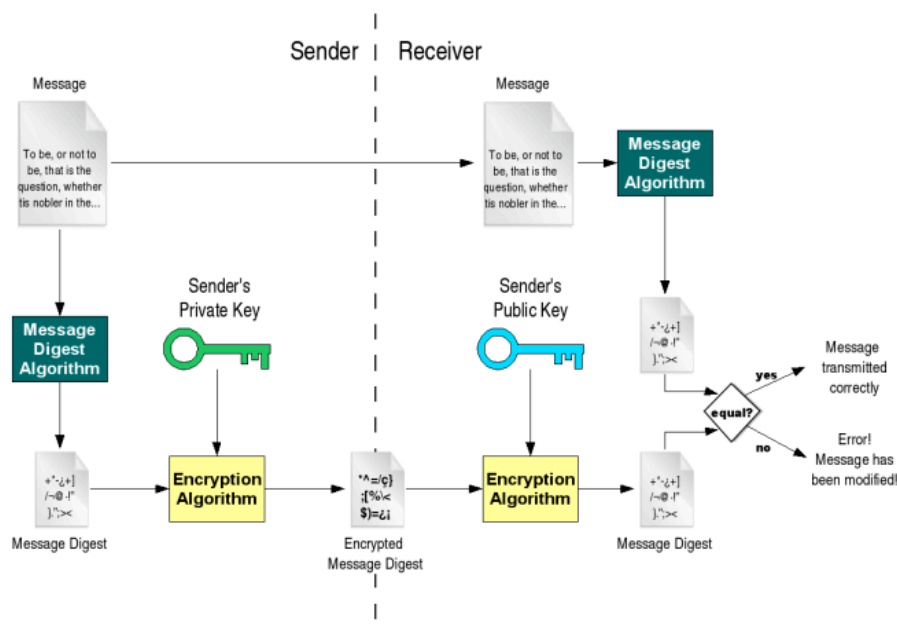


Figure 2-14 Digital Signature

Web Services Security

SOAP provides end-to-end message security using the WS-Security suite of standards. Using SOAP headers, it offers a comprehensive solution that involves significant performance overhead (of course there is nothing preventing SOAP from using HTTP's own point-to-point transport layer security, namely HTTPS). In fact, such is the performance penalty with the WS-Security suite of standards that:

- (a) one can expect to use hardware accelerators [38] and
- (b) as advised by O hEigartaigh in [41], if HTTPS is an option then use HTTPS.

WS-Security

WS-Security is the foundation specification for security in Web services. WS-Security defines a format for including:

- security tokens e.g. a username/password pair or a digital certificate.
- digital signatures via W3C's XML Signature specification.
- encryption via W3C's XML Encryption specification.

The format of a WS-Security message is detailed in Listing 2-16:

```
<soap:Envelope xmlns:S=http://www.w3.org/2003/05/soap-envelope>
  <soap:Header>
    <wsse:Security xmlns:wsse=http://schemas.xmlsoap.org/ws/2003/06/secext>
      <Signature xmlns=http://www.w3.org/2000/09/xmldsig#>
        ...
      </Signature>
      <EncryptedKey xmlns=http://www.w3.org/2001/04/xmlenc#>
        ...
      </EncryptedKey>
      <wsse:UsernameToken xmlns=http://schemas.xmlsoap.org/ws/2003/06/secext>
        ...
      </wsse:UsernameToken>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    ...
  </soap:Body>
</soap:Envelope>
```



Listing 2-16 Sample WS-Security document

The generic XML specifications XML Encryption and XML Signature provide confidentiality and integrity respectively. In addition XML Signature provides nonrepudiable evidence of who created

the document [148]. Neither specification defines new cryptographic algorithms; rather they define how to apply existing algorithms to XML.

The XML Signature specification defines the parent `Signature` element and its children. WS-Security defines how to embed the `Signature` element in a SOAP header. Within the `Signature` subtree, one will find [16]:

- a reference (the `Reference` element) to the section of the document to be used in calculating the digest e.g. the `soap:Body` section.
- the digest value and the algorithm used to calculate it (`DigestValue` and `DigestMethod` elements respectively)
- the digital signature itself and the algorithm used to calculate it (`SignatureValue` and `SignatureMethod` elements respectively)

The XML Encryption specification enables encryption of all or various elements of XML documents. Initially, a shared secret symmetric key is generated. This symmetric key is used to encrypt the data. To ensure the symmetric key remains secure it is encrypted with the receiver's public key i.e. only the receiver's private key can be used to gain access to the symmetric key. Thus, only the receiver can gain access to the symmetric key and decrypt the message. WS-Security defines how XML Encryption is incorporated into SOAP headers. The `EncryptedKey` element is a directive to message receivers detailing which sections are encrypted and how to decrypt them. The `EncryptedKey` element contains the following elements [16]:

- the `EncryptedData` element points to the encrypted section e.g. `soap:Body`
- the `EncryptionMethod` element informs the receiver of the encryption algorithm used
- the `CipherData` element contains the encrypted symmetric key
- the `KeyIdentifier` element contains the public key

The `UsernameToken` element represents a security token i.e. a requestor's claims. This is an XML representation of a username/password pair (as in Listing 2-16). A digital certificate would be represented by the `BinarySecurityToken` element and is used for implementing authentication.

2.8 Plain Old XML (POX)

POX is defined by Microsoft as “*messages that consist solely of XML payloads without any enclosing SOAP envelope*” [42]. POX is a lighter approach to the WS-* approach and is attractive to enterprises that “*exchange data over HTTP and have no requirement to use the advanced protocol capabilities of SOAP and WS-* such as non-HTTP transports, message exchange patterns other than request/response, and message-based security, reliability, and transactions*” [42]. POX relies purely on XML Schema i.e. POX has no WSDL file.

Its relevance to this thesis is that POX is an alternative paradigm to implementing Web Services. Many POX implementations follow the SOAP model i.e. the POX messages are POSTed to a gateway URI where the message is parsed and the identified Web service invoked. A sample POX file from industry is shown in Listing 2-17. Note that there is no SOAP content.

```
<?xml version="1.0" encoding="UTF-8"?>
<Request>
  <Log>N</Log>
  <ID>
    <version>1.0</version>
    <AppID>NBP</AppID>
    <AppName>null</AppName>
    <UsrID>5324654654</UsrID>
    <UnqID>83480</UnqID>
  </ID>
  <regionCode>ROI</regionCode>
  <sourceNSC>931012</sourceNSC>
  <staffNumber>83480</staffNumber>
  <deviceId/>
  <Transaction>CltviewService002</Transaction>
  <CltviewService002>
    <ICLOSPC_NAMEI index="1">JOHN DALY</ICLOSPC_NAMEI>
    <ICLOSPC_CNTYCDI index="1">DN</ICLOSPC_CNTYCDI>
    <ICLOSPC_TYPI index="1">P</ICLOSPC_TYPI>
  </CltviewService002>
</Request>
```

Listing 2-17 Sample POX file [157]

2.9 HTTP Tunnelling

“Though it is often mistaken for a transport protocol, HTTP is really an application protocol. Those seeking to exploit HTTP’s ubiquity – to transport SOAP over it, for instance – tend to focus on its ability to transport other protocols, an act commonly called tunnelling” [43]. This is achieved by using HTTP POST to transport the other protocol inside the entity-body of the message. Typically there is only one gateway URI used, where the XML is parsed to elicit the Web Service to execute and its associated parameters. Both SOAP and POX both use HTTP in this fashion.

HTTP provides a generic application interface with defined semantics via its verbs: GET, PUT, POST and DELETE. Tunnelling other protocols through HTTP *“is an abuse of the protocol if the tunnelling does not respect those semantics”* [43]. This is not how HTTP was intended to be used and can result in the disabling of Web intermediaries such as proxies or caches.

As outlined in section 2.4.8, SOAP 1.2 attempted to address this issue by enabling SOAP usage without violating REST principles [137]. SOAP 1.2 introduced a new SOAP Response MEP, a new WebMethod feature and modified the SOAP/HTTP binding to support both POST and GET. However, the new features of SOAP 1.2 are not widely used [26][25]. This was suspected in [137]: *“the problem will be in the adoption process – for there is nothing that prevents the non-RESTful use of the SOAP standard”*. In any event, the MEP of direct interest to this thesis is the SOAP Request-Response MEP i.e. where POST is used to send the SOAP request. This MEP is the one used by SOAP 1.1.

2.10 Web Services Interoperability

In similar research, Briggs outlines a design, which, from a high level, is similar in appearance to the design proposed in this thesis (a RESTful back end with a SOAP front end) [8]. Briggs’ paper was written in 2006 and the motivation for the architecture was to introduce RESTful techniques in a SOAP-based culture. In fact, Briggs states that the article is aimed at those *“wanting to design REST systems, but stuck [sic] delivering SOAP messages because it’s the standard, and what everyone expects you to deliver”*. To do this, Briggs suggests implementing the backend system using REST principles and then plugging in a SOAP wrapper (on the server) to enable SOAP clients to access the RESTful backend (see Figure 2-15). For example, an existing *order* resource that supports GET is

mapped to a new *GetOrder* SOAP interface. In short, Briggs is SOAP enabling RESTful Web Services.

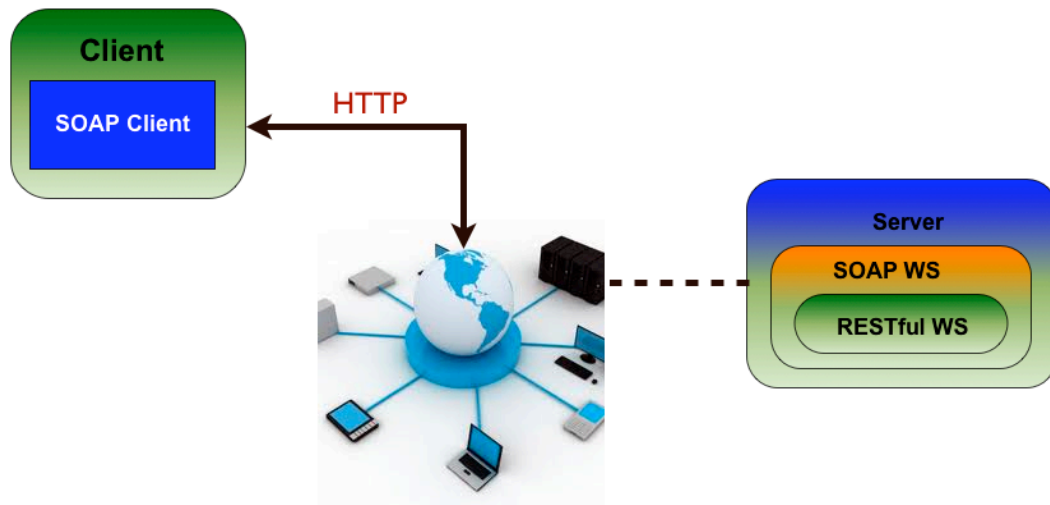


Figure 2-15 SOAP-REST Web Services interoperability

As outlined in the Introduction, the major differences between Briggs' model and the model in this thesis are:

- Briggs puts a SOAP wrapper around the RESTful Web Services. The approach outlined in this thesis is to transform the SOAP messages to RESTful HTTP format.
- Briggs' adapter is on the server and thus the on-the-wire client messages are SOAP based. The adapter presented in this thesis resides on the client and the on-the-wire client messages are RESTful HTTP based.
- Briggs does not take the existing SOAP interface into account whereas the adapter outlined in this thesis does, leading to a seamless migration tool for transitioning SOAP clients to RESTful HTTP format.
- Briggs does not include POX WS or leverage the Semantic Web whereas the architecture outlined in this thesis does both.

2.11 Summary

In this section, SOA and one of its most popular implementation paradigms, WS-* Web Services, was discussed. The most important standards in the WS-* stack, namely SOAP and WSDL [121], were covered in detail. Quality of Service, from the perspective of Security and Message Reliability

was outlined. An alternative Web Services implementation model, Plain Old XML (POX) was also discussed. The similarities of SOAP and POX Web Services were outlined from the perspective of HTTP tunnelling. This is a core issue in this thesis. Addressing this issue formulates the requirements of the adapter to be detailed later. Lastly, Briggs Web Services interoperability model was outlined and contrasted with the model in this thesis.

The next chapter will focus on Representational State Transfer (REST), the architectural style behind the Web : *“the largest and most successful distributed system in existence”* [43].

Chapter 3

Representational State Transfer (REST)

3.1 Introduction

In the previous chapter, SOA and WS-* Web Services were outlined. In this chapter, an overview of an alternative methodology for implementing Web Services, based on the REST architectural style [111] is given. As REST is “*an abstract model of the Web architecture*” [112], the Web itself i.e. HTTP, URI and Media Types [44] is also outlined.

3.1.1 Background

The initial version of REST in late 1995 was used as a means of communicating Web concepts while developing HTTP 1.0 and the initial HTTP 1.1 proposal. Over the next 5 years REST was iteratively improved as a model of how the Web *should work*. The REST architectural style informed the Internet Engineering Taskforce on flaws in the existing Web and helped validate extensions prior to

deployment e.g. HTTP 1.1 (1996) and URI (1998) [112]. As REST was published in 2000 [3], it is in some respects “*a retrospective abstracting of the principles that make the World Wide Web scalable*” [137]. In [124], Fielding discusses the possibilities of the Web moving from just a barrier remover for personal publishing to a platform for virtual enterprises (“*an organisation unconstrained by geographic location*” [124]).

Web Services that conform to REST principles are known as “RESTful Web Services” and have “*long been championed as a competing Web service paradigm to the WS-* stack*” [125]. There are a variety of reasons for the increase in RESTful WS popularity:

- RESTful WS offer a more lightweight and simpler alternative to the more complex and heavyweight WS-* Web Services [121][154]. Even though the goals of RESTful WS and WS-* WS are similar i.e. interoperability for distributed applications, RESTful WS achieve this in a “*more lightweight manner seamlessly integrated with the Web*” [131]. In contrast, the specifications in the WS-* stack are complex, numerous and daunting [101][127].
- RESTful WS are based on open Web standards such as HTTP, URI and XML. There is no risk of vendor lock-in and competing, duplicate or overlapping specifications [127]. In addition, the infrastructure is ubiquitous i.e. HTTP clients and servers are available for all major programming languages and operating systems/hardware platforms and the default HTTP port (port 80) is left open by default in most firewall configurations [101].
- REST’s Uniform Interface constraint stipulates that messages must be self-describing and therefore visible to intermediaries. This enables critical distributed systems features such as proxying, caching and intermediation [12].
- RESTful WS representation formats are decoupled from their resources i.e. a resource is free to return any MIME type. This enables RESTful WS clients to use content negotiation to request their preferred representation format. As Vinoski states in [7], “*the ability to have formats other than XML is huge*”.

However, RESTful WS are not without their critics. In [101] Pautasso et al stated that WS-* Web Services are to be preferred “*for professional enterprise application integration scenarios, with a longer lifespan and advanced QoS requirements*”. The authors highlighted the following weaknesses:

- RESTful WS use HTTPS, which is a point-to-point SSL (Secure Sockets Layer) protocol [101]. This may not be suitable in a Web Services context where multiple intermediary nodes may exist between two communicating endpoints. In addition, SSL secures messages at the transport level and not the message level i.e. messages are only protected while they are being transmitted. This means that sensitive data residing on someone’s machine is unprotected (unless some proprietary encryption is used) [108].

- Even though there is no URI length limit in the HTTP specification, servers impose their own limits e.g. Apache imposes an 8Kb limit [45]. The HTTP verbs GET and DELETE have no entity bodies to carry data so the arguments to be passed are encoded in the URI. If a requirement arises where a large amount of data must be passed, the URI (for a GET or DELETE) may become too long and lead to malformed URIs [101]. In addition, some proxies and firewalls support only GET and POST connections; refusing to handle PUT or DELETE connections [101][154]. This has led to workarounds where the “real” verb is either passed via special HTTP headers e.g. X-HTTP-Method-Override [101] or appended at the end of the URL [154].
- In [101], the authors also pointed out the lack of standards regarding advanced requirements in enterprise computing i.e. Quality of Service (QoS). This lack of standards results in ad-hoc solutions [101][125]. With regard to message reliability, options available to RESTful WS are Post Once Exactly (POE) [56], HTTPLR [57] and Joe Gregorio’s common best practice approach [58]. POE is an IETF draft that expired in 2005 and HTTPLR is a pre-draft memo. Gregorio’s approach outlined in [58] is the common best practice. None of these approaches have been standardised. In addition, the lack of a framework for distributed transactions was also noted [101].

Given the Web’s influence in today’s society, and the fact that Fielding’s thesis [3] is the definitive reference point for REST [26], Fielding’s thesis is regarded in the Web field as seminal. REST “*lies at the heart of the Web*” [131] and has influenced many authors, for example:

- In [44], Allamaraju has authored a cookbook for designers and developers of RESTful Web Services. In the book, the author states that REST “*describes the foundation of the World Wide Web*” [44].
- Ruby and Richardson’s book “*RESTful Web Services*” [45] is regarded in the REST community as very important. It was the first book on the topic and explained the principles behind RESTful Web Services. The authors also define a Resource Oriented Architecture (ROA) that is based on REST. In essence, whereas REST is abstract and does not mandate a specific protocol, interface or technologies, ROA (Resource Oriented Architecture) mandates the use of HTTP, its interface GET, PUT, POST and DELETE, and the use of URI’s and XML.
- The Java API for RESTful Web Services (JSR-311) [126] was created in 2007 in response to the growing popularity of RESTful Web Services.
- In October 2011, Prentice Hall are publishing a new book entitled “*SOA with REST: Principles, Patterns and Constraints*” by Pautasso et al. Patterns such as content negotiation

and response caching will be covered.

- WS-REST (an international workshop on RESTful Design) was established in 2010 and brings together academics and enterprise practitioners to share research and ideas in the area.
- In academia, REST has received attention in the “Web of Things” [134][135] where its simplicity, lightweightness and ubiquitous aspects are attractive for integrating the physical world of devices (e.g. wireless sensors) with the virtual world of software systems. In [134], HTTP servers are embedded directly onto the sensor nodes enabling REST APIs (verbs on resource URIs) to be used to read temperatures on the sensor nodes. In [135], sensor networks and the Internet are integrated via a “TinyREST” protocol. This is achieved by implementing an HTTP-2-TinyREST gateway which maps HTTP requests to native sensor node calls i.e. TinyOS operating system calls.
- In [123], Khare extends REST to define new architectural styles enabling features such as asynchronous event notification and mutually exclusive access to resources for decentralised systems i.e. independent agencies that can make their own decisions.
- In [125], RETRO, a RESTful Transaction Model is outlined to address one of the criticisms levelled against REST i.e. transactions [38][101]. In effect, Web applications (running over HTTP, which is based on REST) have to resort to ad-hoc solutions to address this need [125]. RETRO is a model that attempts to address this.
- The evolution of Web 2.0 has led to the increased adoption of RESTful WS. Using XML messaging, discrete data from disparate RESTful services can be integrated together into one service. This is referred to as building a mashup i.e. the creation of a new service from two or more existing services. In other words, a mashup is a composition of RESTful services [83]. To build a mashup, a services output is chained to another services input with necessary format changes. Tools exist to simplify this process e.g. Yahoo! Pipes and Google Mashup Editor [83]. In [150], the issue of novice Web users requiring expertise with mashup editors is addressed. The authors propose a Mashup Services System (MSS) framework that contains a Graphical User Interface (GUI) front end where “*users only need to declare their needs*”. MSS, executing queries over a service registry, builds a mashup that reflects their requirements. The process is automated by the use of Semantic models i.e. the service descriptions refer to a domain ontology.

3.2 REST

In this section, REST is explained with the aid of an example. In addition, terminology that is

important to understanding REST is outlined.

3.2.1 Definition

The REST architectural style was defined by Fielding in this doctoral thesis “*Architectural Styles and the Design of Network-based Software Architectures*” [3]. An architectural style is a named set of constraints, applied to elements and the relationships between those elements, for any architecture conforming to that style [3][112]. A style provides a name by which we can refer to a set of architectural decisions and the properties induced by applying the style. Constraints are chosen for the properties they induce [112].

Fielding describes REST as follows: “*The name Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use*” [3]. Thus, REST is an architectural style for building distributed hypermedia systems (systems linked by hyperlinks).

Figure 3-1 illustrates an example.

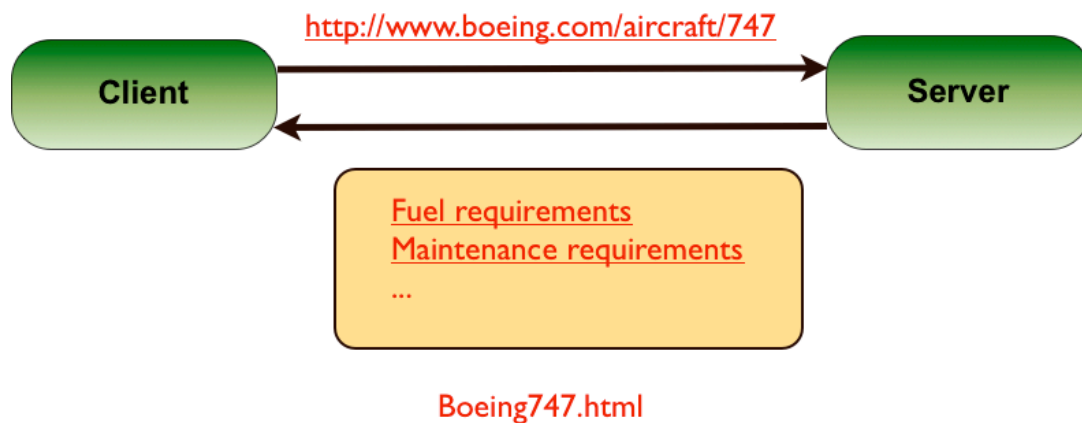


Figure 3-1 REST [5]

Costello explains Fig. 3-1 as follows, “*the Client references a Web resource using a URL. A **representation** of the resource is returned (in this case as an HTML document). The representation (e.g., Boeing747.html) places the client application in a **state**. The result of the client traversing a hyperlink in Boeing747.html is that another resource is accessed. The new representation places the client application into yet another state. Thus, the client application changes (**transfers**) state with each resource representation, hence Representational State Transfer*” [5]. A resource in REST is an

abstract concept, which is made concrete via a representation of that resource. For example, note that in Fig. 3-1 that the resource being accessed is logical, and not a physical object i.e. the URI finishes with “747” as opposed to “747.html”. The URI accessed is the abstract resource and what is returned is the HTML representation of that resource i.e. Boeing747.html. This decouples the implementation of the resource from the clients [5] and allows a server to handle millions or even billions of URI’s [25].

3.2.2 REST nomenclature

In order to understand the REST architectural style, an explanation of REST terminology is appropriate. As is shown in Fig. 3-2, REST resources are manipulated via the transfer of representations of those resources.

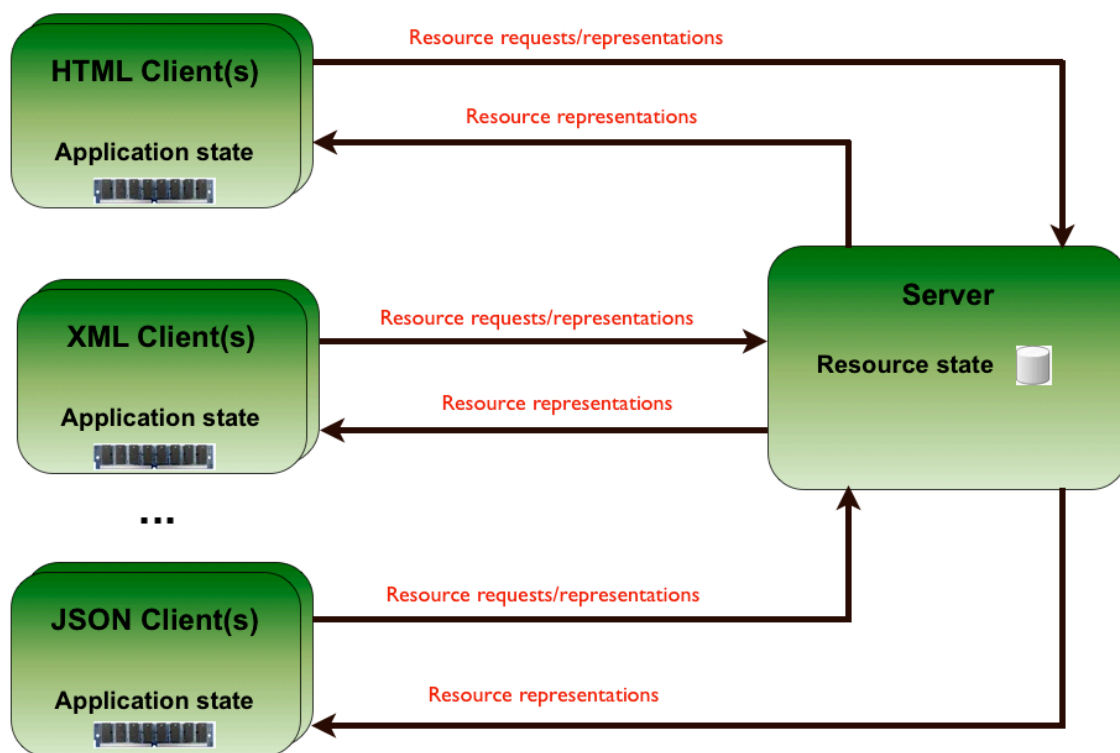


Figure 3-2 Terminology used in REST

Resources

REST refers to information abstractly as “resources”. *“A resource can be anything a client might want to link to”* [45]. Examples include *“today’s weather at Dublin airport”* and *“version 1.1 of the software”*. A resource has an identifier and a representation. For example, on the Web, one uses a URI to identify a resource, which may have a HTML representation. The representation can change over time e.g. *“the authors preferred version”* of an academic paper or can remain static e.g. *“the paper published in the proceedings of conference X”*.

Representations

Actions are performed on resources by transferring representations that capture the current or intended state of the resources [112]. A representation consists of data and metadata describing the data. The data represents the physical version of the abstract resource e.g. an HTML representation of the resource “*Sky News home page*”. The data format of a representation is known as a media type [3][112]. Media types are well known standard types maintained by the Internet Assigned Numbers Authority (IANA) [46]. The metadata is in the form of name-value header pairs that inform intermediaries of, for example, the media type contained in the message, when the resource was last changed etc..

Resource State

Resource state is information about resources that is the same for all clients. It resides on the server (in databases typically) and is sent to the client via representations. An example would be a Flickr [47] image resource, which resides on a server available to everybody.

Application State

Application state is information about the path taken by a client through an application and is therefore client-specific i.e. application state is different for each client. Application state resides on the client. Examples of application state stored locally by a browser during a Google search would be the *Next*, *Previous* links and the page *1..n* links. The previous pages visited by the user are also stored as application state. This enables users to go back through their browsing history (i.e. their previous application states) via the *Back* button.

3.2.3 Deriving REST

REST is a hybrid style derived from other network-based architectural styles. In his thesis, Fielding defines an architectural style as a “*coordinated set of architectural constraints that has been given a name for ease of reference*” [3]. The use of an architectural style applies the associated constraints on the system. Each constraint induces certain desired properties e.g. simplicity and scalability. Thus, a style applies (its) constraints, which induce certain desired properties.

Fielding defines the properties of key interest when considering the target architecture of network-based hypermedia systems. For example: scalability, simplicity, visibility and independent evolvability. Fielding evaluates several common network-based architectural styles (e.g. client-server) for the properties they would induce. Fielding then derives REST by applying the styles that induce the properties he requires. To do this, Fielding firstly defines the “null” style i.e. a style with

no constraints at all. Fielding then adds certain pre-defined styles, which induce the desired properties for the target architecture of network-based hypermedia. This hybrid style is combined with other constraints (most notably the uniform interface constraint) to form the REST architectural style (see Fig. 3-3).

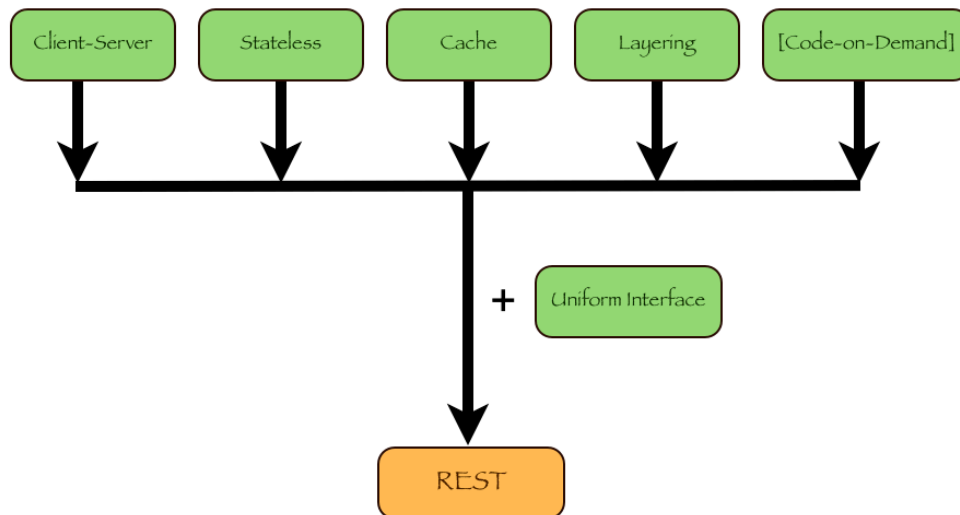


Figure 3-3 Deriving REST

Client-Server style (CS)

This is the first style to be added to the “null” style. The software engineering principle guiding this style is the “separation of concerns” principle. The separation of concerns is with respect to the “*allocation of functionality within components*” [3] e.g. separating GUI and database concerns. This separation of concerns is the primary method by which architectural styles induce simplicity [3]. REST separates the interface from both the implementation and the data format (the resource representation) [48]. This simplifies the server components, improving scalability [112]. This separation of concerns also enables components to evolve independently, an important consideration given the scale of the Web and the large number of enterprises it hosts [3].

Stateless constraint (CSS = Client-Stateless-Server)

The stateless constraint is added to the client-server interaction and constrains that each client request is independent of any other request i.e. “*each request from client to server must contain all the information necessary to understand the request*” [112]. The server considers each request in isolation, bearing in mind the current resource state. The server (except for resource state), must not rely on information from any previous requests. This is why a bookmark in a browser works regardless of what page you are currently viewing now. If the client wants any application state to be taken into consideration, the client must submit it as part of the request [45]. For example, when a

user clicks a browser's *Next* link on a list of search results, the application state representing that "next page" is sent along with the request. Statelessness does not mean that you cannot have state persisted in a database on a server – this is the resource state outlined earlier. "*A RESTful service is 'stateless' if the server never stores any application state*" [45].

The stateless constraint induces the properties of visibility and scalability. Visibility is improved because an intermediary can inspect and understand a message in complete isolation. This is because all the information necessary to understand the message is present in the message. For example, a cache can decide whether or not to cache a response by looking at that one response on its own i.e. there is no concern that a previous response may affect the cacheability of this one. As no two requests depend on each other, different servers can handle the requests i.e. it does not matter which server you talk to (as long as the resource state exists on the server). Thus, improving scalability is simply a matter as adding extra servers to a load balancer [45].

Cache constraint (C\$SS = Client-Cache-Stateless-Server)

The next constraint added is the cache constraint. This constraint is added to improve network efficiency. The cache constraint specifies that a response to a request can be cached so as to satisfy subsequent equivalent requests. Caches can be inserted at any point along the message path. Caches improve network efficiency (as the request may not travel all the way to the origin server) and scalability (a shared cache allows the introduction of extra client components).

Layered System (LC\$SS = Layered-Client-Cache-Stateless-Server)

A layered system is organised hierarchically such that each layer provides services to the layer above it and uses services from the layer below it e.g. the TCP/IP model. Each layer is constrained from seeing beyond the layer with which it is communicating thereby placing a bound on the overall system complexity [112]. Thus, an application can interact with a resource without knowing if or how many intermediaries exist between it and the server actually holding the information. A layered client-server model adds components such as proxies and firewalls to the client-server style, thereby adding features such as caching and security checking to systems.

Code-On-Demand (LCODC\$SS = Layered-Code-On-Demand-Client-Cache-Stateless-Server)

With this style, a client downloads code from a server and executes it locally e.g. applets or scripts. Extensibility is improved because features can be added to pre-implemented clients. Clients are simplified as the number of pre-implemented features can be reduced. The most significant limitation is the lack of visibility as the server is sending code not simple data and thus it is only an optional constraint in REST [3].

Uniform Interface (REST = LCODCSSS + Uniform Interface)

The last constraint added to define REST is the uniform interface constraint (see Fig. 3-4). The uniform interface is the most differentiating of REST's constraints [3][112]. Generality and information hiding are the software engineering principles guiding this constraint. The generality principle encourages simplicity whereas the information hiding principle ensures that implementations are decoupled from the interface thereby encouraging independent evolvability.

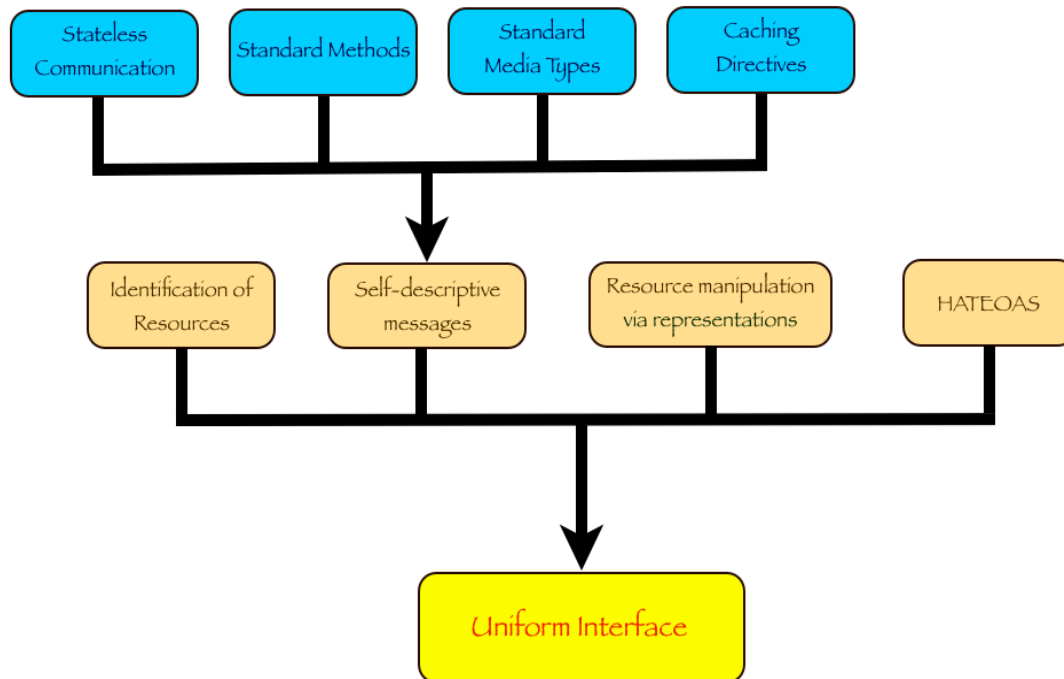


Figure 3-4 Uniform Interface

The uniform interface constraint enables visibility into component interactions and is obtained with the support of the following sub-constraints [3][112]:

- 1) Identification of resources
- 2) Self-descriptive messages
- 3) Manipulation of resources through representations
- 4) Hypermedia as the engine of application state

Identification of Resources

All resources that merit identification get an identifier. On the Web, “resources are named with URI’s” [49] and as URI’s make up a global namespace, using URI’s to identify your resources means they get a unique, global ID. This is how the online store Amazon.com operates: every one of its products gets a unique ID (a URI). Resources are abstract and can relate to anything one merits as being identifiable, for example:

- an individual item e.g. <http://example.com/customers/1234>
- a collection of items e.g. <http://example.com/customers>
- a process step e.g. <http://example.com/processes/salary-increase-123>
- a non-virtual object e.g. <http://example.com/people/john>

Self-descriptive messages

“REST constrains messages between components to be self-descriptive in order to support intermediate processing of interactions” [3]. This means that the message semantics are visible without looking at the message body [50] enabling intermediaries (e.g. proxies and caches) to inspect the message and act accordingly. This constraint enables *“critical distributed systems concepts such as proxying, caching, intermediation and monitoring”* [12]. For a message to be self-descriptive requires the following [3]:

- stateless communication: each request contains all the information necessary to understand the request, independent of any previous request
- use standard methods: in HTTP, the primary methods are GET, PUT, POST and DELETE. These methods have well defined semantics as per the HTTP specification [51]. For example, a proxy understands that a HTTP GET is a retrieval message.
- standard media types: the format of the information exchanged by components is standard and well-known e.g. on the Web, one has the Internet media types [46].
- response messages that are to be cached must be marked accordingly

Manipulation of resources through representations

Resources are manipulated via representations in the format of media types. If a resource supports more than one representation, the client can use content negotiation to select the most appropriate one. For example, in HTTP, if a resource supports both HTML and XML representations then both browser and application-based clients can be supported. A browser client can issue a GET with the **Accept** header set to “text/html”; whereas an XML application client can issue a GET with the **Accept** header set to “text/xml”. The **Content-Type** header on the response will depend on the representation contained in the message.

Clients can send content to create/update the resource state by sending the resource representation with the message. For example, to update a resource in HTTP, a client PUTs the new representation type identified by the **Content-Type** header e.g. text/xml, to the resource identified by the URI.

Hypermedia as the engine of application state (HATEOAS)

This sub-constraint is based on the idea of hypermedia i.e. links. *“HATEOAS is a defining characteristic of the REST architectural style. State transitions in an application occur when moving*

from one resource to another using the links provided in representations” [103] In practical terms it means that resources return representations that contain URI’s so that clients can navigate from them to other resources [136]. Given that the Web is a RESTful application it helps to refer to it when explaining this sub-constraint. When one executes a Web application, for example visiting *www.nuigalway.ie*, one receives back HTML, images and **hyperlinks**. At this point, we are in the initial state of a virtual state machine representing the Web application. By clicking on a hyperlink on the page we execute a state transition whereby a new resource representation (a HTML page), is transferred to the browser. This reflects our transfer into a new application state. Thus, we move between application states in the virtual state machine by clicking on hyperlinks.

Listing 3-1 is an example file demonstrating how HATEOAS should be supported in a programmatic client i.e. a non-browser based client. Firstly, note that the **ref** attribute contains a link. However this link is not just some application-specific identifier: it is a URI. As URI’s are global standards, all the resources on the Web can be linked to each other i.e. using URI’s as links means that the links can point to resources provided by a different application on a different server anywhere in the world [52]. Note that it is the server that is guiding the client through the application because it is the server that builds the representations containing the links that guide the client through the application state.

```
<order self='http://example.com/customers/1234' >
  <amount>23</amount>
  <product ref='http://example.com/products/4554' />
  <customer ref='http://example.com/customers/1234' />
</order>
```

Listing 3-1 HATEOAS example [52]

If implemented properly, a client need only be aware of the entry-point URI; the other URI’s will be in the resource representations [48]. This is similar to a Web site visit i.e. one enters the first URL in a browser and follows hyperlinks from that point on.

3.2.4 Summary

REST is an abstract architectural style that places constraints on the elements within the architecture. It was defined for optimal performance for the common case on the Web i.e. large-grain (large amounts) data transfer [3]. Fielding identified other architectural styles that demonstrated properties

desirable in the Web. He amalgamated these styles together to obtain a hybrid style to which he added a uniform interface constraint: “*to form a new architectural style that better reflects the desired properties of a modern Web architecture*” [3]. This new style he called REST. Table 3-1 outlines the most important properties induced by the REST style and their origin.

Style/Properties	Simplicity	Scalability	Independent Evolvability	Visibility	Extensibility	Efficiency
Client-Server	✓	✓	✓			
+ Stateless		✓		✓		
+ Cache		✓				✓
+ Layering	✓	✓				
+ [Code-on-Demand]					✓	
+ Uniform Interface	✓	✓	✓	✓		
REST	✓	✓	✓	✓	✓	✓

Table 3-1 Key properties induced by REST

REST is an architectural style and as such it is abstract. The best-known implementation of REST principles today is the Web. The Web is the topic of the next section.

3.3 The Web

The Web consists of many technologies; the core technologies being HTTP, URI, HTML and the Internet Media Types [46][44]. In this section the topics detailed are HTTP, URI and media types as they have direct relevance to REST.

3.3.1 HyperText Transfer Protocol (HTTP)

HTTP is a text-based application protocol that has ubiquitous deployment on the Web. It is the Web’s primary information transfer protocol and is based on a client/server model i.e. the server waits for client requests, locates the requested resource, applies the method requested to that resource and returns the response back to the client [124]. The latest version is 1.1 [51]. For the purposes of this thesis, it is necessary to define “RESTful HTTP” as “*using HTTP as intended and as described by the REST principles*” [4].

Request

When creating a client request, a client creates a message that contains a request line, followed by some message headers, a blank line and an optional entity body. Figure 3-5 outlines the overall format of the request and Listing 3-2 details an example request.

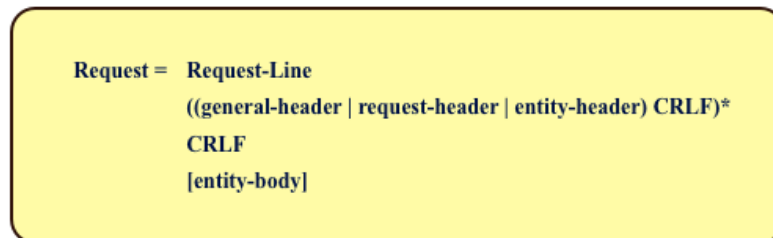


Figure 3-5 HTTP request message format [53]

The diagram shows a sample HTTP request message. It is enclosed in a yellow rounded rectangle with a dark border. The text inside is as follows:

```
GET /PartsServer/resources/parts/200 HTTP/1.1
Host: 127.0.0.1:8083
User-Agent: Noelios-Restlet-Engine/1.1.2
Accept: text/xml
Connection: close
```

Listing 3-2 A sample HTTP request message

Fig. 3-5 states that a request is a request line followed by some headers, each on a separate line in the form of name-value pairs, followed by a blank line and an optional entity body. CRLF stands for carriage return and line feed i.e. `/r/n` [53]. The request line details the HTTP verb to be used (`GET` in Listing 3-2), the resource URI path (`/PartsServer/resources/parts/200` in Listing 3-2) and the HTTP version (`HTTP/1.1` in Listing 3-2).

General headers - there are a few headers that have general applicability for both request and response messages but not to the entity itself [51]. Examples are **Date** and **Connection**. The **Connection** header states whether or not the underlying TCP connection will be kept open by the server. The **Connection** header should be set to `keep-alive` if the TCP connection is to be persisted or set to `close` if not. The reason for keeping TCP connections alive arose from the performance hit in re-opening connections that were closed by the server – in HTTP1.0, the servers automatically closed the TCP connection after sending back the response to the client. However, if the HTML page had for example, images, these would not have been downloaded initially with the HTML page and thus the browser would have to re-open the TCP connection (which the server had just closed) [53]. Note that in Listing 3-2 the client has requested that the TCP connection be closed

after the server sends its response – this is because that version of RESTlet software (version 1.1.2) did not support persistent connections (fixed in version 2.0).

Request-headers - the request-header fields allow the client to pass additional information about the request, and about the client itself, to the server [51]. These header fields are called metadata, information describing attributes of the representation [124]. Examples include **Host** which identifies the host and port number for the resource requested. Note that the URI resource path from the request line is appended onto the host to give the absolute path to the resource. So for example, in Listing 3-2, the full address is

`http://127.0.0.1:8083/PartsServer/resources/parts/200`. **User-Agent** is another request header. Usually it identifies a browser but in Listing 3-2 the user agent is the RESTlet library. **Accept** is a request header that informs the server what representation(s) the client will accept. In Listing 3-2, XML is the representation requested.

Entity-headers – metadata about the entity body is defined by entity headers. Examples include **Content Type** and **Content Length**, which signify the media type in the entity body and its size in octets (8 bit bytes) respectively. As Listing 3-2 is a GET there is no entity body and thus no entity headers.

Response

When creating a server response, a server creates a message that contains a status line, followed by some headers, a blank line and an optional entity body. Figure 3-6 outlines the overall format of the response and Listing 3-3 details an example response.

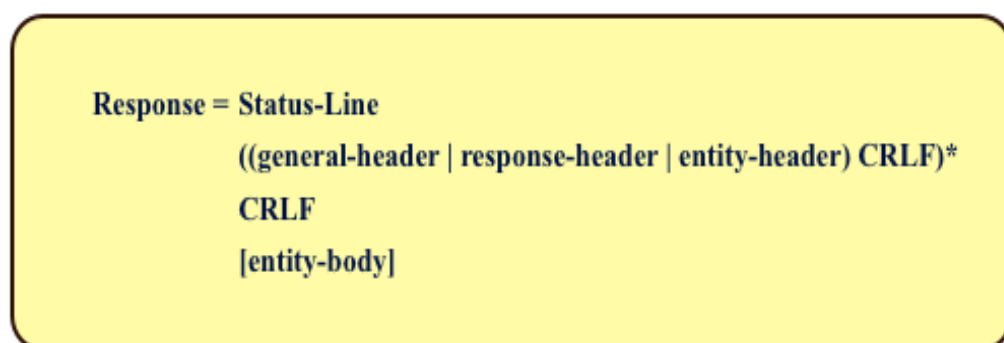


Figure 3-6 HTTP response message format [53]

```

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/xml
Date: Fri, 20 Nov 2009 14:58:51 GMT
Connection: close

<?xml version='1.0'?>
<partNumber>
  <value>200</value>
  <partName>Engine</partName>
  <partDesc>Integral</partDesc>
  <link rel="src" href="/parts/200" />
</partNumber>

```

Listing 3-3 A sample HTTP response message

The status-line outlines the protocol, followed by a numeric status code and its textual phrase [51]. The status codes are grouped into 5 classes or categories as shown in Table 3-2:

HTTP Error Code	Description
1xx	Continue; for example '100 Continue' means that the request has been received and is being processed by the server.
2xx	Success; for example '200 OK' means the request has succeeded; a '201 Created' means a new resource has been created (see the 'Location' header field for the created URI).
3xx	Redirection; for example '301 Moved Permanently' means the requested URI has been moved permanently to a new location (see the 'Location' header field for the URI of the new location).
4xx	Client error; for example, '400 Bad Request' means the server was unable to understand the request due to bad syntax; a '404 Not Found' is returned if the server was unable to locate the resource; a '401 Unauthorized' is used by the server if the client is attempting to access a protected resource without presenting the correct credentials (e.g. username and password).
5xx	Server error; for example '500 Internal Server Error' means the server was unable to process the request due to some unexpected condition.

Table 3-2 HTTP Response codes

The status line in Listing 3-3 states that the protocol is HTTP 1.1 and that this is a response to a successful request. The headers section is similar to the request headers except that in the response, the server can provide certain headers specific to the response:

Response-headers - the response-header fields allow the server to pass additional information about the server and/or response. For example, the **Server** header details information about the server that handled the request and the **Location** header is used to inform the client of the location of a newly

created resource. In Listing 3-3, the response header **Server** indicates that the server was Apache's Coyote [54].

Listing 3-3 contains the entity-header **Content-Type**, indicating to the client (and intermediaries) the media type (`application/xml`) of the message in the entity body. Listing 3-3 also contains the general headers **Date** and **Connection** indicating the date of the response and that the server intends to close the TCP connection.

HTTP methods

HTTP specifies various methods as part of the uniform interface constraint. These methods have well-defined semantics as per the HTTP specification [51]. One is restricted to manipulating resources via these methods only.

GET

To retrieve a resource representation one issues a GET on the resource's URI. The server sends back a representation in the response entity-body. The client can issue a "conditional GET" by including certain request header fields. For example, the client can include the request header **If-Modified-Since** stating the date/time of the latest representation of the requested resource that the client currently has. If the server determines that the resource has not changed since the date/time the client provided i.e. the client has an up-to-date version, the server returns a **304 Not Modified** response with no message body, thereby reducing unnecessary network usage [49].

HEAD

"The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response" [51] i.e. HEAD gives you exactly what GET would give you, but without the entity-body. HEAD is used to retrieve metadata (headers) about the resource without downloading the possibly enormous entity-body. A client can use HEAD to test for the existence of a resource and for finding out information about the resource (e.g. does it support an XML representation) without downloading the entire representation [45].

OPTIONS

The OPTIONS method enables the client to determine what methods are supported by a resource. The server responds with the **Allow** header e.g. **Allow: GET, HEAD** (a read-only resource) [45].

DELETE

To delete a resource representation one issues a DELETE on the resource's URI. If the request was successful, the server can either send back a 200 OK message with further status information in the entity-body or a 204 No Content if there is no entity-body returned i.e. success but no entity-body.

PUT

"The PUT method requests that the enclosed entity be stored under the supplied Request-URI" [51]. If the request-URI does not exist, then the server creates the resource at the given URI with the content (of the entity-body) in the message. If the request-URI does exist, then the server updates the resource identified by the URI with the content in the message. Thus, PUT is used to create/update a resource when the client is in charge of the resource URI to be used [45].

POST

POST can be used in different situations [45]:

- To create new resources underneath a parent resource. For example, a client may send a request to create a customer by passing the customer details to a /customers resource. The server, based on some internal identification system, assigns a customer number to the customer, for example, "2561". The server stores the details in a database (resource state) and sends a 201 Created response to the client with the new URI /customers/2561 in the Location header of the response.
- To append data to an existing resource. For example, assume an event log service that is exposed as URI /log. To add an event, a client POSTs a message to /log. However, in this instance, the server appends the given entity to the end of the log
- POST can be used to turn the resource into a tiny message processor. This is the version of POST that forms and SOAP/POX WS use. This use is described in [51] as *"providing a block of data, such as the result of submitting a form, to a data-handling process"*. Essentially, this is a directive for the server to look elsewhere in the request for the real method e.g. the entity-body. Sometimes however, this *"is the easiest way to express complex operations that span multiple resources"* [45].

Some of the methods have certain useful properties (for free) as shown in Table 3-3:

Features/HTTP Methods	GET	PUT	POST	DELETE	HEAD	OPTIONS
Safety	✓				✓	✓
Idempotence	✓	✓		✓	✓	✓

Table 3-3 Safety/Idempotence of HTTP methods

Safety

A method is considered “safe”, if making the request once is the same as making it n times or not making the request at all. In Mathematics, multiplying by 1 is safe because $4 = 4*1 = 4*1*1*1$ [45]. This property applies to the retrieval methods GET and HEAD. Essentially, side-effects (if any) have not been user driven. This does not mean that GET or HEAD cannot have side-effects e.g. log files on the server are often updated and hit counters incremented based on retrieval requests. “*The important distinction here is that the user did not request the side-effects, so therefore cannot be held accountable for them*” [51].

Idempotence

A method is considered “idempotent”, if making the request once is the same as making it n times i.e. “*the second and subsequent requests leave the resource state in exactly the same state as the first request did*” [45]. In Mathematics, multiplying by 0 is idempotent as $4*0 = 4*0*0*0 = 0$. However, multiplying by 0 is not safe as $4*0$ is not the same as 4. GET, HEAD, PUT and DELETE are all idempotent (POST is not). If you DELETE a resource it is gone; if you DELETE it again it is still gone. If you create a new resource with PUT, it is created; if you resend the same PUT request, the resource is still there and it has the same properties as when you created it. If you use PUT to update the resource state, sending the same PUT request results in resource state which is the same as before. Note that PUT has to be programmed properly to avail of idempotence. For example, “setting x to 5” is idempotent whereas “adding 1 to x ” is not.

Quality of Service (QoS)

HTTP is a sophisticated protocol that supports many advanced features such as caching and Conditional GET. Two of the most important QoS requirements, namely, security and message reliability are now discussed. How HTTP addresses these features is encapsulated by Table 3-4:

Quality of Service		HTTP Implementation		
Feature	Message Reliability	POST Once Exactly (POE)	HTTPLR	Common Best Practice
	Security	HTTP Basic Authentication		HTTP Digest Authentication
		HTTPS		

Table 3-4 Security and Message Reliability in HTTP

Firstly, the Message Reliability approaches are outlined.

Post Once Exactly (POE)

POE is an IETF draft (expired 2005) that renders POST idempotent. *“If a resource supports Post Once Exactly, then it will only respond successfully to POST once, over its entire lifetime”* [45]. This is achieved via the use of the custom HTTP headers POE (on the request) and POE-Links (on the response). On the request, the header POE tells the server that the client is expecting a link to a POE resource and on the response, POE-Links informs the client of the link(s). The client can POST its representation to a link returned in POE-Links. Once a successful POST has been processed by the service, any further POSTs to that resource will result in a **405 Operation Not Supported** response. Thus the resource supports one and only one POST.

Figure 3-7 demonstrates how POE works.

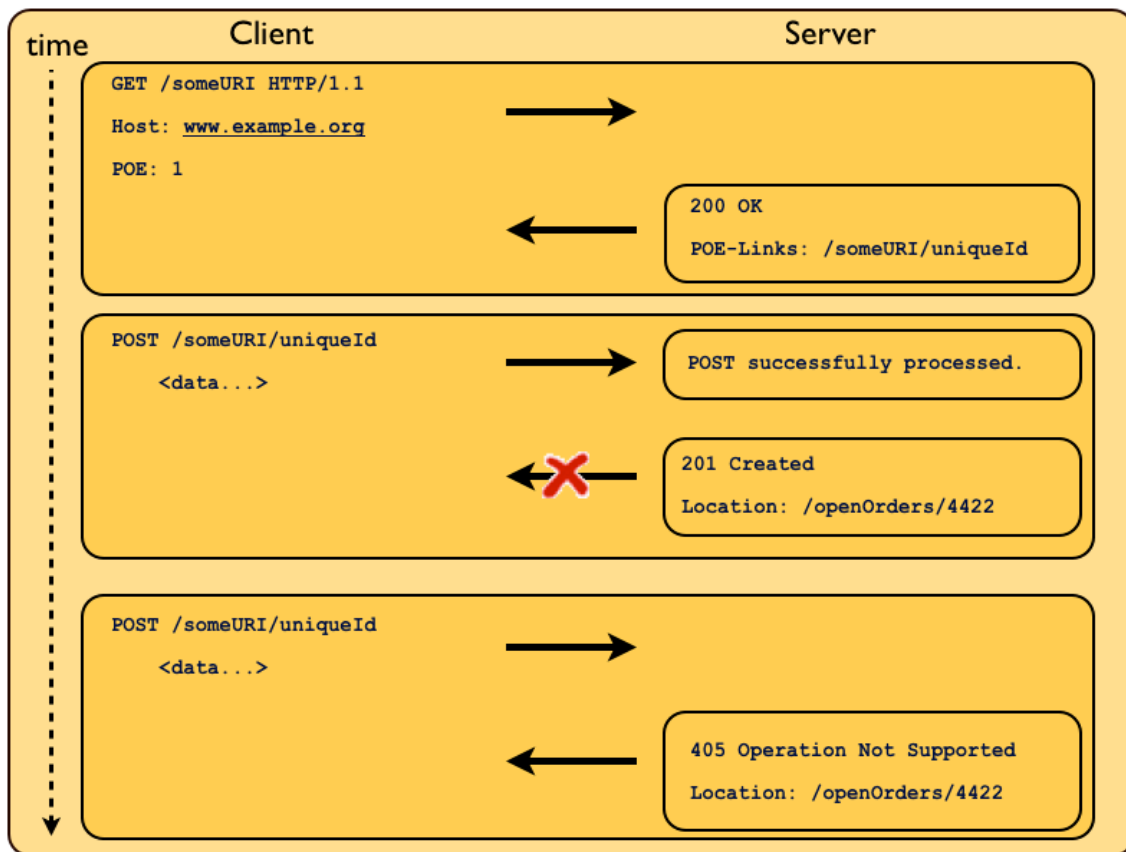


Figure 3-7 POE example

HTTPLR

HTTPLR is a pre-draft memo published by Bill de hOra. The document “*describes an application protocol for guaranteed once and only once transmission of messages using HTTP*” [57]. Its key requirement is that the server and client can agree on the success of the message exchange. This is achieved by representing state transitions in the message exchange using URI’s. The URI assigned by the server for the message exchange is known as an “exchange URI”. The exchange URI, which is unique for each message exchange, is associated on both the client and the server with a URI, represented the current state of the exchange. These well-defined URI’s that represent the state of the message exchange are as follows:

- `http://purl.oclc.org/httpIrr/state/created/` - this notifies the client that the server has created the exchange URI and that the client can now send its message
- `http://purl.oclc.org/httpIrr/state/accepted/` - the server has successfully received the client message at the exchange URI
- `http://purl.oclc.org/httpIrr/state/finished/` - the client has received the server’s response

Both the client and server maintain and link these URI’s with the exchange URI. Consequently, the client and server are aware at all times, of the state of the exchange. Fig 3-8 outlines the process:

1. Establish the Exchange URI
2. Send the Message
3. Reconciliation

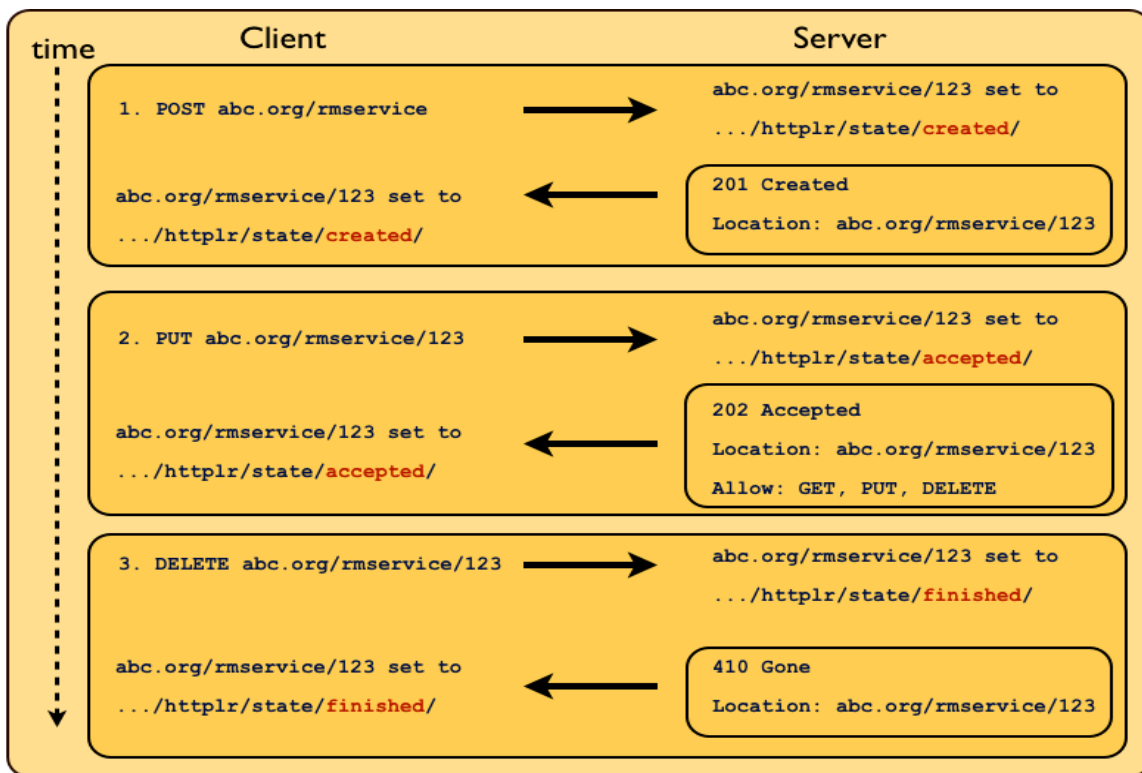


Figure 3-8 HTTPLR example

The process is for the client to request an exchange URI (using POST). The server sets this up and the exchange is in the `http://purl.oclc.org/httpplr/state/created/` state. The client then sends its message to the exchange URI on the server; the server accepts the message, updates the state for the exchange to `http://purl.oclc.org/httpplr/state/accepted/`. The server sends a **202 Accepted** response to the client and the client updates its state to `http://purl.oclc.org/httpplr/state/accepted/` also. The client then sends a reconciliation message to the exchange URI on the server to indicate that it has received the server's response. The server updates its state to `http://purl.oclc.org/httpplr/state/finished/` and responds with a **410 Gone** message. When the client receives the **410 Gone** message it updates its state to `http://purl.oclc.org/httpplr/state/finished/` also. Out of order requests can be handled also; for example, if a client sends a reconciliation message to an exchange URI that is in the **created** state and not the **accepted** state, the server responds with a **405 Not Allowed** message.

Gregorio's Best Practice

The approach in [58] is best practice [55]. Essentially, the approach is premised on the following:

- PUT is idempotent
- The difference between PUT and POST is that: with PUT, the client knows the resource URI that will be used; with POST, the server decides on the resource URI to use (the client POST's to a collection URI).

The pattern outlined by Gregorio is shown in Fig. 3-9. In the pattern, a client initially POST's to a pseudo-live (pending) collection with no data. The server assigns a pending URI and returns it in the **Location** header of a **201 Created** response. The client now knows the (pending) resource URI and can therefore PUT to it, with the actual data. Upon a successful PUT, the pending resource will be made "live" i.e. the server creates a live URI for the representation in the PUT request and then sends a **303 See Other** response with the live URI in the **Location** header. If the PUT request or the response to the PUT request gets lost, the client can re-submit the request, as PUT is idempotent; if the POST gets lost, we can also re-submit the request, as there was no data within the POST message (all we lose are some logical URI's) [58].

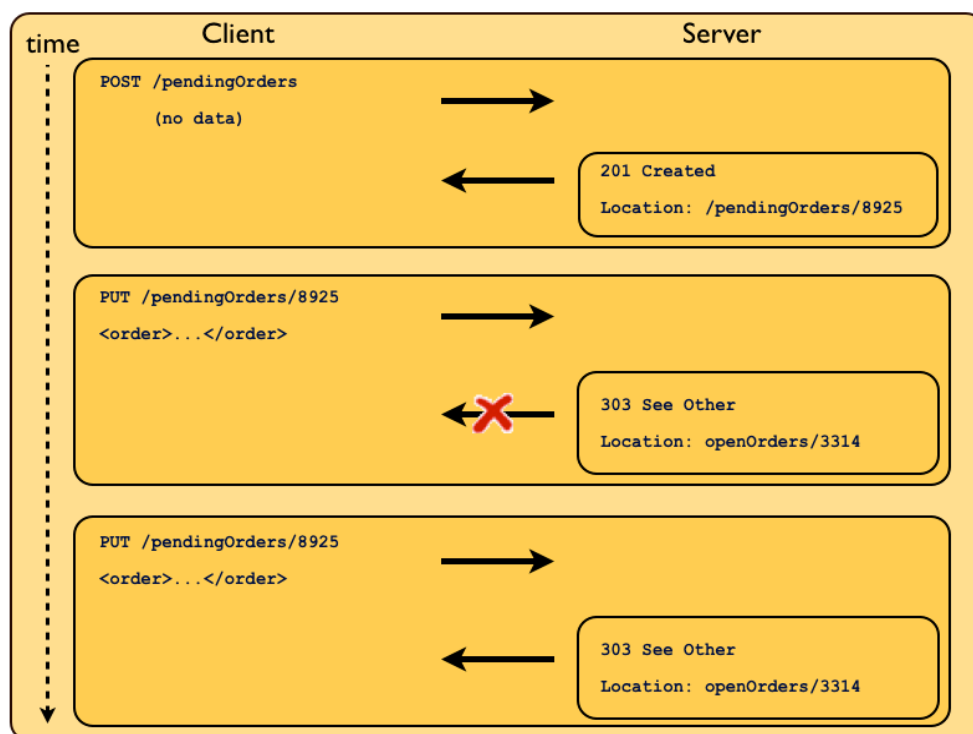


Figure 3-9 Message Reliability [based on 58]

Message Reliability

The essential requirements of message reliability are:

- Notification of success/failure
- Once and only once processing
- In-order message processing

Notification of success/failure

HTTP has a rich set of response codes that identify both successful (2xx) and unsuccessful responses (e.g. 4xx and 5xx). From a client's perspective, receiving a 200 OK response indicates that your request has been received and processed successfully.

Once and only once processing

Questions arise when no response is received: was it the request or the response that was lost; if it was the response that was lost, was the request processed? The simplest solution would be for the client to keep re-sending the request until it gets a successful response. This is only possible if the server can “handle” duplicates. HTTP achieves this with the *idempotence* property, whereby executing the request more than once is the same as executing it once. GET, PUT and DELETE are all idempotent and thus, providing the server is implemented properly, if the client does not receive a response for a GET, PUT or DELETE request, the client can simply re-issue the request [55]. POST, however, is not idempotent. There are various options available to address this: Mark Nottingham's Post Once Exactly (POE), Bill de hOra's HTTPLR and Joe Gregorio's common best practice approach.

In-order message processing

HTTP's native response codes and Gregorio's best practice provide support for notification of success/failure and once and only once processing respectively. Thus, the first two requirements for message reliability are catered for. The last requirement of in-order messaging can be provided via appropriate use of headers and response codes [59]. This approach involves the client maintaining a token representing the client's view of the current state of the resource. Each time the client sends up a request it inserts this token; the server checks the client's token (or view) with its own view of the resource state. If these tokens do not match then a message has been duplicated or lost. The server could respond with a **409 Conflict** indicating that there is “*a conflict with the current state of the resource*” [51]. Fig. 3-10 gives an example scenario. Note that in the figure, the token being maintained is an XML attribute named `expectedState`. The **409 Conflict** message should include information or a link that enables the client to refresh its view of state so that it can re-sync with the server.

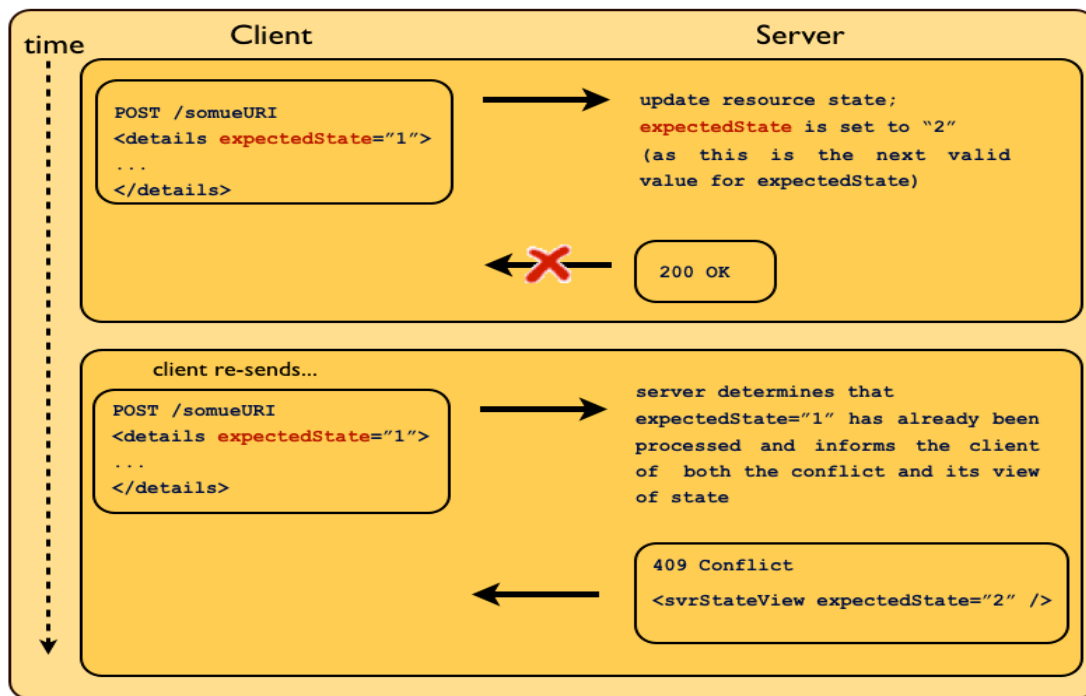


Figure 3-10 In-order message delivery

At this point, the second QoS requirement, namely security is outlined.

Security

HTTP security is referred to as HTTPS (HyperText Transfer Protocol Secure). HTTPS is not a separate protocol in itself, but is in fact an HTTP message (at the application layer) being transmitted over an encrypted channel setup (at the lower transport layer) by Transport Layer Security (TLS) or its predecessor SSL. HTTPS is a point-to-point protocol where the entire message is encrypted i.e. made secure [18].

The Internet has the following security requirements:

- authentication (ensures that access is only given to those who can prove their identity)
- authorisation (determines the resources a given identity has access to)
- confidentiality (ensures that sensitive data such as credit card numbers remain private)
- integrity (ensures that a message is not modified in transit)
- nonrepudiation (means that a sender cannot later deny having sent a message)

HTTP provides authentication and authorisation via the use of Basic authentication and Digest authentication. Both of these authentication schemes use HTTP headers to transmit sensitive data. Both approaches use the following HTTP headers:

- **WWW-Authenticate**
 - server to client header; the server uses this header to inform the client what it must provide in order to authenticate
- **Authorization**
 - client to server header; the client uses this header to pass information regarding its credentials to the server

Basic Authentication. [45]

If the client tries to access a resource that is protected by Basic authentication and the client does not provide the proper credentials, the client receives a challenge and has to make the request again. For example, if the client makes a request to a protected resource as in Listing 3-4:

```
GET /apples.html HTTP/1.1
Host: www.example.com
```

Listing 3-4 GET on a protected resource

This request does not include the correct credentials; in fact it has no credentials at all. The server will response with a challenge similar to the following (Listing 3-5):

```
401 Unauthorized
WWW-Authenticate: Basic realm="Private fruit"
```

Listing 3-5 401 Unauthorized access message

The server informs the client (via the 401 Unauthorized challenge) that the credentials provided were incorrect. The server uses the WWW-Authenticate header to inform the client of the type of authentication it is using (Basic) and the realm. The realm is used to identify a collection of resources protected in a certain way. Certain realms may use Basic authentication whereas other realms may use Digest authentication. Users may be authorised to access some realms but not others. The realm states: “*what is being protected*” and the authentication type states: “*how the realm is protected*”. To meet a Basic authentication challenge, the client provides a username/password pair, combined as a single base 64 encoded string.

The resulting string is the value of the Authorization header on the client's next request:

```
GET /apples.html HTTP/1.1
Host: www.example.com
Authorization: Basic QWxpYmFiYTpvcGVuIHNlc2FtZQ==
```

Listing 3-6 Authorization header

The server will decode the supplied string and check it against its username and password list. If they match, authorisation is given and the request is processed; if not the **401 Unauthorized** is again returned to the client. However, if the server can decrypt the password then so can network snoops. To prevent this, instead of using plain HTTP, HTTPS is used (as HTTPS encrypts all the communication between the client and server, including the `Authorization` header).

Digest Authentication

Digest authentication is another way to hide the authorization credentials from network snoops. It is more complex than Basic authentication but is secure, even over plain HTTP. It is however, less well supported [50]. Digest authentication follows the pattern of Basic authentication, in that the client issues a request and gets a challenge. Assuming the same resource as above, here is a sample challenge:

```
401 Unauthorized
WWW-Authenticate: Digest realm="Private fruit",
    qop="auth",
    nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
    opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

Listing 3-7 Digest authentication challenge

The `qop` parameter returned from the server indicates the “quality of protection”. If it is set to “auth” (as above) this means that the client calculates the second MD5 (Message-Digest algorithm 5) hash (known as HA2) based on the HTTP method and resource path. If `qop` is set to “auth-int” then the entity body of the request is included in the HA2 calculation also. The `nonce` (number used once) is a randomly generated single-use value that is different for each **401 Unauthorized** challenge. It acts as a session token i.e. any attempt to use those credentials other than in the current session will be rejected e.g. a replay-attack where the message is intercepted and the same credentials are re-used.

The **opaque** parameter is a lookup that provides context for the server. The client is expected to send it back unchanged. The client uses the information provided by the server to create an encrypted string. This encrypted string needs to prove that the client knows the password without actually providing the password [45]. This is called a “digest” and the sequence of steps involved for a client in calculating a digest is as follows (where HA1, HA2 and Digest are all MD5 hash functions):

- HA1 = MD5 (*username* : *realm* : *password*) where *realm* is retrieved from the server challenge
- HA2 = MD5 (*HTTP method* : *resource path*) where *qop*=”auth”
- Digest = MD5 (*HA1* : *server nonce* : *client sequence_number* : *client nonce* : *qop* : *HA2*)

The client uses the *realm* (“Private fruit”), *server nonce* (“dcd98b7102dd2f0e8b11d0f600bfb0c093”) and *qop* (“auth”) from the server challenge; and its own *HTTP method* (“GET”), *resource path* (“/apples.html”), *username* (“skennedy”), *password* (“jellyfish”), *sequence number* (“00000001”) and *client nonce* (“0a4f113b”) to calculate the digest. The sequence number is expected to increase by one for each request for that server nonce (prevents the same credentials being used in replay-attacks). So, for example, the client would calculate the Digest as follows (backslash means a continuation):

- HA1 = MD5 (“skennedy” : “Private fruit” : “jellyfish”)

= 939e7578ed9e3c518a452acee763bce9
- HA2 = MD5 (“GET” : “/apples.html”)

= 39aff3a2bab6126f332b942af96d3366
- Digest = MD5 (“939e7578ed9e3c518a452acee763bce9: \

dcd98b7102dd2f0e8b11d0f600bfb0c093: \

00000001 : 0a4f113b : auth: \

39aff3a2bab6126f332b942af96d3366”)

= 6629fae49393a05397450978507c4ef1

Now that the client has calculated the Digest, the client can re-submit the request with the correct credentials (as in Listing 3-8):


```
GET /apples.html HTTP/1.1
Host: www.example.com
Authorization: Digest
    username="skennedy",
    realm="Private fruit",
    nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
    uri="/apples.html",
    qop=auth,
    nc=00000001,
    cnonce="0a4f113b",
    response="6629fae49393a05397450978507c4ef1",
    opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

Listing 3-8 **Sample client request using Digest Authentication**

The server, based on the same information (except the password), re-calculates the Digest and if they match, the message is processed. This means that in Digest authentication, the server need not keep passwords. This is possible because, when the password is being set for a user on that realm, the server stores the HA1 value (not the password). When a Digest authentication request is received, the server retrieves the HA1 value based on the *username* and *realm* passed in the request. The Digest can then be calculated based on the HA1 retrieved and the parameters in the message.

Both Basic and Digest authentication follow a “request, challenge, response” model. However, one difference is that Basic authentication will not receive a challenge if the initial request provides the correct credentials (username and password). Part of the Digest authentication model is for the server to challenge the client on every initial request [45]. As pointed out earlier, another difference is that in Digest authentication, the server does not need to know or store the client’s passwords.

As stated earlier, TLS encrypts the complete communication thereby providing confidentiality. Both symmetric and asymmetric encryption is used. The initial setup (known as a TLS handshake), involves the client initially requesting a secure connection from the server. The server returns its certificate, including the server’s public key. The client encrypts a random number with the server’s public key and sends this value to the server. The server uses its private key to decrypt the random number sent by the client. This random number is used to generate shared symmetric keys (called session keys) for the remainder of the secure connection.

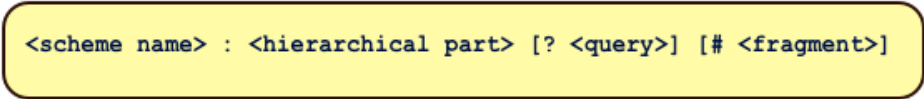
Authentication is provided at two levels:

1. both Basic and Digest authentication ensure the client provides the correct credentials. Once authentication is complete, authorisation is normally implemented via access control lists.
2. digital certificates provide proof of identity i.e. they ensure that the individual/organisation is who they say they are. This is achieved by the server sending a digital certificate, which is digitally signed by a trusted third party, the CA. The CA binds the public key on the certificate to the identity on the certificate. The client can, if desired, authenticate that the CA issued the certificate.

Digital signatures provide integrity and non-repudiation (as opposed to confidentiality). Once a message is signed, it cannot be modified without invalidating the signature. In addition, an organisation cannot subsequently claim that they did not send a signed message (while at the same time claiming that their private key is secret).

3.3.2 Uniform Resource Identifier (URI)

REST specifies that resources be given unique names. On the Web, resources are named with URI's. A URI is a string of characters used to identify a resource on the Internet. Fielding describes URI's as *"both the simplest element of the Web architecture and the most important"* [3]. URI's enable us *"to describe the location of something, anywhere in the world, from anywhere in the world"* [60]. The syntax of a URI is outlined in Fig. 3-11:



```
<scheme name> : <hierarchical part> [? <query>] [# <fragment>]
```

Figure 3-11 **URI syntax structure**

The URI structure is best explained with a simple example. In this example, we are looking for specific details of an employee's educational history. We will assume HTML format and a browser client. In Listing 3-9, the details of employee 15481 are requested but the details in the representation are narrowed to just his/her educational history by passing in a query parameter (`p=education`). By specifying the fragment `#thirdLevel`, the browser will jump to that section of the employees educational history (bypassing their second level education for example).

```
http://example.org/employees/15481/details?p=education#thirdLevel
```

Listing 3-9 Sample URI

In Listing 3-9:

- `http` is the scheme name
- `//example.org/employees/15481/details` is the hierarchical part which can be broken down in to the naming authority (`//example.org`) and the path (`/employees/15481/details`)
- `p=education` is the query
- `#thirdLevel` is the fragment (the section of the HTML representation that the browser will jump to)

On the Web, URI's identify resources and resource state is transferred using resource representations. URI's are opaque. This means that one should not infer the nature of a resource by inspecting the URI that identifies it [61]. For example, a resource is identified as *http://example.org/page.html* does not have to serve HTML as its representation i.e. HTTP does not infer the media type from the URI [62]. Note that servers can, if they wish, use this mechanism or they can use the **Accept** header from the request [45]. The Web architecture “*promotes independence between an identifier and the state of the identified resource.*” [62]. This is the information-hiding principle governing the uniform interface constraint and it means that changes in resource state do not require changes to the URI identifying that resource state [62].

URI opacity versus readable URI's

REST states that resources should have names but nothing about whether the names should be readable or opaque. A readable URI is a meaningful URI. In order to be meaningful, a URI must refer to the resource and therefore it contains elements of resource state. A problem arises if the resource state being used as part of the URI changes e.g. someone gets married and changes their name. Any links to the old resource URI from other web pages or bookmarks are broken.

An opaque URI is a URI where “*you should not look at the contents of the URI string to gain other information*” [61]. Even though the URI may look meaningful, one cannot make any assumptions i.e. one cannot assume that by varying the URI that one is varying the resource. Opaque URI's are also known as “cool” URI's [63] and are designed for longevity, not human readability. Cool URI's can

expose resource state also but because the intention is that the URI will never change, the resource state used is usually meaningless e.g. a database table ID [45].

Whether to use opaque URI's, where there is no visible relationship to the resource it names, or meaningful (readable) URI's, where the URI breaks when its resource state changes, is very subjective. Both [45] and [4] prefer meaningful URI's. Richardson in [45] prefers to use resource state that is least likely to change and thus least likely to result in broken URI's. Ruby in [45] would use resource state that may change and use a permanent redirect at old/broken URI's.

Note: URI's can act as URN's (Uniform Resource Name) or URL's. A Uniform Resource Name (URN) is like a person's name (how you identify someone) whereas a Uniform Resource Locator (URL) resembles that persons address (where and how to find them). The example of a book helps: one can discuss an exact edition of a certain book by referring to its International Standard Book Number (ISBN) e.g. urn:isbn:0-486-27557-4 specifies the unique identifier within the ISBN identifier system that identifies a specific edition of Shakespeare's play "Romeo and Juliet". One can discuss the play based on the URN of the play. However, to read the book one needs its URL (it's location and how to access it) e.g. <http://example.org/RomeoAndJuliet.pdf>. One can type this link into a browser and the browser will (try to) find the resource "Shakespeare's play RomeoAndJuliet" based on the protocol (**http**) and the network host (**example.org**) specified in the URL. The PDF (Portable Document Format) representation of the resource is returned and rendered by the browser [64].

3.3.3 Representations and Media Types

The term REST comes from Representational State Transfer and this term comes from the fact that "*RESTful clients and resources transfer resource state representations to each other*" [49]. The data format of the representations are known as media types [46]. Media types are registered with IANA and are available globally. RESTful Web Service authors need to understand their client types so as to serve meaningful representations. For example, a service that has browser and application based clients should return (X)HTML for the browser clients and XML or JSON (JavaScript Object Notation) for the programmatic clients. The client and server must agree on the resource representation. This can be done out-of-band. For example, the client developer looks up the RESTful Web Service description beforehand (typically published on a Web page) and develops the client application accordingly. Alternatively, the client application can use HTTP's *Content Negotiation* whereby a client sends a request with the **Accept** header containing the list of media types that it understands (in order of preference). Assuming the server supports one of the media types the client understands, the server responds with the **Content-Type** header set to the representation data format (media type) it is returning.

Services must be aware of the HATEOAS constraint when returning representations. To quote Vinoski: *“To support the hypermedia constraint, resource representations should contain hyperlinks to related resources”* [49].

3.3.4 Web Caching

Transforming opaque messages into visible messages is an important contribution of the research presented in this thesis because message visibility enables the Web infrastructure. One of the most important features native to the Web is caching. Caching is a critical feature in enabling the scalability and efficiency of the Web. The goal of caching in HTTP/1.1 is to: *“eliminate the need to send requests in many cases, and to eliminate the need to send full responses in many other cases. The former reduces the number of network round-trips required for many operations; we use an “expiration” mechanism for this purpose. The latter reduces network bandwidth requirements; we use a “validation” mechanism for this purpose”* [51]. Note that, in caching, it is the servers that control whether their responses can be cached and if so, for how long [49].

Expiration

To mark a response as cacheable, the origin server (i.e. the server from where the representation was *originally served*) inserts certain HTTP headers into the response message. The `Expires` header is one such header. This header enables servers to insert a specific time indicating how long a particular response can be considered “fresh” or “stale”. Subsequent equivalent requests to the cache can be served with fresh copies. If however, the content is stale then the request is forwarded onto the origin server. Thus, Web servers can specify static content such as navigation bars on Web sites to expire well into the future; in addition, news stories can be can expired at a specific time of day e.g. 6am, forcing caches to retrieve new representations. One of the issues with an absolute time approach is that because it is time-related, both the Web server and cache clocks must be synchronised. If the Web server and cache have different ideas of the time, then stale content could be considered fresh.

The `Cache-Control` header addresses the `Expires` header issue and enables Web server’s greater control over caches as it enables the specification of directives that *“MUST be obeyed by all caching mechanisms along the request/response chain”* [51]. For example, `“max-age=3600”` specifies that this response is fresh for 3600 seconds (one hour) from the time of the request. This time setting is relative as opposed to the absolute `Expires` header time.

Validation

If a cache has a stale representation that it would like to serve, the cache needs to “validate” if the cached entry is still usable. HTTP/1.1 supports “Conditional GET” in order to enable the cache to check:

- If the cached entry is valid: the server should inform the cache of this fact without retransmitting the full response, as this is an unnecessary overhead [51]:
- If the cached entry is invalid: the server should serve the up-to-date representation to the cache without the need for an extra round trip, as this also is unnecessary overhead

Conditional GET

Conditional GET can be implemented in two different ways. In the first method, when a client issues a GET for a resource, the server returns the date and time of the most recent change to that resource in the `Last-modified` header. The response can of course include caching headers/directives as outlined above, enabling caches to store a copy of the representation. Now, when a cache needs to check if its local copy is still fresh, it can use the date/time from the `Last-modified` header in an `If-modified-since` request header. The server uses this header to see if the resource has changed; if not, the server returns a status code `304 Not Modified` with an empty message body, thereby informing the cache that it can use the representation it has. If the server determines that the resource has changed then the new representation is served to the cache [49].

This date/time approach to Conditional GET leaves open a one second where changes cannot be detected. The entity tag mechanism is the second method for implementing Conditional GET and it addresses this one second window issue. A server returns a hash value as a string in the `Etag` header. Caches can send this hash value back to the server in the `If-none-match` header. If the server rehashes the resource and the values match then the cache has an up-to-date representation of the resource and the server returns a status code `304 Not Modified` with an empty message body (as above). If the hash values do not match then the server returns the new representation to the cache. Note that for the entity tag mechanism to be of benefit, the cost of calculating the hash value must be less than the cost of acquiring and returning the whole representation [49].

In summary, caching improves the efficiency of the Web by lowering the latency of requests and reducing network bandwidth. Latency is reduced due to the fact that, instead of a request having to travel all the way to the origin server, an intermediate cache, which contains an up-to-date copy of the representation, can satisfy that request. Network bandwidth is reduced because conditional

requests can be sent to the server and if the cache has an up-to-date representation, the server will inform the cache of this fact without sending the (possibly large) entity body.

3.4 Web Service architectures

For the purposes of this thesis, it is important to be able to outline the criteria by which a service is considered RESTful. The criteria are outlined in [45] and relate to the location in the request of the *method* and *scoping information*. The method informs the server what operation the client wants to perform, for example: to retrieve or delete data. The scoping information informs the server what data the client wants to operate on, for example: customer data or order data. Table 3-5 shows the matrix of how the locations of both the method and scoping information affect whether a service is considered to be RESTful, RPC or REST-RPC hybrid.

Web Service Architectures		Scoping Information	
		URI	Entity Body
Method Information	URI	REST-RPC (GET only)	N/A
	Entity Body	N/A	RPC (POST only)
	HTTP Method	RESTful	N/A

Table 3-5 Criteria for evaluating Web Service Architectures

The three Web Service architectures defined by [45] are as follows:

- RESTful - In a RESTful architecture, the method information is in the HTTP method and the scoping information goes into the URI. Listing 3-10 is an example RESTful request:

```
GET http://localhost:8081/PhoneDirServer/resources/phoneDirectory/Kennedy/Sean HTTP/1.1
Host: 127.0.0.1:8081
User-Agent: Noelios-Restlet-Engine/1.1..2
Accept: */*
Connection: close
```

Listing 3-10 RESTful request

- **RPC**
 - RPC-style services encapsulate the method and scoping information in an envelope. The kinds of envelope are unimportant but HTTP and SOAP are popular examples. Every RPC-style service defines its own brand new vocabulary [45]. Only one URI is exposed (the “endpoint”) and POST is the only verb supported. The messages are parsed to elicit the Web Service to execute and its associated parameters. Examples of RPC-based Web Services are SOAP and POX Web Services. Note that, to improve efficiency, some SOAP services also insert the Web Service to be executed into a specific “*SOAPAction*” header (thereby saving on the overhead of parsing the entity body). Listing 3-11 is an example RPC-style request (SOAP in this instance):

```
POST /PhoneDirServer/PhoneDirectory HTTP/1.1
SOAPAction: ""
Content-Type: text/xml;charset="utf-8"
Accept: text/xml
User-Agent: JAX-WS RI 2.1.4-b01-
Host: 127.0.0.1:8081
Connection: keep-alive
Content-Length: 248

<?xml version="1.0" ?>
  <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
      <ns2:getPhoneNumber xmlns:ns2="http://PhoneDirServerPkg/">
        <firstName>Sean</firstName>
        <surname>Kennedy</surname>
      </ns2:getPhoneNumber>
    </S:Body>
  </S:Envelope>
```

Listing 3-11 RPC-style request

- **RPC-REST**
 - RPC-REST services encapsulate both the method and scoping information in the URI. The RPC classification is due to the fact that these services define their own vocabulary and use HTTP as an envelope format, inserting the method and scoping information

wherever they please. The method GET is used for *all* messages. For example, Listing 3-12 is the HTTP message used when searching for “kingfisher” on the Flickr online photo sharing service:

```
GET /services/rest?api_key=XXX&method=flickr.photos.search&tags=kingfisher HTTP/1.1
Host: www.flickr.com
```

Listing 3-12 RPC_REST request [45]

Note that in Listing 3-12, both the method and scoping information are passed in the URI i.e. the method is *flickr.photos.search* and the scoping information is *kingfisher*. As GET is for retrieving, this message is in fact RESTful. Many read-only Web Services (which use GET) are in fact RESTful even though the method information is passed in the URI. The issue arises when a client wants to modify the data set. For example, deleting a photo on Flickr involves making a GET request to a URI that includes *flickr.photos.delete*. In this scenario, the method semantics of GET conflict with the operation carried out. The Flickr Web API is in fact a hybrid: RESTful when retrieving data and RPC-style when modifying the data set.

3.5 REST anti-patterns

Based on the architectures identified in the previous section, REST anti-patterns identified in [26] merit further discussion. Of particular relevance to this thesis is “POST Tunnelling”. It is worth noting, that the motivation for the development of the REST architectural style was predicated on the following: “*how do we introduce a new set of functionality to an architecture that is already widely deployed, and how do we ensure that its introduction does not adversely impact, or even destroy, the architectural properties that have enabled the Web to succeed?*” [112].

3.5.1 Tunnelling everything through GET

This anti-pattern corresponds to the REST-RPC hybrid architecture above. Rather than a URI identifying a resource, the URI becomes a convenient means for encoding parameters. This leads to URIs like the following:

```
http://example.org/some-api?method=deleteCustomer&id=8778
```

Listing 3-13 GET anti-pattern example

In this scenario, it is unlikely that the GET will be “safe” and that the client will be held responsible for the customer deletion. “*The spec says that GET is the wrong method to use for such cases*” [26]. The attraction for developers is that it is very easy to test from a browser i.e. paste the URI into the address bar, edit some parameters and press Enter.

As with the REST-RPC architecture, certain URIs can be accidentally RESTful. Take for example, the following URI:

```
http://example.org/some-api?method=retrieveCustomer&id=5477
```

Listing 3-14 Accidentally RESTful GET anti-pattern example

It can be argued that this URI identifies a resource (as it can be argued that this URI identifies an operation and its parameters). Doing a GET on the URI may well be “safe”. However, this is often unintentional. Often, APIs start this way, exposing a “read” interface, but when developers start adding “write” functionality, “safety” breaks. In many cases, if the developer wanted to update a customer, rather than using PUT, the developer would use GET and encode the operation and parameters in the URI [26].

3.5.2 Tunnelling everything through POST

This anti-pattern [26] corresponds to the RPC architecture above. In this scenario, HTTP POST is used for all operations. The entity body contains the real operation to be executed along with its parameters. This message is POSTed to a gateway URI where the message is parsed and the relevant Web service is executed. This is how SOAP and POX systems operate. Listing 3-11 is a sample SOAP message. Rather than actually violating REST principles, this anti-pattern simply ignores them. One of the implications of this anti-pattern is that native Web caching is disabled. This is due to the fact that the Uniform Interface is ignored even when retrieving data. Instead of using GET where appropriate, the real intention of the message is located inside the XML. This is demonstrated in Fig 3-12:

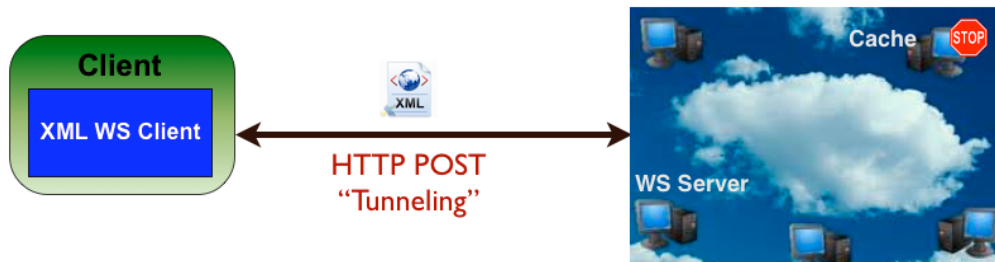


Figure 3-12 HTTP POST Tunnelling

Essentially, for a read/write Web Service to be considered RESTful, it should use GET for retrieval and PUT/POST/DELETE for modification of the data set. Other REST anti-patterns exist, but they are not as relevant to this thesis e.g. services which ignore response codes [26].

3.6 Summary

In this section the REST architectural style and its constituent constraints were presented. The reasoning behind the constraints i.e. their benefits to a distributed hypermedia system were highlighted. As REST is, in itself abstract, the most popular implementation of REST to date, the Web, was outlined. The Web's constituent elements, namely: HTTP, URI's and Media Types were covered from the viewpoint of "RESTful HTTP" i.e. *"using HTTP as intended and as described by the REST principles"* [4]. Web Service architectures and REST anti-patterns were also discussed.

The next chapter will focus on the Semantic Web, a suite of technologies that enables machines to reason about formatted information.

Chapter 4

Semantic Web

4.1 Introduction

In the previous chapter, the REST architectural style and its most ubiquitous and popular implementation, namely HTTP were outlined. At this point, the major Web Service implementation paradigms have now been detailed i.e. XML Web services (SOAP/POX) and RESTful Web services. The goal of this thesis is to map the XML Web service to its RESTful HTTP counterpart. This can of course be achieved in a manual fashion with human intervention. However, a state of the art implementation would avail of Semantic Web technologies to automate this mapping. This chapter introduces the core technologies underpinning the Semantic Web before going on to explain the technologies required by the architecture central to this thesis.

4.1.1 “The Web” versus “The Semantic Web”

The classic Web focuses on publishing and linking documents and is designed for humans i.e. web pages are designed to be read and understood by humans. The focus of the Semantic Web is on publishing and linking both documents and data [105] and is designed for computers i.e. the information content is structured so that machines can understand it. The vision of the Semantic Web is of “*building a global Web of machine-readable data*” [104]. The rationale behind this vision is the fact that the HTML files that constitute the Web are a major repository of hidden data [108]. “*The*

ultimate goal of the Semantic Web is to transform the Web into a medium through which data can be shared, understood and processed by automated tools” [151]. The Semantic Web is a set of technologies, which provide “semantics” or meaning to the current Web content. Thus, the Semantic Web is not a replacement for the existing Web but “is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.” [66].

4.1.2 Background

The Semantic Web is the vision of Tim Berners-Lee, first outlined in [66]. This vision of a Web of data having well-defined meaning has received a lot of attention in academia in recent years. In [115], Hepp states that: *“the Semantic Web is, without doubt, gaining momentum in both industry and academia”*. The Semantic Web was identified as having the potential to address issues facing Web Services researchers in areas such as interoperation and especially in the areas of automation of discovery and composability. This area of research is known as Semantic Web Services (SWS) and is an active area of research:

- In [155], the authors state *“Semantic Web services (SWS) are a research effort towards automation of the use of Web services, enhancing existing SOA capabilities with intelligent and automated integration”*. This is exactly how the architecture outlined in this thesis utilises the Semantic Web.
- In [156], the authors propose a semantic-based Web service registry that filters client requests in order to improve Web Service discovery.
- In [83], how Semantic Web technologies can assist in the composability of RESTful WS is addressed. A RESTful WS composition is known as a “mashup” and a semantically informed mashup is called as “SMashup” (semantic mashup). The authors semantically annotate RESTful WS descriptions using a poshformat called SA-REST (Semantic Annotation of Web Resources: discussed in section 4.5.2.6) thereby giving their *“SMashups the ability to know more about what the service is doing and what the inputs and outputs are so that data mediation can be done automatically without human intervention”* [83].
- In [154], the authors outline Semantic Bridge for Web Services (SBWS); a framework for integrating both SOAP and RESTful WS into the Semantic Web. The SOAP WS described by WSDL and the RESTful WS described by WADL are semantically annotated, thereby enabling SBWS to execute queries against the semantically-enhanced WS descriptions via a SPARQL (SPARQL Protocol and RDF Query Language) endpoint. Thus, SBWS adapts existing SOAP and RESTful WS for semantic query. The authors also present Semantic REST, a framework that maps between the HTTP verb set and SPARQL commands e.g.

GET maps to SELECT. The result is the ability to interact with RDF (Resource Description Framework) data via RESTful endpoints i.e. “*semantic resources double as web-resolvable pages*” [154].

Semantic Web Services are discussed further in section 4.5.

4.2 Resource Description Framework (RDF)

On the Semantic Web, information is represented as a set of assertions called *statements*. Statements are made of three parts (or *triples*): subject, predicate and object (in that order). The subject is the thing the statement describes and the predicate is the relationship between the subject and the object [67]. The Semantic Web represents these statements using RDF.

RDF [68], a W3C specification, is the Semantic Web’s data model for representing statements. RDF is modelled abstractly as a graph. RDF identifiers are URI references and RDF’s official standard syntax is RDF/XML. The RDF model encodes data in the form of subject, predicate, object triples. The subject and object of a triple are both URIs that identify a resource (or a URI and string literal respectively). The predicate is also represented by a URI and describes the relationship between the subject and object. For example, an RDF triple can relate a person A to a scientific article B by stating that A is the author of B [104]. These topics are now discussed in more detail.

4.2.1 RDF graphs

“*RDF is based on the idea that the things being described have properties which have values*” [69]. RDF statements have the triple form previously described i.e. subject, predicate and object. These statements are modelled abstractly as a labelled directed graph. For example, in Listing 4-1 we have some English statements, which are graphed in Fig 4-1. In Fig. 4-1, the nodes, represented by ovals, are the subjects and objects of the statements. The relationships between the nodes are the predicates (or properties) and are represented by directed arrows. For example, in Fig. 4-1, *Sean* is a subject node, *Rob* is an object node and their relationship is represented by the predicate “*knows*”. Literals, represented by rectangles e.g. “*Kennedy*”, are strings, dates, numbers etc.. and cannot be used as subjects or predicates.

```

Sean knows Rob.
Seans surname is Kennedy.
Rob knows Paul.
Declan works with Paul.

```

Listing 4-1 Information about the authors

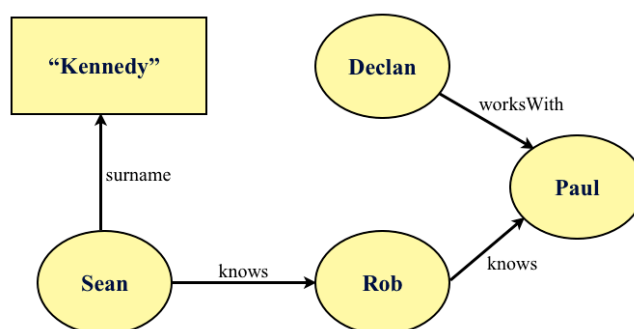


Figure 4-1 RDF graph of Listing 4-1

Natural language, such as English, is ambiguous e.g. *Sean* in Listing 4-1 could be anybody whose first name is *Sean*. To make these statements machine-processable, there are two requirements [69]:

- identifiers used for subjects, predicates and objects must be unique across the Web
- a machine-processable language for representing these statements and exchanging them between machines

The first is covered by RDF Identifiers (section 4.2.2) and the second is covered by RDF Serialisation (Section 4.2.3).

4.2.2 RDF Identifiers [69]

A Uniform Resource Identifier (URI) is the more general term for Uniform Resource Locators (URLs) and Uniform Resource Names (URNs). A URN is similar to a person’s name whereas a URL resembles a person’s address i.e. an URN identifies an item whereas a URL provides a method for finding it. URI references (URIs) are URI’s with an optional fragment appended (separated by a #) . For example, assuming a base URI for an ontology of

“*http://www.leitrimmills.ie/ontologies/animals*”, one could have a URIRef such as “*http://www.leitrimmills.ie/ontologies/animals#goats*”. It is the URIRefs that RDF uses for identifying resources. RDF defines a resource as anything that is identifiable by a URI reference and thus RDF can describe anything and any relationship. RDF describes resources in terms of simple properties and property values. Fig. 4-2 represents an RDF graph of the following sentences:

- *<http://www.leitrimmills.ie/index.html> has a *creator* whose value is Sean Kennedy*
- *<http://www.leitrimmills.ie/index.html> has a *creation-date* whose value is June 11, 2006*
- *<http://www.leitrimmills.ie/index.html> has a *language* whose value is English*

In Fig. 4-2, <http://www.leitrimmills.ie/index.html> represents the subject in all the statements and thus all arcs leave from it. The *creator*, *creation-date* and *language* predicates/properties have been replaced with URIs. The value of the *creator* property is a URIref representing “Sean Kennedy” (as staff id 458). The values of the *creation-date* and *language* properties are literals (strings, numbers, dates). Note that the *creator* property is very specific i.e. rather than using a literal to identify one of the thousands of “Sean Kennedy” resources, we use a URIref. This creates a very specific association to the “Sean Kennedy” at leitrimmills.ie whose staff id is 458. Another benefit in using a resource for “Sean Kennedy” and not a literal, is that, as “Sean Kennedy” is now an object resource, we can assign properties to him. We could not do this if we have directly used the literal “Sean Kennedy” as the value for the *creator* property.

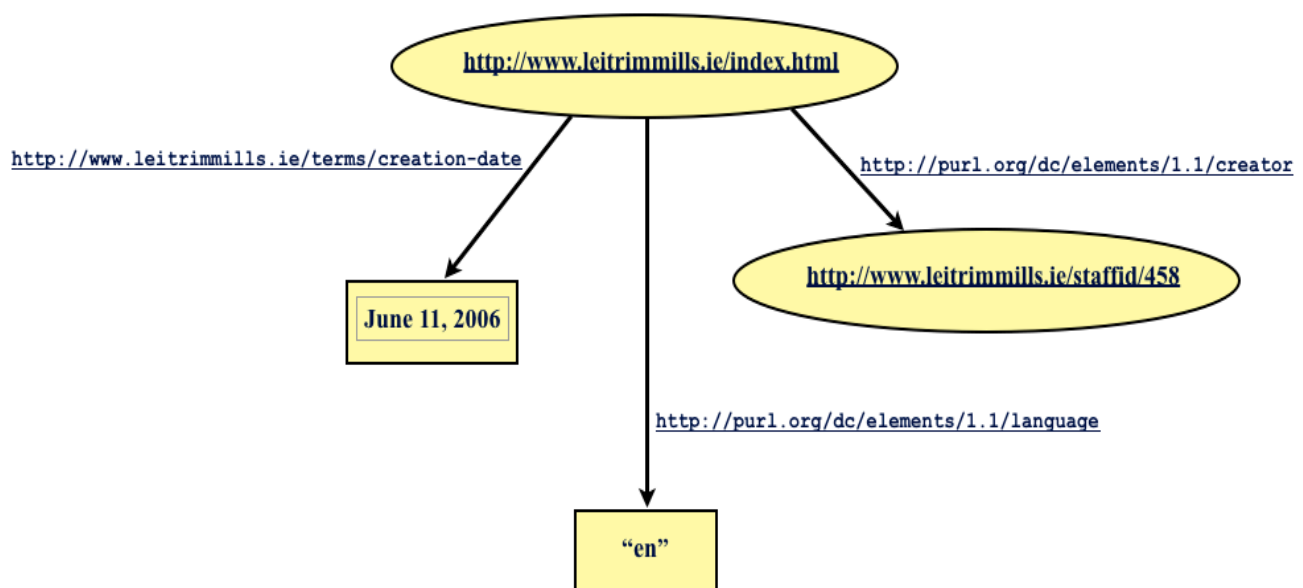


Figure 4-2 RDF graph

4.2.3 RDF Serialisation

The graph in Fig. 4-2 can be shown in triple notation format. This notation takes the format of each statement consisting of a subject, predicate and object in that order. Listing 4-2 is the triple notation form of the graph in Fig. 4-2. (Note that there should be just three lines, one for each statement, but space and clarity of text required the use of more lines):


```

<http://www.leitrimmills.ie/index.html>
  <http://purl.org/dc/elements/1.1/creator>
    <http://www.leitrimmills.ie/staffid/458> .

<http://www.leitrimmills.ie/index.html>
  <http://www.leitrimmills.ie/terms/creation-date>
    "June 11, 2006" .

<http://www.leitrimmills.ie/index.html>
  <http://purl.org/dc/elements/1.1/language>
    "en" .

```

Listing 4-2 The graph in Fig. 4-2 in triple notation format

4.2.3.1 RDF/XML

In order to exchange RDF graphs between applications, the graphs must be serialised to file. RDF/XML [70], an XML syntax defined by the W3C is the standard format for representing RDF triples (as in Listing 4-2). As a result, RDF/XML “*must be supported by all well-behaved Semantic Web applications*” [67]. As RDF/XML is XML based, it is machine processable. However, unlike HTML, the links are URIs and not URL’s. Consequently, RDF/XML can refer to things that are not retrievable on the Web. XML allows you to define namespaces for your elements. These namespaces are URI-like strings and guarantee that your elements are unique across the Web. XML element references are shortened by using qualified names (QNames) i.e. a prefix which stands for the namespace is used instead of the longer namespace string; this is followed by a colon (:) and the local (element) name. Listing 4-3 gives an outline of how a statement is represented in RDF/XML:

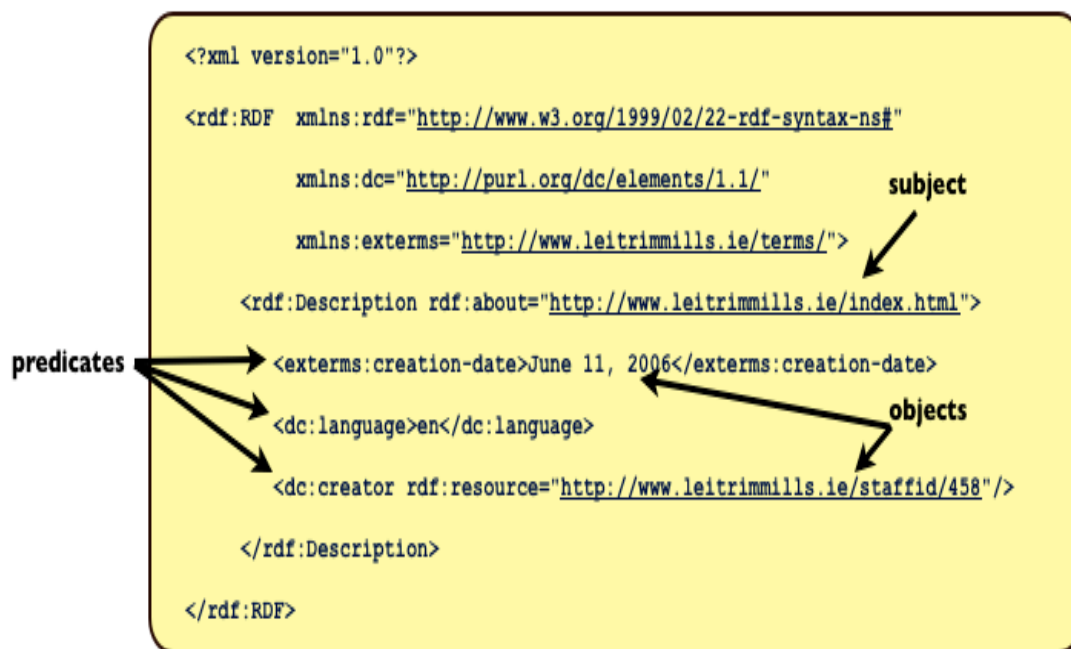
```

<rdf:Description rdf:about="subject">
  <predicate>literal value</predicate>
  ...
  <predicate rdf:resource="object"/>
</rdf:Description>

```

Listing 4-3 The general form of a statement in RDF/XML [67]

Given the form above, Listing 4-4 is the RDF/XML serialisation of the graph in Fig. 4-2.



Listing 4-4 RDF/XML of graph in Fig. 4-2

As this is an XML document, the first line is the XML declaration, which indicates that the content is XML and the version of XML is “1.0”. The root of all RDF documents is the `RDF` element from the “`http://www.w3.org/1999/02/22-rdf-syntax-ns#`” namespace. This `RDF` element (the root) defines the namespace prefixes to be used. As the namespace prefixes (in Listing 4-4) are defined on the root element, the prefixes have global scope within the document. The prefixes defined are for `RDF` (`rdf`), the Dublin Core vocabulary (`dc`) and the example organisation (`exterm`). This means that any tags prefixed with `rdf` come from the “`http://www.w3.org/1999/02/22-rdf-syntax-ns#`” namespace. Any tags prefixed with `dc` come from the Dublin Core vocabulary, which is a set of properties widely used in documenting Internet resources. Typical properties in the Dublin Core vocabulary are Title, Creator, Date, Publisher etc.. The `exterm` prefix points to the organisation’s own tags in the namespace `http://www.leitrimmills.ie/terms/`.

At this point we have the content defined as `RDF/XML` and the namespaces to be used, along with their associated prefixes. Now, we must define the statements: “*an obvious way to talk about any RDF statement is to say it is a description, and that it [the description] is about the subject of the statement*” [69]. Within the root `RDF` element, there are a series of `rdf:Description` elements. A *Description* element is the `RDF/XML` serialisation of a statement. The `rdf:about` attribute states the *subject* of the statement. The children elements of the `Description` element (`creation`, `language` and `creator`) are the *predicates* of the statement. If the property value is a literal, then

the literal will be enclosed in start and end tag names of the property. If the property value is another resource (i.e. URIref) then the property is serialised as an empty-element tag (no closing tag) and the attribute `rdf:resource` is used to refer to this other resource.

Blank nodes [69]

Real world data is often more complicated than the statements presented thus far. Take for example, Sean Kennedy's address: if the address is modelled as a string it is straightforward to graph (and serialise as RDF/XML). If however, one wanted to break down the address further into street, town and county, then this would require a new node (as RDF is a binary relationship). One of the nodes is for "Sean Kennedy" and the second new node is an aggregate node to represent his address. The new node, being a resource would get a new URIref. However, this may occur often and lead to many new nodes and many new URIrefs. These new aggregate nodes may never need to be referred to directly from outside the particular graph, and hence may not require "universal" identifiers.

Blank nodes (or anonymous nodes) take the place of these intermediate nodes. They are given an identifier so as to distinguish them from other blank nodes in the graph. Blank node identifier's have no visibility outside the current graph. If it is expected that a node in a graph will need to be referenced from outside the graph, a URIref should (instead) be assigned to identify it. While a blank node has no identifier in the graph (see Fig. 4-3), an identifier is used when serialising the graph (as XML/RDF for example). Fig. 4-3 is a sample graph with a blank node:

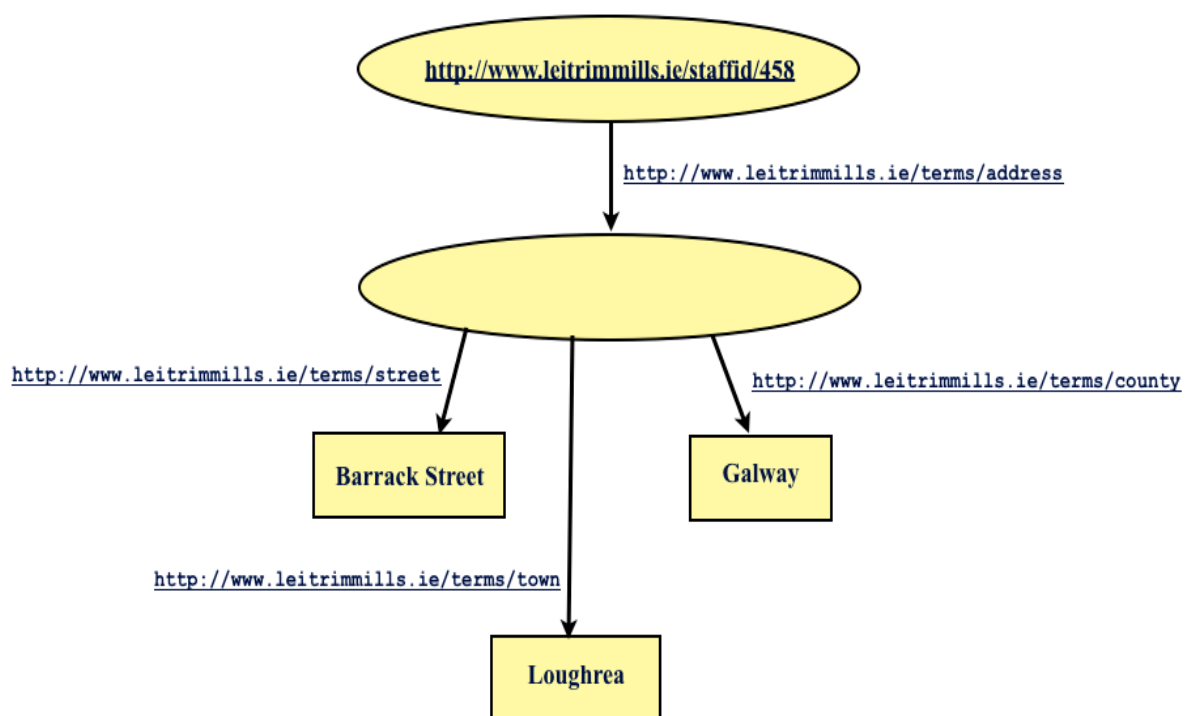


Figure 4-3 RDF graph with a blank node

The blank node is an aggregate node for the three properties representing Sean Kennedy's address. A blank node is referred to in RDF/XML using an `rdf:nodeID` attribute, with a blank node identifier as its value, in places where the `URIref` of a resource would otherwise appear. Specifically, a statement with a blank node as its subject uses `rdf:nodeID` attribute instead of an `rdf:about` attribute in the `rdf:Description` element. Similarly, a statement with a blank node as its object can be written using a property element with an `rdf:nodeID` attribute instead of an `rdf:resource` attribute. Listing 4-5 is the RDF/XML serialisation of the RDF graph in Fig. 4-3.

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:extermis="http://www.leitrimmills.ie/terms/">
  <rdf:Description rdf:about="http://www.leitrimmills.ie/staffid/458">
    <extermis:address rdf:nodeID="SeanKennedyAddress"/>
  </rdf:Description>

  <rdf:Description rdf:nodeID="SeanKennedyAddress">
    <extermis:street>Barrack Street</extermis:street>
    <extermis:town>Loughrea</extermis:town>
    <extermis:county>Galway</extermis:county>
  </rdf:Description>
</rdf:RDF>
```

Listing 4-5 **RDF/XML of graph in Fig. 4-3**

Typed literals [69]

Fig. 4-4 shows how a graph representing Sean Kennedy's age. Note that the literal value "43" is untyped. Does this represent the number 43; the string "43" or the octal value 43 ? Given that the type is not represented in the graph, some out-of-band information would be needed to understand that this value is indeed the decimal/integer value 43.

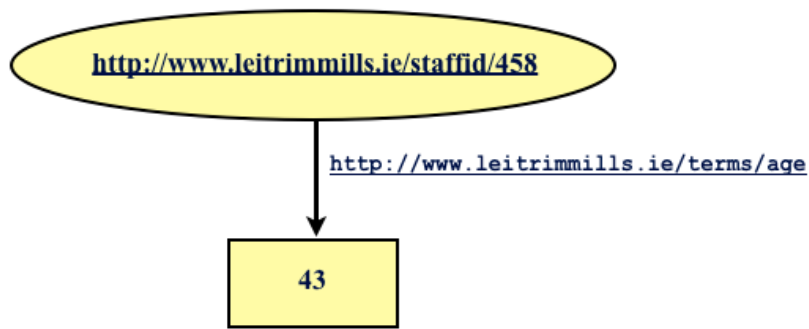


Figure 4-4 RDF graph with the *age* property untyped

Giving the literal value a type and representing that type in the graph can improve this situation. RDF has no built-in datatypes of its own and uses the datatypes defined with XML Schema Datatypes [71]. An RDF typed literal is formed by pairing a string with a URIref that identifies the particular datatype. This results in a single literal node in the RDF graph with the pair as the literal. Fig. 4-5 shows the new graph with the *age* property typed as an integer. Note that the type is stored with the literal value in the graph in the format *literal_value*^^*URIref_of_datatype*.

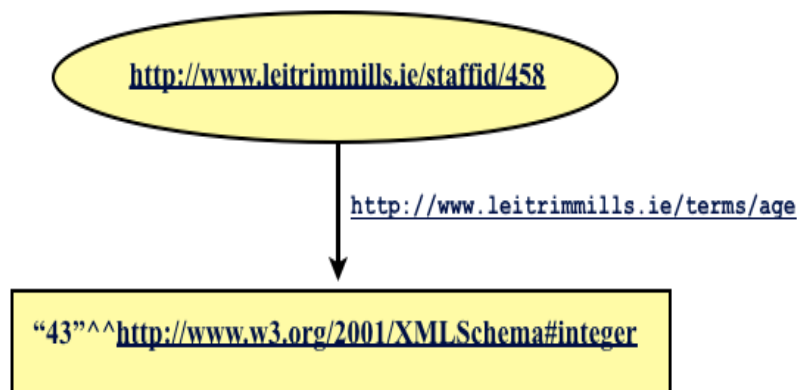


Figure 4-5 RDF graph with the *age* property typed

A typed literal is represented in RDF/XML by adding an `rdf:datatype` attribute specifying the datatype URIref, to the property element containing the literal. Listing 4-6 shows Fig. 4-5 serialised as RDF/XML.

```

<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:extermns="http://www.leitrimmills.ie/terms/">

  <rdf:Description rdf:about="http://www.leitrimmills.ie/staffid/458">
    <extermns:age rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
      43
    </extermns:age>
  </rdf:Description>
</rdf:RDF>

```

Listing 4-6 **RDF/XML of graph in Fig. 4-5**

Listing 4-7 is the RDF/XML representation of the examples so far.

```

<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:extermns="http://www.leitrimmills.ie/terms/">
  <rdf:Description rdf:about="http://www.leitrimmills.ie/index.html">
    <extermns:creation-date>June 11, 2006</extermns:creation-date>
    <dc:language>en</dc:language>
    <dc:creator rdf:resource="http://www.leitrimmills.ie/staffid/458"/>
  </rdf:Description>

  <rdf:Description rdf:about="http://www.leitrimmills.ie/staffid/458">
    <extermns:age rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">43</extermns:age>
    <extermns:address rdf:nodeID="SeanKennedyAddress"/>
  </rdf:Description>

  <rdf:Description rdf:nodeID="SeanKennedyAddress">
    <extermns:street>Barrack Street</extermns:street>
    <extermns:town>Loughrea</extermns:town>
    <extermns:county>Galway</extermns:county>
  </rdf:Description>
</rdf:RDF>

```

Listing 4-7 **Collated RDF/XML of examples so far**

4.2.3.2 N-Triples

As RDF/XML is based on XML, it is tree based and this leads to verbosity and issues when integrating different trees i.e. which root becomes the overall root? N-Triple notation is an RDF-specific syntax. It is not based on XML and therefore does not have to represent the graph as a tree. As a result, it is more concise and readable. Each line in N-Triple notation represents a single statement containing a subject, predicate and object followed by a dot. Subjects, predicates and objects are expressed as absolute URI's enclosed in angle brackets (except for blank nodes and literals). Subjects and objects which are blank nodes are expressed as `_:nodeName`. Object literals are double quoted strings. The file extension is `.nt` and when these files are sent over HTTP the media type `text/plain` is used. Listing 4-8 is the N-Triple representation of Listing 4-7.

```
<http://www.leitrimmills.ie/index.html> <http://www.leitrimmills.ie/terms/creation-date> "June 11, 2006".
<http://www.leitrimmills.ie/index.html> <http://purl.org/dc/elements/1.1/language> "en".
<http://www.leitrimmills.ie/index.html> <http://purl.org/dc/elements/1.1/creator> <http://www.leitrimmills.ie/staffid/458>.
<http://www.leitrimmills.ie/staffid/458> <http://www.leitrimmills.ie/terms/age> "43"^^<http://www.w3.org/2001/XMLSchema#integer>.
<http://www.leitrimmills.ie/staffid/458> <http://www.leitrimmills.ie/terms/address> _:SeanKennedyAddress.
_:SeanKennedyAddress <http://www.leitrimmills.ie/terms/street> "Barrack Street".
_:SeanKennedyAddress <http://www.leitrimmills.ie/terms/town> "Loughrea".
_:SeanKennedyAddress <http://www.leitrimmills.ie/terms/county> "Galway".
```

Listing 4-8 N-Triple notation of Listing 4-7

4.2.3.3 Turtle (Terse RDF Triple Language)

While N-Triple notation is conceptually simple, it is verbose and leads to a lot of repetition. If the dataset is large this can lead to inefficiencies in transmission and parsing of the dataset. Turtle, a notation defined by the W3C [72], is a superset of N-Triples that condenses much of the repetition in the N-Triple format. For example, Turtle allows us to define a URI prefix much like XML allows us to define namespace prefixes. Using the N-Triple format requires the URI of the subject to be used regardless of the fact that we may be saying many things about that same subject. Turtle reduces this repetition by allowing you to combine several statements about the same subject by using a semi-colon after the first statement. This means that the subject is stated only once - you only need to state the predicate and object for other statements relating to the same subject [73]. Turtle uses the media type `text/turtle` and the file extension `.ttl`.

4.2.3.4 Notation 3 (N3)

N3 is a superset of Turtle i.e. all of the shorthand statements introduced by Turtle are supported by N3. Thus a library capable of handling N3 will handle Turtle (and also N-Triples). N3 uses the file suffix `.n3` and the media type `text/n3`.

Fig. 4-6 shows the relationship between the N-Triples, Turtle and N3 notations.

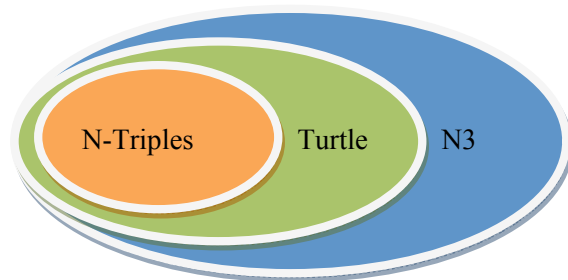


Figure 4-6 N-Triples, Turtle and N3 relationships

Listing 4-9 is the N3 version of Listing 4-7. Note that, in Listing 4-9, the square angle brackets [] represent a blank node and the ; represents “AND” i.e. the subject is the same but the predicate and object are different. This saves on repeating the subject.

```
@prefix extermis: <http://www.leitrimmills.ie/terms/>.
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

<http://www.leitrimmills.ie/index.html>      dc:creator      <http://www.leitrimmills.ie/staffid/458>;
                                              dc:language      "en";
                                              extermis:creation-date  "June 11, 2006".

<http://www.leitrimmills.ie/staffid/458>    extermis:address  [
                                              extermis:county "Galway";
                                              extermis:street "Barrack Street";
                                              extermis:town  "Loughrea"
                                              ];
                                              extermis:age      43 .
```

Listing 4-9 N3 notation of Listing 4-7

4.3 Adding semantics to RDF data

RDF is all about describing information: it allows you to make statements about resources that are identified using URIs. These statements take the form of triples that associate a resource (subject) with a value (object) using a property (predicate). Flexible though it is, RDF lacks explicit support for specifying the semantics, or meaning, behind the descriptions [67]. How to add semantics to RDF data is the purpose of this section.

4.3.1 RDFS (Resource Description Framework Schema)

RDFS, a W3C specification [74] is a vocabulary (i.e. a well-defined set of terms) for describing properties, classes and their inter-relationships in a hierarchical fashion. To describe “kinds of things”, RDFS provides classes. Classes are analogous to the generic concept of types or categories.

RDFS provides extensions to RDF that enable us to describe classes of things and their properties and to indicate which classes and properties will be used together: “*RDF Schema provides a type system for RDF*” [69]. The extensions have a well-defined meaning thereby lending semantics. For example, if we have the triple:

```
ex:Van rdfs:subClassOf ex:MotorVehicle
```

it implies the following [67]:

- `ex:Van` is a specialisation (a subclass of) of `ex:MotorVehicle`
- each member of `ex:Van` must pass all restrictions imposed by `ex:MotorVehicle`
- instances of `ex:Van` automatically become instances in `ex:MotorVehicle`
- properties of `ex:MotorVehicle` are inherited by `ex:Van`

Note the use of shorthand QNames to refer to URIs. This is the shorthand used in the RDF primer [69] and will be used here also in the interests of brevity.

Semantic classes versus OOP classes

RDFS classes are similar but not exactly the same as classes in an OOP language such as Java. It is similar in that, using RDFS, you can arrange classes and properties into generalisation/specialisation hierarchies. Also, resources that belong to a class are called *instances*. However, in OOP, an object is an instance of the class i.e. the class is the blueprint for the object. This means that when an object is created in OOP, it contains all the members (methods and variables) of that class. In OOP, when you know the class an object is derived from, you know which properties it will contain. This is not the case in semantic systems as “*properties are defined independently*” [73]. RDFS defines properties in terms of the classes to which they apply (`rdfs:domain` and `rdfs:range` properties). For example, we could define the `ex:author` property to have a domain of `ex:Document` and a range of `ex:Person` (a document has an author who is a Person), whereas in OOP one might typically define a class `ex:Book` with an attribute called `ex:author` of type `ex:Person`. This property-centric approach enables properties to be arbitrarily assigned to classes.

Classes

The RDFS vocabulary is defined in the `http://www.w3.org/2000/01/rdf-schema#` namespace which, by convention is given the `rdfs:` prefix. In RDF Schema, a class is any resource having an `rdf:type` property whose value is the resource `rdfs:Class`. For example, the following triple defines the class `MotorVehicle`:

```
ex:MotorVehicle rdf:type rdfs:Class .
```

To describe an instance of a class (as opposed to a class itself) requires the following triple:

```
ex:CompanyCar    rdf:type    rdfs:MotorVehicle .
```

Note that a resource can be an instance of more than one class [69] (see `MiniVan` in Listing 4-10). To explain how RDFS enables a taxonomy (a hierarchy of terms) to be built up, the example from [69] is useful.

Fig. 4-7 shows the UML diagram of the hierarchy.

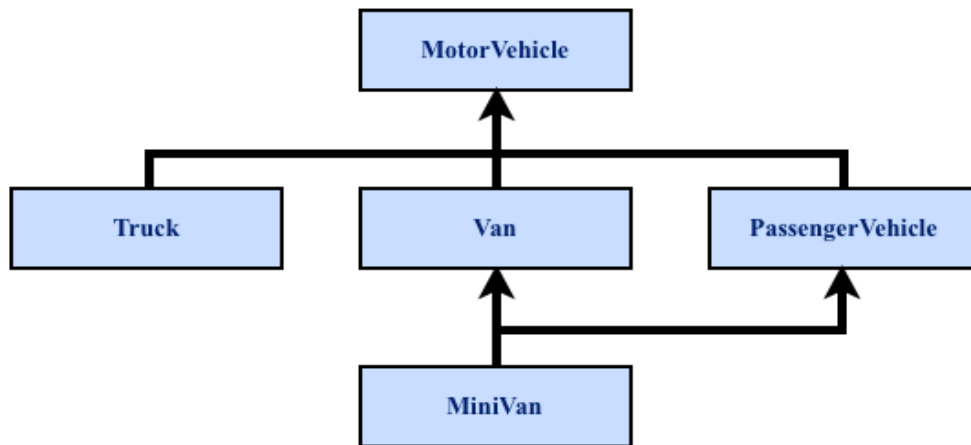


Figure 4-7 UML Diagram of the hierarchy

This is represented in Turtle/N3 format in Listing 4-10. In the example, the `ex:` prefix stands for the URIref “`http://www.example.org/schemas/vehicles`”.

```

ex:MotorVehicle    rdf:type    rdfs:Class .
ex:Truck           rdf:type    rdfs:Class .
ex:Van             rdf:type    rdfs:Class .
ex:PassengerVehicle rdf:type    rdfs:Class .
ex:MiniVan         rdf:type    rdfs:Class .
ex:Truck           rdfs:subClassOf ex:MotorVehicle .
ex:Van             rdfs:subClassOf ex:MotorVehicle .
ex:PassengerVehicle rdfs:subClassOf ex:MotorVehicle .
ex:MiniVan         rdfs:subClassOf ex:Van .
ex:MiniVan         rdfs:subClassOf ex:PassengerVehicle .
  
```

Listing 4-10 Turtle/N3 serialisation of hierarchy [69]

In Listing 4-10, the classes are defined first using the `rdf:type` property with `rdfs:Class` as the object of the statement. The `rdfs:subClassOf` is then used to create a hierarchy within the classes with the subject of the `rdfs:subClassOf` property becoming a subclass of the statement object.

Note that the `ex:MiniVan` class is a subclass of both `ex:Van` and `ex:PassengerVehicle`.

Listing 4-11 shows how this is serialised in RDF/XML:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://example.org/schemas/vehicles">

  <rdfs:Class rdf:ID="MotorVehicle"/>

  <rdfs:Class rdf:ID="PassengerVehicle">
    <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="Truck">
    <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="Van">
    <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="MiniVan">
    <rdfs:subClassOf rdf:resource="#Van"/>
    <rdfs:subClassOf rdf:resource="#PassengerVehicle"/>
  </rdfs:Class>

</rdf:RDF>
```

Listing 4-11 **RDF/XML format of hierarchy [69]**

There are two things to note from Listing 4-11. The first point to note is the use of the `rdf:ID` attribute instead of the `rdf:about` attribute. Both serve the same purpose but with `rdf:ID`, one creates a label that can be referred to from this or other RDF documents. The `rdf:ID` attribute is similar to the `ID` attribute in XML and HTML, in that it defines a unique name relative to the current base URI. For example, assuming the base URI (the identifier and location for the RDF/XML file) for Listing 4-11 is `http://www.example.org/schemas/vehicles`, then the `Van` resource is available externally (to the RDF document) as

`http://www.example.org/schemas/vehicles#Van` and internally as `#Van`. The second point is the use of `rdfs:Class` where normally we would see `rdf:Description`. This is an RDF/XML abbreviation for “typed nodes” (instances of a class). *“In this abbreviation, the `rdf:type` property and its value are removed, and the `rdf:Description` element for the node is replaced by an element whose name is the QName corresponding to the value of the removed `rdf:type` property”* [69]. So, for example, the `Truck` resource segment from Listing 4-11 :

```
<rdfs:Class rdf:ID="Truck">
  <rdfs:subClassOf rdf:resource="#MotorVehicle" />
</rdfs:Class>
```

Listing 4-12 Truck resource segment from Listing 4-11

could also be written (in the more verbose format):

```
<rdf:Description rdf:ID="Truck">
  <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class" />
  <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdf:Description>
```

Listing 4-13 Alternative RDF/XML syntax for Truck resource

Properties

RDFS properties specify the relationships between subject resources and object resources [74]. The most important properties defined by RDFS are `rdf:type`, `rdfs:subClassOf` (see above example), `rdfs:domain` and `rdfs:range`. The `rdf:type` property states that a triple of the form *R rdf:type C*, means that resource R is an instance of class C.

Domain and Range

When defining a property, one can indicate which types of resources have the property. The collection of types that are being described by the property (the subject of the triple), are known as the *domain* of the property. The values of the property (the object of the triple) are known as the *range* [73]. The W3C explains them as follows [74]:

- `rdfs:domain` – triples of the form *P rdfs:domain C* state that P is an instance of `rdf:Property` (a property), C is an instance of `rdfs:Class` (a class) and that the *subjects* of triples whose predicate is P are instances of C [74]. In other words, `rdfs:domain` declares the class of the subject that use this property.
- `rdfs:range` – triples of the form *P rdfs:range C* state that P is an instance of `rdf:Property` (a property), C is an instance of `rdfs:Class` (a class) and that the *objects* of triples whose predicate is P are instances of C [74]. In other words, `rdfs:range` declares the class of the datatype or object that use this property.

The following example demonstrates domain and range more clearly:

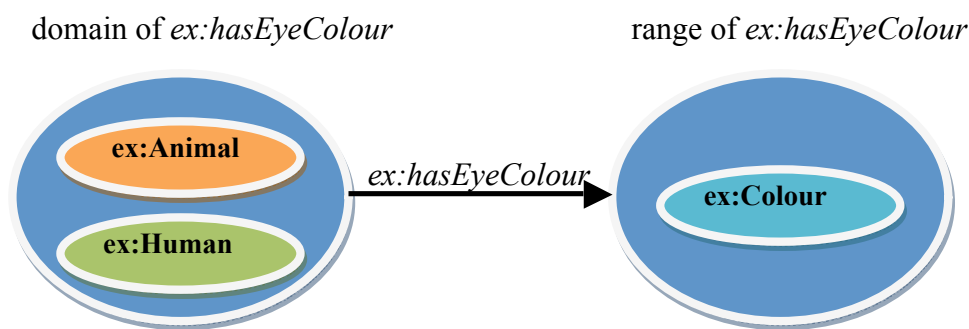


Figure 4-8 Domain and Range of a property expressing eye colour [73]

The model in Fig. 4-8 represents information about a person’s eye colour. This can be stated in N3 as follows (where *ex:* is the prefix for *example.org* namespace):

<code>ex:hasEyeColour</code>	<code>rdf:type</code>	<code>rdf:Property</code>
<code>ex:hasEyeColour</code>	<code>rdfs:domain</code>	<code>ex:Animal</code>
<code>ex:hasEyeColour</code>	<code>rdfs:domain</code>	<code>ex:Human</code>
<code>ex:hasEyeColour</code>	<code>rdfs:range</code>	<code>ex:Colour</code>

Listing 4-14 Domain and Range example [73]

In Listing 4-14, the property `ex:hasEyeColour` is defined initially to be a property type. The property is then given a domain of `ex:Animal` and `ex:Human`. This means that the subject in statements where this property is used, is of type `ex:Animal` or of type `ex:Human`. Lastly, the property is given a range of `ex:Colour`. This means that the objects of the statements are of type `ex:Colour`.

4.3.2 Ontologies

RDFS is a fundamental building block of ontologies. Using RDFS you can define generalisation/specialisation hierarchies for both classes and properties, assert class membership and specify datatypes. Before we look at ontologies and how to create them using OWL, the Web Ontology Language, we will first define an ontology: “*An ontology uses a specific vocabulary of terms to define concepts and the relationships between them for a specific area of interest, or domain*” [67]. While RDFS is perfectly acceptable to build ontologies, OWL enables us to build more expressive ontologies [67].

4.3.3 OWL Web Ontology Language

OWL extends the RDFS vocabulary by introducing extra terms that have special semantics associated with them. These terms are used to give meaning to the data they describe. The OWL vocabulary (terms) is defined in the <http://www.w3.org/2002/07/owl#> namespace, which, by convention is given the `owl:` prefix. OWL 2, delivered in late 2009, is the latest version [75]. OWL ontologies model domain knowledge and the language is designed to be flexible, allowing users to choose from a number of predefined subsets, called “profiles”. These profiles, which are discussed later, trade off expressivity with computational complexity.

OWL is built on top of RDF and RDF describes resources using URIs i.e. they are inherently distributed. Consequently, how information is represented (the knowledge model) of OWL is also distributed. To be able to make valid inferences (i.e. where new data is inferred from existing data) in this distributed knowledge model, OWL makes two assumptions: the Open World Assumption and the No Unique Names Assumption.

Open World Assumption [67]

The Open World Assumption states that the truth of a statement is independent of whether it is known or not. In other words, not knowing whether a statement is explicitly true, does not imply that the statement is false. This is best explained by contrasting the open world assumption with its opposite: the closed world assumption. In this example, we have a database table *customer* (note: databases presume the closed world assumption). In the closed world assumption, information is considered complete and true and any unknown data is considered false. For example, if we have a customer “John Smith” in the *customer* table then we know we have a customer by that name. If “John Smith” does not exist in the *customer* table then we know we have no customer of that name. In the open world assumption, if we do not have a customer “John Smith” then that does not mean anything about him as a customer i.e. we cannot assume that he is not a customer. The open world assumption impacts OWL in the way information is inferred. Reasoning is performed on known data only i.e. the absence of information cannot be used to infer the presence of other information.

No Unique Names Assumption [67]

Resources are described using URIs. Given the distributed nature and massive scale of the Web, it is unrealistic to assume that everyone uses the same URI to identify a specific resource. A simple example of this is email: one can reach the resource “Sean Kennedy” using either *skennedy@ait.ie* or

sean@leitrimmills.ie. Therefore, this assumption states: you cannot assume that resources identified by different URIs are different.

OWL Profiles

OWL has several profiles (or sub-languages) that balance expressiveness and computational efficiency [76]:

- **OWL-Full** not a sub-language but rather, is the full unrestricted OWL specification. It places no restrictions on RDF and thereby maintains RDF's ability "*to say anything about anything*" [67]. This flexibility comes at a cost: OWL-Full is not computationally decidable i.e. there are no guarantees that all conclusions can be computed or that the computations will ever finish.
- **OWL-DL** is a computationally efficient alternative to OWL-Full. It uses the same OWL-Full constructs but with some restrictions in place to ensure computational completeness and decidability. For example, the URI's used for classes, properties and individuals must be mutually disjoint i.e. the same URI cannot be used for a class, a property or an individual.
- **OWL-Lite** is a further subset of OWL-DL with additional cardinality constraints (0 or 1 only). OWL-Lite is used to create classification hierarchies with straightforward constraints.

4.3.4 OWL Ontologies [67]

OWL ontologies are commonly stored as RDF documents. There is no official partition between ontologies and the data they describe (i.e. the instances), although it is common to maintain the ontologies and instances separately. Ontologies can contain the following elements: header, annotations, datatypes, classes, individuals and properties. These are discussed in turn.

Ontology header

This is an optional element that usually contains annotation properties that describe the ontology. This is where other ontologies can be imported using the `owl:imports` property. Labels and comments can be inserted also, using the `rdfs:label` and `rdfs:comment` properties.

Annotations

Annotation properties are semantic-free properties and are used mainly by tools and user interface environments. The properties outlined above, namely, `rdfs:label` and `rdfs:comment` are examples of annotation properties.

Datatypes

Datatypes represent ranges of data values. Pre-defined datatype examples are string and integer. OWL 2 allows us to define our own custom datatypes by defining value range (or facet) restrictions. For example, one could create a datatype to represent the valid age for driving in the Republic of Ireland i.e. 17.

Classes

A class represents the category of “thing” being modelled. The `rdf:type` property is used to assign class membership and the `owl:Class` resource is the resource that specifies that the subject of the statement is a class. For example:

```
ex:Person    rdf:type    owl:Class.
```

states that `Person` is a class.

Individuals

The individuals of a class are the instances of a class. Extending the example above:

```
ex:Person    rdf:type    owl:Class.  
ex:John      rdf:type    ex:Person.
```

states that `John` is an instance of the class `Person`; in other words, `John` is a person. The “class extension” is the set of individuals (instances) of a class. Note that, “*containment in a class extension does not preclude an individual from being a member of any other class*” [67]. OWL individuals can have any structure i.e. they are not restricted to their class types (unlike OOP instances, which get their information from their class types).

Properties

Properties establish relationships between resources. They are the predicates in the subject-predicate-object triples. The two principle types of properties in OWL are:

- `owl:ObjectProperty` – used when defining a relationship between individuals
- `owl:DatatypeProperty` – used when defining a relationship between an individual and a literal value

Both are subclasses of the RDF class `rdf:Property`. Listing 4-15 below states that `John` and `James` are cousins and that `James` is known as “`Jim`”:


```

ex:Person rdf:type owl:Class.           #Person is a class
ex:John   rdf:type ex:Person.             #John is an individual, type Person
ex:James  rdf:type ex:Person.             #James is a person
ex:name   rdf:type owl:DatatypeProperty. #name refers to a literal
ex:cousin rdf:type owl:ObjectProperty.  #cousin refers to an individual

ex:James ex:name "Jim";
          ex:cousin ex:John.

```

Listing 4-15 DatatypeProperty and ObjectProperty example

Important Classes and Properties

The full vocabulary of OWL uses URIs from the RDF, RDFS and OWL namespaces, as well as XML Schema literal definitions. When creating an ontology, the following classes (appearing as objects in a triple) are important [73]:

<code>owl:Thing</code>	The class of all things in OWL. All classes implicitly subclass this class and all instances (individuals) are implicitly instances of <code>owl:Thing</code> . This is similar to <code>Object</code> in Java.
<code>owl:Class</code>	All classes are instance of this class i.e. when describing a class, this is the object value for an <code>rdf:type</code> predicate.
<code>owl:DatatypeProperty</code>	defines a property where the ranges are literals
<code>owl:ObjectProperty</code>	defines a property where the ranges are instances of <code>owl:Class</code>

4.4 Querying RDF data

James Hendler, a leading figure in the Semantic Web, stated in [77] that there were three core areas in the Semantic Web:

- a data format that can include links i.e. RDF
- domain descriptions i.e. OWL ontologies
- a language for querying the data

The language for querying RDF datastores, known as SPARQL, is the subject of this section.

4.4.1 SPARQL

SPARQL, a W3C standard [78], is a “*Semantic Web extension of SQL*” [77]. SPARQL is both a query language and a protocol [67]. The protocol refers to how a SPARQL client (such as one accessible via a browser) interacts with a SPARQL endpoint/processor e.g. <http://dbpedia.org/sparql>. The endpoint is a service that follows the protocol i.e. the endpoint accepts and processes SPARQL queries and returns the results in various formats depending on the query form [67]. The W3C has created a special XML format for the results called: SPARQL Query Results XML format [79]. This specification [79] defines a document root element called `sparql`, which has two sub-elements, `head` and a results element (either `results` or `boolean`) in that order. Whether the results element is `results` or `boolean` depends on the query form (`results` for SELECT and `boolean` for ASK) .

SPARQL supports four query forms [67] [78]:

SELECT – SELECT is similar to SQL’s SELECT statement. In SPARQL, SELECT instructs endpoints to bind RDF terms such as blank nodes, URIrefs and literals to variables based on a given graph pattern (the WHERE clause). Variables are prefixed with ? or \$. The WHERE clause specifies our graph pattern (also known as triple patterns). Triple patterns are very similar to RDF triples except that the subject, predicate or object may be a variable. Assuming the subgraph specified in the WHERE clause is matched in the RDF graph we are searching, the variables are bound and the bindings returned. Here is a straightforward example from [78] using the Turtle/N3 format (note the “_:blankNodeID”):

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Johnny Lee Outlaw" .
_:a foaf:mbox <mailto:jlow@example.com> .
_:b foaf:name "Peter Goodguy" .
_:b foaf:mbox <mailto:peter@example.org> .
_:c foaf:mbox <mailto:carol@example.org> .
```

Listing 4-16 (a) SPARQL SELECT (data)

```

PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbbox
WHERE
{
  ?x foaf:name ?name .
  ?x foaf:mbbox ?mbbox }

```

Listing 4-16 (b) SPARQL SELECT (syntax)

name	mbbox
"Johnny Lee Outlaw"	<mailto:jlow@example.com>
"Peter Goodguy"	<mailto:peter@example.org>

Table 4-1 SPARQL SELECT (results)

CONSTRUCT - CONSTRUCT allows you to reformulate the bound variables into any RDF graph you choose. This allows us to transform from one graph into a completely different one. The following example from [78] uses Friend Of A Friend (FOAF) information to create vcard properties:

```

@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:a foaf:name "Alice" .
_:a foaf:mbbox <mailto:alice@example.org> .

```

Listing 4-17 (a) SPARQL CONSTRUCT (data)

```

PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX vcard:   <http://www.w3.org/2001/vcard-rdf/3.0#>
CONSTRUCT
{
  <http://example.org/person#Alice> vcard:FN ?name }
WHERE
{
  ?x foaf:name ?name }

```

Listing 4-17 (b) SPARQL CONSTRUCT (syntax)

```
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
<http://example.org/person#Alice> vcard:FN "Alice" .
```

Listing 4-17 (c) SPARQL CONSTRUCT (results)

ASK - ASK is used to determine whether a particular graph exists or not. If the graph exists then the boolean *true* is returned otherwise *false* is returned.

```
@prefix foaf:      <http://xmlns.com/foaf/0.1/> .
_:a foaf:name      "Alice" .
_:a foaf:homepage  <http://work.example.org/alice/> .
_:b foaf:name      "Bob" .
_:b foaf:mbox       <mailto:bob@work.example> .
```

Listing 4-18 (a) SPARQL ASK (data)

```
PREFIX foaf:      <http://xmlns.com/foaf/0.1/>
ASK { ?x foaf:name "Alice" }
```

Listing 4-18 (b) SPARQL ASK (syntax)

This ASK command returns *true*.

DESCRIBE - DESCRIBE returns an RDF graph. The endpoint decides what RDF data is returned with limited input from the client. DESCRIBE is the least used of the forms [67].

4.5 Semantically annotated Web Services

While the Semantic Web relates to semantically annotating data derived from the human-readable Web, Semantic Web Services (SWS) relates to annotating functionality (as opposed to data) [115] i.e. “*syntactic service descriptions that have been augmented with machine-processable semantic information*” [121]. In this section, the technologies that provide semantic meaning to both traditional Web Services (the WS-* stack) and RESTful Web Services are discussed. Fig 4-9 shows the SWS and semantically annotated RESTful Web Services (RWS) stacks, side by side. With regard to SWS, WSDL is used for machine automation (from a syntactic viewpoint). SAWSDL enables us to non-intrusively annotate the purely syntactic WSDL descriptions with pointers to the

(conceptually higher) semantic layer [40][109][121]. From the RESTful Web Services viewpoint, hRESTS and MicroWSMO accomplish what WSDL and SAWSDL accomplish respectively. These technologies form the basis of this section. The WSMO-Lite ontology is common to both.

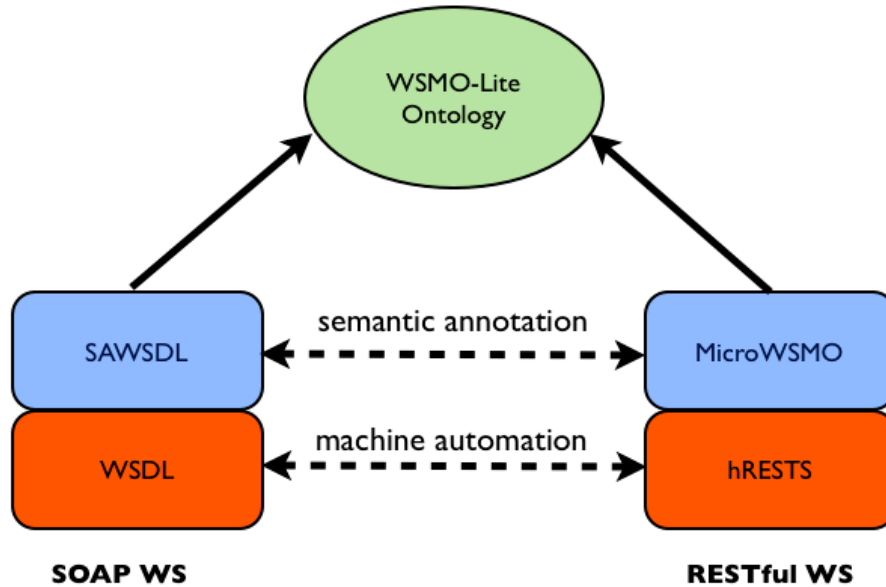


Figure 4-9 Semantic Layers [80]

An example is used to explain both approaches. The Web Service is a Hotel Web Service that supports one operation: a retrieval of hotel details given a hotel identifier. The RESTful Web Service description is in HTML for the following reasons:

- a) there is no commonly accepted description standard yet for RESTful Web Services [101][122]
- b) there is a multitude of plain HTML service descriptions already on the Web [121][122]

Netbeans was used to create the WSDL description of the SOAP service (.wsdl) and Dreamweaver used to create the RESTful Web Service description (.html). The tool used to annotate the WSDL description with SAWSDL is Core Dashboard [81]. The tool used to annotate the HTML description with hRESTS and MicroWSMO is SWEET (hosted on Core Dashboard). Core Dashboard [81] is a suite of tools developed by the Knowledge Media Institute at Milton Keynes in England. These tools enable semantic annotation of Web Service descriptions, both the WS-* and RESTful variety.

4.5.1 Semantic Web Services

The traditional WS-* stack provides WSDL to describe the syntax of the operation and the structure of the data sent and received. WSDL provides no facility to apply semantics to either the operation or the data. SWS are a suite of technologies that semantically annotate traditional SOAP-based Web Service descriptions. With regard to Fig. 4-9, we are referring to SAWSDL and WSMO-Lite

technologies. SAWSDL is discussed first. WSMO-Lite is common to both stacks and is discussed along with the technologies that enable semantic annotation of RESTful Web Services.

4.5.1.1 SAWSDL

SAWSDL (Semantic Annotations for WSDL and XML Schema) [82] is a W3C specification that “*defines how semantic annotation is accomplished using references to semantic models, e.g. ontologies*” [82]. SAWSDL is a W3C recommendation since August 2007. SAWSDL builds mainly on WSDL 2.0 but also supports WSDL 1.1. SAWSDL is independent of any semantic technology and assumes that semantic concepts can be pointed to via URIs [109]. While SAWSDL itself does not provide any specific semantics, it does enable us to annotate the purely syntactic WSDL descriptions with pointers to semantic concepts. “*SAWSDL extends WSDL with pointers to semantics that are crucial for achieving automation*” [109]. Conceptually SAWSDL resides above WSDL (see Fig. 4-9). With WSDL, one is concerned with service signature, service location and the protocol to use when invoking the Web service. SAWSDL, on the other hand, is concerned with mapping elements from WSDL to a higher conceptual level i.e. the ontology level. To achieve this SAWSDL defines the following extension attributes [82]:

- `modelReference` – an extension attribute that can be applied to any WSDL or XML Schema element in order to point to, via a set of URIs to one or more semantic concepts [109].
- `liftingSchemaMapping` – added to element declarations and type definitions for specifying the mapping from a messages XML structure to the semantic model
- `loweringSchemaMapping` – added to element declarations and type definitions for specifying the mapping from the semantic layer to XML

The `liftingSchemaMapping` and `loweringSchemaMapping` attributes enable data mediation between Web services. Take for example, two Web services that are trying to communicate where the output from the first does not match the input of the second. The output from the first Web service can be lifted to the ontology layer by that Web services `liftingSchemaMapping`. Note that the second Web service is also pointing at the same ontology concept. The second Web service provides a `loweringSchemaMapping` that lowers from the (shared/common) ontology to the input format it requires [83].

Fig 4-10 shows the Hotel WS WSDL file being semantically annotated using Core Dashboard:

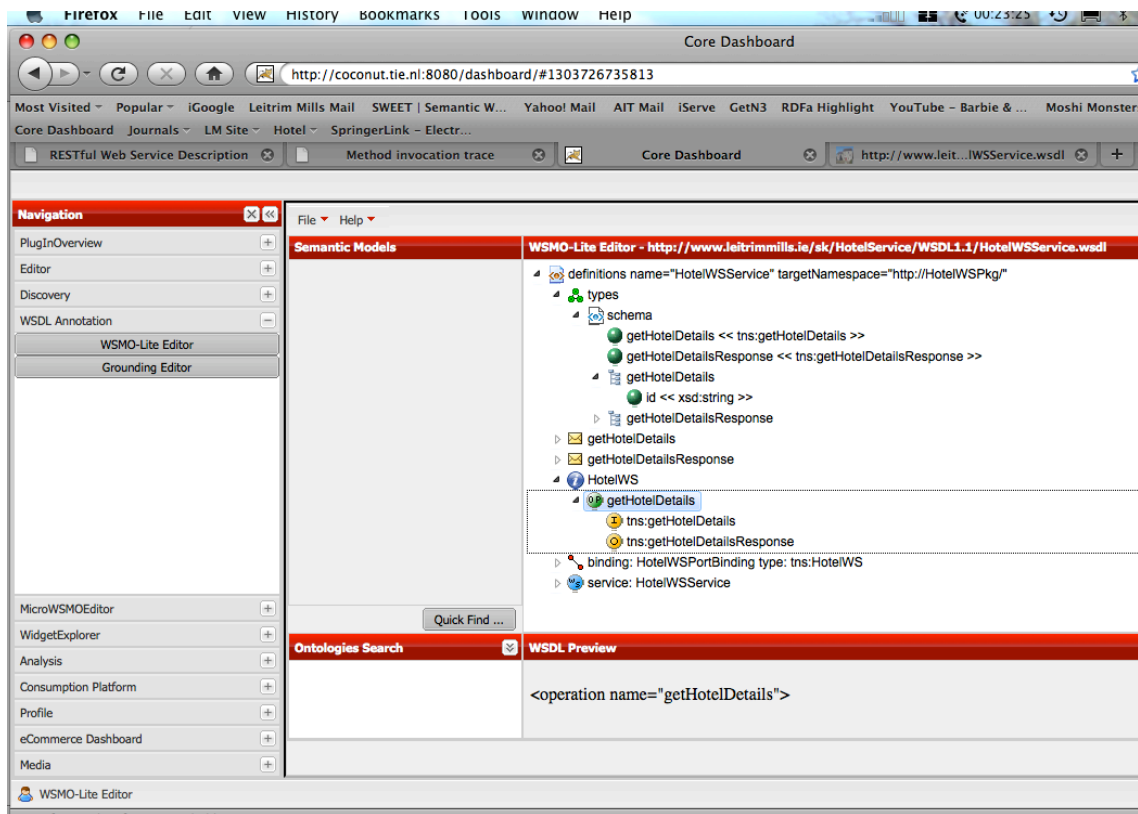


Figure 4-10 SAWSDL annotation using Core Dashboard

Listing 4-19 is the resultant SAWSDL file with the model references pointing to the semantic concepts highlighted in red. Note that there are no `liftingSchemaMapping` or `loweringSchemaMapping` elements as this example did not require lifting/lowering to/from the semantic layer.

Note: Listing 4-19 is based on WSDL 1.1 (use of `portType` for the abstract interface). SAWSDL was primarily built for WSDL 2.0 but does support the more prevalent WSDL 1.1 version. Note that in the operation “*getHotelDetails*” of the portType “*HotelWS*” that the `modelReference` is an attribute of the element `attrExtensions`. This is because the WSDL 1.1 schema prohibits attribute extensions on the operation element. It does however allow element extensions. Thus, SAWSDL defines the element `attrExtensions` to carry extension attributes in places where only element extensibility is allowed [109].

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:sawSDL="http://www.w3.org/ns/sawSDL"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://HotelWSPkg/"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="HotelWSService" targetNamespace="http://HotelWSPkg/">
  <types>
    <xsd:schema>
      <xsd:element name="getHotelDetails" type="tns:getHotelDetails"/>

      <xsd:element name="getHotelDetailsResponse" type="tns:getHotelDetailsResponse"/>
      <xsd:complexType name="getHotelDetails">
        <xsd:sequence>
          <xsd:element minOccurs="0" name="id"
            sawSDL:modelReference="http://www.example.com/ontologies/Accommodation#Hotel"
            type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="getHotelDetailsResponse">
        <xsd:sequence>
          <xsd:element minOccurs="0" name="return" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </types>
  <message name="getHotelDetails"><part element="tns:getHotelDetails" name="parameters"/></message>
  <message name="getHotelDetailsResponse"><part element="tns:getHotelDetailsResponse" name="parameters"/></message>
  <portType name="HotelWS">
    <operation name="getHotelDetails">
      <input message="tns:getHotelDetails"/>
      <output message="tns:getHotelDetailsResponse"/>
      <sawSDL:attrExtensions sawSDL:modelReference="http://www.example.com/ontologies/Accommodation#SearchService"/>
    </operation>
  </portType>
  <binding name="HotelWSPortBinding" type="tns:HotelWS">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getHotelDetails">
      <soap:operation soapAction=""/>
      <input><soap:body use="literal"/></input>
      <output><soap:body use="literal"/></output>
    </operation>
  </binding>
  <service name="HotelWSService">
    <port binding="tns:HotelWSPortBinding" name="HotelWSPort">
      <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
    </port>
  </service>
</definitions>

```

Listing 4-19 SAWSDL annotation using Core Dashboard [81]

4.5.2 Semantic annotation of RESTful Web Services

RESTful Web Services (RWS) have several options for describing themselves. One option is Web Application Description Language or WADL, a W3C submission [84]. WADL is an XML-based format that provides a machine-readable description of HTTP-based applications. WADL “works

similarly to WSDL with SOAP” [136]. Another alternative is WSDL 2.0 [32] with the HTTP binding. However, *Kopecky et al* in [85], describe HTML as the “*medium of choice*” for describing RWS and claim that WSDL 2.0 and WADL are not gaining traction due to their perceived complexity.

In this section, the technologies of Microformats/poshformats, hRESTS, MicroWSMO, GRDDL (Gleaning Resource Descriptions from Dialects of Languages), WSMO-Lite, SA-REST and RDFa (Resource Description Framework in Attributes) are discussed. Microformats are discussed because hRESTS is a microformat for annotating key parts of an (X)HTML document so that the RWS description can be elicited. MicroWSMO is an extension to hRESTS that annotates key elements of the RWS description to point to the semantic layer. GRDDL is a markup format used to extract RDF triples from the hRESTS/MicroWSMO annotated (X)HTML page. The technology for specifying ontologies in Web Services i.e. WSMO-Lite is also covered. Note that WSMO-Lite applies to both SOAP based Web Services and RESTful Web Services. SA-REST, a poshformat (a less formal microformat) for describing RWS is also outlined. Lastly, RDFa, a W3C specification for inserting RDF data directly into (X)HTML is discussed.

4.5.2.1 Microformats/Poshformats

Microformats are carefully designed (X)HTML class names that extend the semantics of (X)HTML so authors can publish semantic content [86]. Microformats use for example, the *class* attribute in (X)HTML tags to add semantic information to Web content and the result is an (X)HTML format that is readable by both machines and humans. Formats such as *vCard* [87] (for publishing/sharing contact information) and *vCalendar* [88] (for publishing/sharing event information) have proven popular.

A poshformat, where *posh* stands for Plain Old Semantic HTML, is similar to a microformat in that HTML attributes are used to add semantics to the Web page. However, a poshformat [89] is more a once-off, ad-hoc format whereas microformats are more formal and have gone through a rigorous community process. Microformats are therefore a subset of poshformats.

4.5.2.2 hRESTS

RESTful Web service descriptions are generally described in plain, unstructured HTML [80][121]. hRESTS (HTML for RESTful Services) [80] is a microformat aimed at making these HTML based Web APIs machine readable by annotating the key information on the HTML page. In other words, “*hRESTS provides a machine-readable service description within a human-readable document*” [121]. hRESTS defines a set of classes and uses the *class* attribute of XHTML to refer to them. This

identifies the key information of the service description “*effectively creating an analogue of WSDL*” [80]. The hRESTS classes defined are:

- `service` - indicates that this block describes a RESTful Web service
- `label` - the human-readable name of the RESTful Web service
- `operation` - indicates that this block describes a Web service operation
- `method` - specifies the HTTP method used by the operation
- `address` - identifies the URI of the operation (or a URI template in case the URI contains input parameters)
- `input` - marks the input of the operation
- `output` - marks the output of the operation

The hRESTS model is very similar to the WSDL model e.g. both share *service*, *operation*, *input* and *output* elements. In [80] the authors state that “*hRESTS is roughly equivalent to WSDL*”.

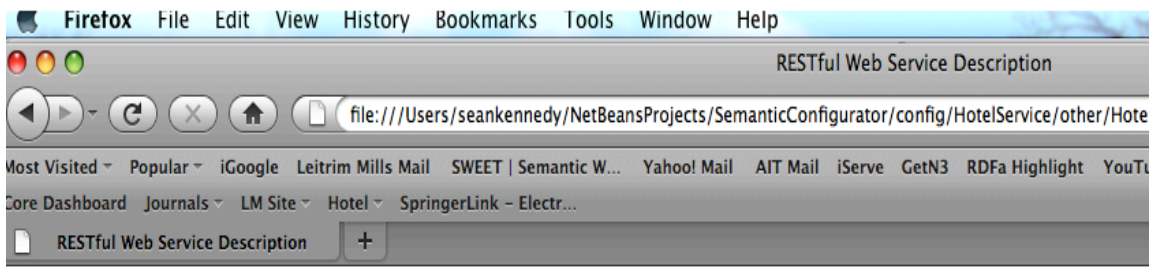
4.5.2.3 MicroWSMO

MicroWSMO [80] is an extension of hRESTS that refers to the semantic layer [85]. As explained earlier, SAWSDL is an extension to WSDL that specifies how to annotate (typically SOAP-based) service descriptions with semantic information. Given that the hRESTS service model is so similar to the WSDL model, it is not surprising that MicroWSMO is very similar to SAWSDL. “*Because the hRESTS view of services is so similar to that of WSDL, we can adopt SAWSDL properties to add semantic annotations*” [80]. In effect, MicroWSMO is to hRESTS, what SAWSDL is to WSDL. MicroWSMO defines three relations i.e. elements with the *rel* attribute set as follows:

- `model` – the *href* attribute points to an ontology concept or instance (corresponds to the `sawSDL:modelReference`).
- `lifting` and `lowering` – the *href* attribute points to the transformations to and from the ontology layer respectively (correspond to `sawSDL:liftingSchemaMapping` and `sawSDL:loweringSchemaMapping` respectively).

hRESTS and MicroWSMO tool support

With regard to this thesis, the architecture requires a semantically annotated representation of the RESTful API description i.e. an RDF representation. This will enable the automated matching of semantically equivalent SOAP requests with their RESTful counterparts. With regard to the Hotel WS example, Fig. 4-11 shows the RESTful WS description in HTML:



ACME Hotels service API

This service is a hotel reservation service.

Operation getHotelDetails is invoked using the method GET at <http://example.com/hotels/{id}>

Parameters: id - the identifier of the particular hotel

Output value: hotel details in an ex:hotelInformation document

Figure 4-11 RESTful Hotel WS description

Fig. 4-12 shows how this description is annotated with hRESTS and MicroWSMO using the SWEET editing tool. Part (a) is hRESTS annotation and part (b) is MicroWSMO annotation:

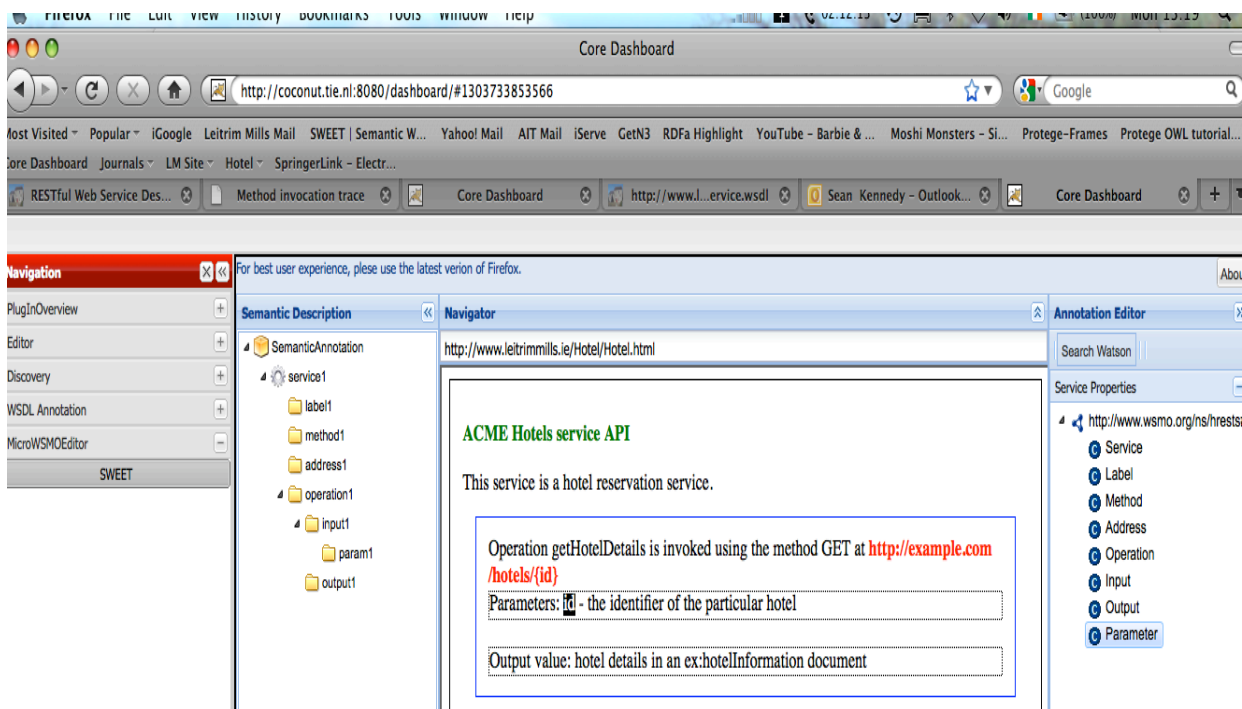


Figure 4-12(a) SWEET: hRESTS annotation

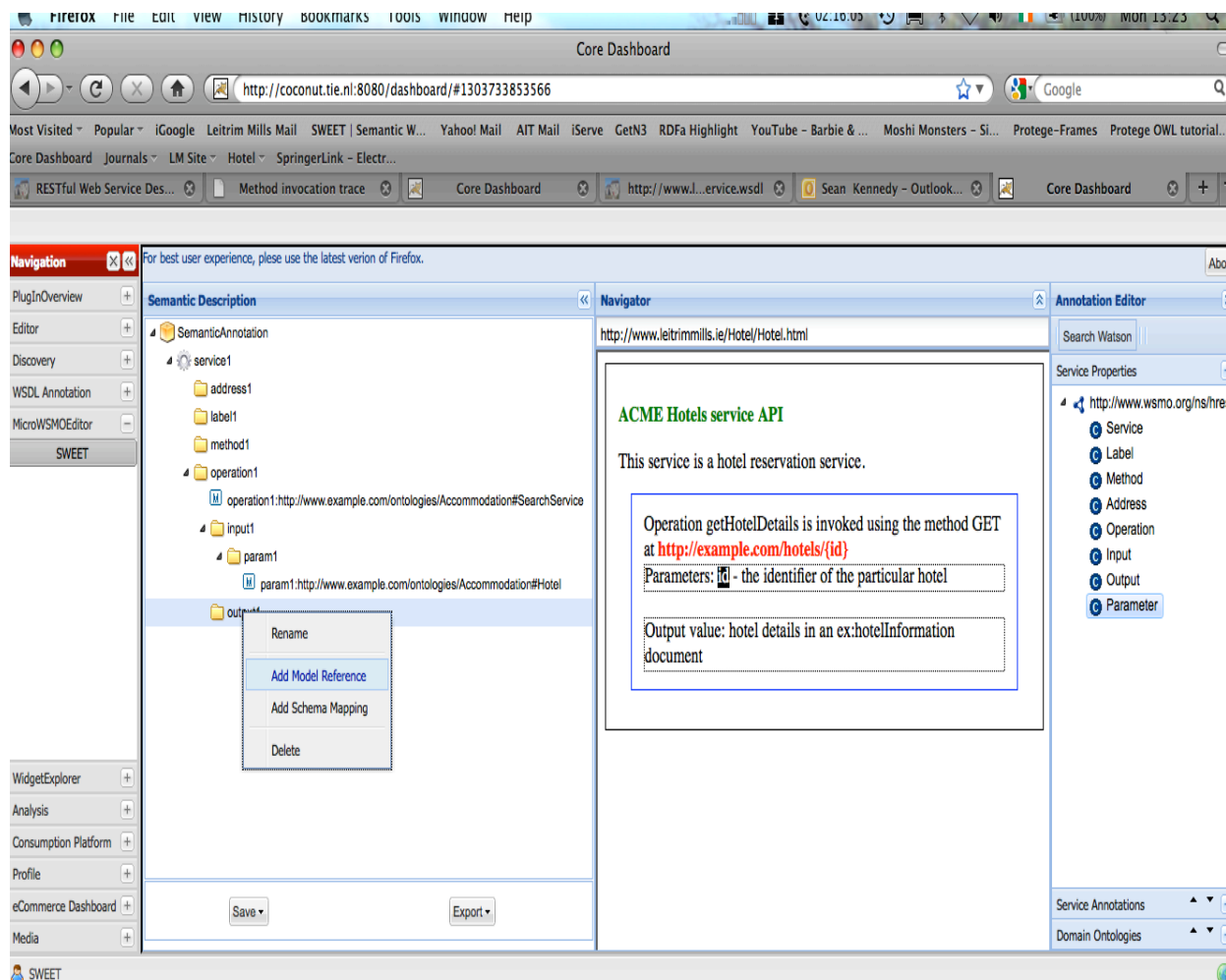


Figure 4-12(b) SWEET: MicroWSMO annotation

Listing 4-20 shows the HTML serialisation of the RESTful Hotel WS with both hRESTS (in red) and MicroWSMO (in brown) annotations.

```

<html>
<title>RESTful Web Service Description</title>

<div class="service" id="service1">
  <p>
    <span class="label" id="label1">ACME Hotels service API</span>
  </p>

  <p>This service is a hotel reservation service.<br>
  <div href="http://www.example.com/ontologies/Accommodation#SearchService" rel="model" class="operation" id="operation1">
    Operation getHotelDetails is invoked using the method <span class="method" id="method1">GET</span> at
    <span class="address" id="address1">http://example.com/hotels/{id}</span><br>
    <div class="input" id="input1">Parameters:
      <span href="http://www.example.com/ontologies/Accommodation#Hotel" rel="model" class="parameter" id="param1">
        id
      </span> - the identifier of the particular hotel
    </div><br>

    <div class="output" id="output1">Output value: hotel details in an ex:hotelInformation document
    </div>
  </div>
</p>
</div>
</html>

```

hRESTS
MicroWSMO

Listing 4-20 hRESTS and MicroWSMO annotated Hotel WS (HTML)

Note how hRESTS uses the *class* attribute and MicroWSMO uses the *rel* attribute. In Listing 4-20, MicroWSMO identifies two model link relations: the first model tells us that this service is an accommodation search service. The second model states that the input to the operation is an instance of the class Hotel in the data ontology. There are no liftings or lowerings as this example did not require them.

Listing 4-21 shows the RDF/XML serialisation of the RESTful Hotel WS with both hRESTS (in red) and MicroWSMO (in brown) annotations.

```

<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:msm="http://cms-wq.sti2.org/ns/minimal-service-model#" xmlns:hr="http://www.wsmo.org/ns/hrests#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" xmlns:sawSDL="http://www.w3.org/ns/sawSDL#"
  xmlns:wsl="http://www.wsmo.org/ns/wsmo-lite#" xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <msm:Service rdf:ID="HotelService">
    <rdfs:isDefinedBy rdf:resource=""/>
    <rdfs:label>ACME Hotels service API</rdfs:label>
    <msm:hasOperation>
      <msm:Operation rdf:ID="RetrievalOperation">
        <hr:hasMethod>GET</hr:hasMethod>
        <hr:hasAddress rdf:datatype="http://www.wsmo.org/ns/hrests#URITemplate">
          http://example.com/hotels/{id}
        </hr:hasAddress>
        <sawSDL:modelReference rdf:resource="http://www.example.com/ontologies/Accommodation#SearchService"/>
        <msm:hasInput>
          <msm:MessageContent rdf:ID="RequestInputParam">
            <sawSDL:modelReference rdf:resource="http://www.example.com/ontologies/Accommodation#Hotel"/>
          </msm:MessageContent>
        </msm:hasInput>
        <msm:hasOutput>
          <msm:MessageContent rdf:ID="output1"/>
        </msm:hasOutput>
      </msm:Operation>
    </msm:hasOperation>
  </msm:Service>
</rdf:RDF>

```

hRESTS
MicroWSMO

Listing 4-21 hRESTS and MicroWSMO annotated Hotel WS (RDF/XML)

Such is the similarity between SAWSDL and MicroWSMO that SWEET re-uses the *sawSDL:modelReference* annotation for MicroWSMO. SWEET also defines a Minimal Service Model (prefix *msm:*) which it shares across vocabularies other than hRESTS. Thus, the *Service*, *Operation*, *hasInput* and *hasOutput* elements are defined in the *msm:* namespace. Both, *hasAddress* and *hasMethod* are defined in the hRESTS namespace.

4.5.2.4 GRDDL

GRDDL is a W3C recommendation [90] that enables RDF to be extracted from XML documents, including XHTML. This is achieved in two steps:

- firstly, the document creator informs document consumers that this is a GRDDL source document by setting the `profile` attribute of the `head` element to `http://www.w3.org/2003/g/data-view`
- secondly, an XSLT transformation that will transform the source document into RDF is specified using the `rel` attribute of the `link` element

For example, Listing 4-22 is the `head` section of Listing 4-20 modified accordingly. The transformation is an openly available XSLT stylesheet that parses (X)HTML documents marked-up with hRESTS and MicroWSMO to produce RDF.

```
<head profile="http://www.w3.org/2003/g/data-view">
  <title>RESTful Web Service Description</title>
  <link rel="transformation" href="http://members.sti2.at/~jacekk/hrests/hrests.xslt" />
</head>
```

Listing 4-22 XHTML section annotated with GRDDL metadata

Note that a GRDDL aware parser is required for the RDF to be produced. One is available at <http://www.w3.org/2007/08/grddl/>. This parser was used to parse Listing 4-22, producing the RDF of Listing 4-23. Thus, annotating a RESTful WS description with GRDDL will also provide the RDF representation required by the architecture.

```

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://www.leitrimmills.ie/sk/HotelService/GRDDL/HotelAPI.html#service1>
  a <http://www.wsmo.org/ns/hrests#Service>;
  <http://www.w3.org/2000/01/rdf-schema#isDefinedBy> <http://www.leitrimmills.ie/sk/HotelService/GRDDL/HotelAPI.html>;
  <http://www.w3.org/2000/01/rdf-schema#label> "ACME Hotels service API";
  <http://www.wsmo.org/ns/hrests#hasOperation> <http://www.leitrimmills.ie/sk/HotelService/GRDDL/HotelAPI.html#operation1> .

<http://www.leitrimmills.ie/sk/HotelService/GRDDL/HotelAPI.html#operation1>
  a <http://www.wsmo.org/ns/hrests#Operation>;
  <http://www.wsmo.org/ns/hrests#hasMethod> <http://www.wsmo.org/ns/hrests#GET>;
  <http://www.wsmo.org/ns/hrests#hasAddress> "http://example.com/hotels/{id}"^^<http://www.wsmo.org/ns/hrests#URITemplate>;
  <http://www.wsmo.org/ns/hrests#hasInputMessage> <http://www.leitrimmills.ie/sk/HotelService/GRDDL/HotelAPI.html#input1>;
  <http://www.wsmo.org/ns/hrests#hasOutputMessage> <http://www.leitrimmills.ie/sk/HotelService/GRDDL/HotelAPI.html#output1> .

<http://www.leitrimmills.ie/sk/HotelService/GRDDL/HotelAPI.html#input1>
  a <http://www.wsmo.org/ns/hrests#Message> .

<http://www.leitrimmills.ie/sk/HotelService/GRDDL/HotelAPI.html#output1>
  a <http://www.wsmo.org/ns/hrests#Message> .

```

Listing 4-23 RDF produced from the GRDDL annotated Listing 4-20

4.5.2.5 WSMO-Lite

WSMO-Lite [91] is a lightweight, bottom-up approach for semantic annotation of Web services. Languages such as RDF, RDFS and OWL are used at this layer. WSMO-Lite is based on the more heavyweight, top-down semantics-first approach of WSMO [92]. WSMO-Lite captures four aspects of service semantics:

- Information model – the information model relates to the ontology and any lifting/lowering transformations to/from the semantic layer. The information model is useful for data mediation i.e. using a common ontology for transforming outputs from one Web service to the input format required by another Web service [109].

- Functional semantics – what the service does. Functional semantics are specified in SAWSDL on the `service` and/or `interface` elements and in MicroWSMO on the `service` and/or `operation` class. Functional semantics “*describe the service capability – that is, what the service can offer*” [109]. Functional semantics are useful in service discovery and composition and are specified by:
 - *Categorisation* – “*the service functionality falls within some category in an agreed classification scheme*” [109] e.g. RosettaNet Technical Directory. WSMO-Lite offers the RDFS class `FunctionalClassificationRoot` to setup classification hierarchies.
 - *Capability* – the functionality can also be defined using logical conditions that must hold before and after service invocation; these conditions are known as preconditions and effects [109]. The preconditions outline the conditions that must hold before the service can be executed and the effects identify the outputs that will hold after successfully executing the service. WSMO-Lite offers the RDFS classes `Capability` and `Effect` so that preconditions and effects can be setup in an ontology.
- Non-functional semantics – this is information that the consumer of the Web service may require even though the information is incidental to the successful execution of the Web service. A good example is the price of the Web service. As a result, non-functional semantics are often used for ranking and selection e.g. execute the cheapest service. WSMO-Lite provides the class `NonFunctionalParameter` to mark items with non-functional semantics.
- Behavioural semantics – similar to Functional semantics in that the behavioural semantics are defined using categories and/or preconditions and effects. However, rather than being guided by an explicit process, the client selects the operation (annotated with behavioural semantics) to invoke next. In SAWSDL, behavioural semantics are added to the `operation` element; in MicroWSMO, they are added to the `input` or `output` class. The client can now reason about which operation can be executed at a particulate point in time (at a particular state in the application) [93].

The WSMO-Lite RDFS ontology is listed in Listing 4-24:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf_schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22_rdf_syntax_ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix wsl: <http://www.wsmo.org/ns/wsmo_lite#> .

wsl:Ontology a rdfs:Class;
  rdfs:subClassOf owl:Ontology.
wsl:FunctionalClassificationRoot rdfs:subClassOf rdfs:Class.
wsl:NonFunctionalParameter a rdfs:Class.
wsl:Condition a rdfs:Class.
wsl:Effect a rdfs:Class.
```

Listing 4-24 WSMO-Lite Ontology (Notation 3)

Listing 4-25 demonstrates a specific WSMO-Lite example. It is based on a telecoms example from [91]. The example defines a telecommunications video on demand service. The classes and properties of the informational model (ontology) are initially setup. The pricing information (the non-functional data) is then outlined. Lastly, the functional semantics are detailed – in this case a categorisation hierarchy is used (as opposed to preconditions and effects).

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf_schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22_rdf_syntax_ns#> .
@prefix wsl: <http://www.wsmo.org/ns/wsmo_lite#> .
@prefix ex: <http://example.org/onto#> .
@prefix xs: <http://www.w3.org/2001/XMLSchema#> .

# This is a WSMO-Lite ontology
# <> is the blank URI, denoting the containing file

# information model
<> a wsl:Ontology.
ex:Customer a rdfs:Class .
ex:hasService a rdf:Property ;
  rdfs:domain ex:Customer ;
  rdfs:range ex:Service .
ex:Service a rdfs:Class .
ex:hasConnection a rdf:Property ;
  rdfs:domain ex:Customer ;
  rdfs:range ex:NetworkConnection .
ex:NetworkConnection a rdfs:Class .
ex:providesBandwidth a rdf:Property ;
  rdfs:domain ex:NetworkConnection ;
  rdfs:range xs:integer .
ex:VideoOnDemandService rdfs:subClassOf ex:Service .

# non-functional semantics
ex:PriceSpecification rdfs:subClassOf wsl:NonFunctionalParameter .
ex:VideoOnDemandPrice a ex:PriceSpecification ;
  ex:pricePerChange "30"^^ex:euroAmount ;
  ex:installationPrice "49"^^ex:euroAmount .

# functional semantics
ex:SubscriptionService a wsl:FunctionalClassificationRoot .
ex:VideoSubscriptionService rdfs:subClassOf ex:SubscriptionService .
ex:NewsSubscriptionService rdfs:subClassOf ex:SubscriptionService .
```

Listing 4-25 WSMO-Lite Example

Listing 4-26 details how SAWSDL would refer to the WSMO-Lite example of Listing 4-25. Part (a) refers to the Information model concept (and the lowering transformation stylesheet). Part (b) refers to the functional semantics by pointing to the classification category created by Listing 4-25. Lastly, part (c) refers to the non-functional data (the price) of the service.

```
<xs:element name="NetworkConnection" type="NetworkConnectionType"
  sawsdl:modelReference="http://example.org/onto#NetworkConnection"
  sawsdl:loweringSchemaMapping="http://example.org/NetworkConnection.xslt" />
```

Listing 4-26 (a) SAWSDL Information model references

```
<wsdl:interface name="NetworkSubscription"
  sawsdl:modelReference="http://example.org/onto#VideoSubscriptionService" >
```

Listing 4-26 (a) SAWSDL Functional semantics reference

```
<wsdl:service name="SomeCompanyService" interface="NetworkSubscription"
  sawsdl:modelReference="http://example.org/onto#VideoOnDemandPrice" >
  <wsdl:endpoint...
</wsdl:service>
```

Listing 4-26 (c) SAWSDL Non-functional semantics reference

4.5.2.6 SA-REST

An alternative approach to the hRESTS/MicroWSMO approach is SA-REST (Semantic Annotations for REST). SA-REST is a W3C submission since April, 2010. In the submission, the authors state that SA-REST “*can be used to annotate RESTful API descriptions*” [94].

SA-REST is a poshformat. Poshformats are a superset of Microformats and differ from microformats in that poshformats may not have gone through the rigorous community process that microformats go through. SA-REST “*define three basic properties that can be used to non-intrusively annotate HTML/XHTML documents, typically to embed ontological meta-data*” [94]. SA-REST uses the (X)HTML `class` and `title` attributes to specify the property and its associated value.

The three properties SA-REST define are [94]:

- `domain-rel` - identifies the domain description of a complete resource. From an SA-REST perspective, `domain-rel` is considered a block level property i.e. other SA-REST

properties are allowed within the (X)HTML element that is annotated with the `domain-rel` property. Listing 4-27 is an example (X)HTML fragment using `domain-rel`.

```
<span class="domain-rel" title="http://example.org/ontologies/library">
  Welcome to the library...
</span>
```

Listing 4-27 *domain-rel* property example

- `sem-rel` – captures the semantics of a link and is only used on the `<a>` tag. From an SA-REST perspective, `sem-rel` is considered an element level property i.e. it is an SA-REST property that should not contain other SA-REST properties. `sem-rel` is often used for referring to external annotations of third-party documents e.g. schemas. Listing 4-28 is an example (X)HTML fragment using `sem-rel`.

```
<a href="http://example.org/schemas/inputOrder.xsd"
  class="sem-rel" title="http://example.org/ontologies/library/schema#book">
  This is the schema we used when ordering a book electronically.
</a>
```

Listing 4-28 *sem-rel* property example

- `sem-class` – an element level property that identifies the semantics of a single entity within a resource. Listing 4-29 is an example (X)HTML fragment using `sem-class`.

```
The most popular
<span class="sem-class" title="http://example.org/ontologies/library#book">book</span>
in the Fiction section is...
```

Listing 4-29 *sem-class* property example

Of relevance to this thesis is SA-RESTS's ability to annotate RESTful API descriptions with a service description (supporting tool automation, similar to hRESTS) and semantic metadata

(enabling advanced tool support, similar to MicroWSMO). SA-REST defines a service model which is based on the hRESTS service model [94] i.e. the elements in the SA-REST namespace resemble those in the hRESTS namespace e.g. *service* and *operation*. SA-REST refers to semantic concepts via (the URI values of) the *title* attribute on the *domain-rel*, *sem-rel* and *sem-class* properties. Thus, in effect, SA-REST is an alternative to the hRESTS/MicroWSMO offering.

However, there are no specific tools available for SA-REST annotation. In addition, to serialise your SA-REST-annotated (X)HTML page to RDF, one requires RDF-in-attributes (RDFa) [95]. RDFa is covered next.

4.5.2.7 RDFa

RDFa, a W3C Recommendation, is a way to express RDF within XHTML [96]. RDFa shares some use cases with microformats. Microformats specify both a syntax for embedding structured data into (X)HTML documents (e.g. *class* and *rel* attributes in hRESTS and MicroWSMO respectively) and a vocabulary of specific terms for each microformat (e.g. *service*, *operation* etc.. in hRESTS and *model*, *lifting* and *lowering* in MicroWSMO). RDFa on the other hand, specifies only a syntax i.e. what XHTML attributes to use and relies on vocabularies of terms independently specified by others. RDFa allows terms from multiple independently-developed vocabularies to be freely intermixed and can be parsed without knowledge of the specific vocabulary being used.

RDFa uses certain existing XHTML attributes (Table 4-2) and also defines some extra attributes (Table 4-3) [97]:

Existing XHTML attributes	Meaning	Subject/Predicate/Object
rel	expresses relationships between two resources	Predicate
rev	expresses reverse relationships between two resources	Predicate
content	a string (literal object in RDF)	Object
href	the resource object (URI)	Object
src	the resource object (URI) when the resource is embedded	Object

Table 4-2 Existing XHTML attributes used by RDFa

New RDFa attributes	Meaning	Subject/Predicate/Object
about	states what the data is about (the subject)	Subject
property	expresses a relationship between a subject and some literal text	Predicate
resource	the object resource when the URI is not intended to be clickable	Object
datatype	expresses the datatype of a literal	n/a
typeof	specifies the RDF type of the subject	n/a

Table 4-3 New XHTML RDFa attributes

In RDFa, a subject is generally indicated using *about*, and predicates are represented using one of *property*, *rel*, or *rev*. Objects which are URI references are represented using *href*, *resource* or *src*, while objects that are literals are represented either with *content* or the content of the element in question (with an optional datatype expressed using *datatype*) [97].

Listing 4-30 is XHTML (Listing 4-20) annotated with SA-REST/RDFa.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN" "http://www.w3.org/MarkUp/DTD/xhtml-rdfa-1.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>RDFa example using SAREST</title></head>
<body>
  <div typeof="sarest:Service"
    about="ex:HotelService"
    xmlns:ex="http://www.example.org/#"
    xmlns:sarest="http://www.knoesis.org/research/srl/standards/sa-rest/#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <h1>
      <span property="rdfs:label">ACME Hotels</span> service API
    </h1>
    <div rel="sarest:hasOperation">
      <span typeof="sarest:Operation" about="ex:retrieval">
        <strong>Operation</strong> <code property="rdfs:label">getHotelDetails</code><br>
        Invoked using the <span property="sarest:hasMethod">GET</span> at
        <code property="sarest:hasAddress" datatype="sarest:URITemplate" >http://example.com/h/{id}</code><br>
        <span rel="sarest:hasInputMessage">
          <span typeof="sarest:Message">
            <strong>Parameters:</strong> <code>id</code> the identifier of the particular hotel
          </span>
        </span><br>
        <span rel="sarest:hasOutputMessage">
          <span typeof="sarest:Message">
            <strong>Output value:</strong> hotel details in an <code>ex:hotelInformation</code> document
          </span>
        </span>
      </span>
    </div>
  </div>
</body>
</html>
```

Listing 4-30 XHTML annotated with RDFa

Note that in Listing 4-30 that it is the SA-REST namespace that is used as opposed to the XHTML attributes in the SA-REST specification. This is the flexibility of RDFa – one can refer to independently created vocabularies. Listing 4-31 is the RDF, serialised as N3, extracted from Listing 4-30 using the W3C's RDFa Distiller tool at <http://www.w3.org/2007/08/pyRdfa/>.

```
@prefix ex: <http://www.example.org/#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sarest: <http://www.knoesis.org/research/srl/standards/sa-rest/#> .
@prefix xhv: <http://www.w3.org/1999/xhtml/vocab#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<ex:HotelService> a sarest:Service ;
  rdfs:label "ACME Hotels" ;
  sarest:hasOperation <ex:retrieval> .

<ex:retrieval> a sarest:Operation ;
  rdfs:label "getHotelDetails" ;
  sarest:hasAddress "http://example.com/h/{id}"^^sarest:URITemplate ;
  sarest:hasInputMessage
    [ a sarest:Message
    ] ;
  sarest:hasMethod "GET" ;
  sarest:hasOutputMessage
    [ a sarest:Message
    ] .
```

Listing 4-31 **RDF extracted from Listing 4-30 (N3 notation)**

4.6 Summary

In this chapter, the technologies underpinning the Semantic Web were detailed. The Semantic Web's data model (RDF), its abstract graph model and its various serialisation formats were discussed. The technologies that actually add the semantics, namely RDFS and ontologies were then covered. Lastly, the Semantic Web Services stacks relevant to this thesis were explained. The technologies in the stacks will be central to the automated solution provided by the architecture. The architecture is the subject of the next section.

Chapter 5

StoRHm

5.1 Introduction

In the previous chapters, the background technologies relevant to this thesis, namely XML Web Services, the REST architectural style and the Semantic Web were outlined. In this chapter, the architecture of StoRHm is presented. Initially, an overview is presented; the purpose of which is to link the research objectives presented in the Introduction chapter with the architecture of StoRHm. Following from that, the architectural evolution of StoRHm is presented in which StoRHm's evolution into two distinct architectures is outlined. Lastly, the Requirements, Architecture and Design of both StoRHm v1 and v2 are explained in detail.

5.2 Architecture overview

The research objectives stated in the Introduction chapter are:

- 1) *Can XML WS 'POST tunnelling' be addressed?*

2) *Is it possible for an enterprise to gradually migrate from XML WS to RESTful WS?*

3) *Can the architecture ensure that:*

- a) *Messages are immediately visible?*
- b) *There is minimal impact on the client?*
- c) *Semantic Web intelligence is available of to automate the mapping?*

These objectives lead to the architecture shown in Fig. 5-1. Fig. 5-1 contrasts abstractly the architectures of a normal WS invocation (Fig. 5-1 (a)) with a WS invocation with StoRHm in place (Fig. 5-1 (b)). The StoRHm elements are highlighted in blue.

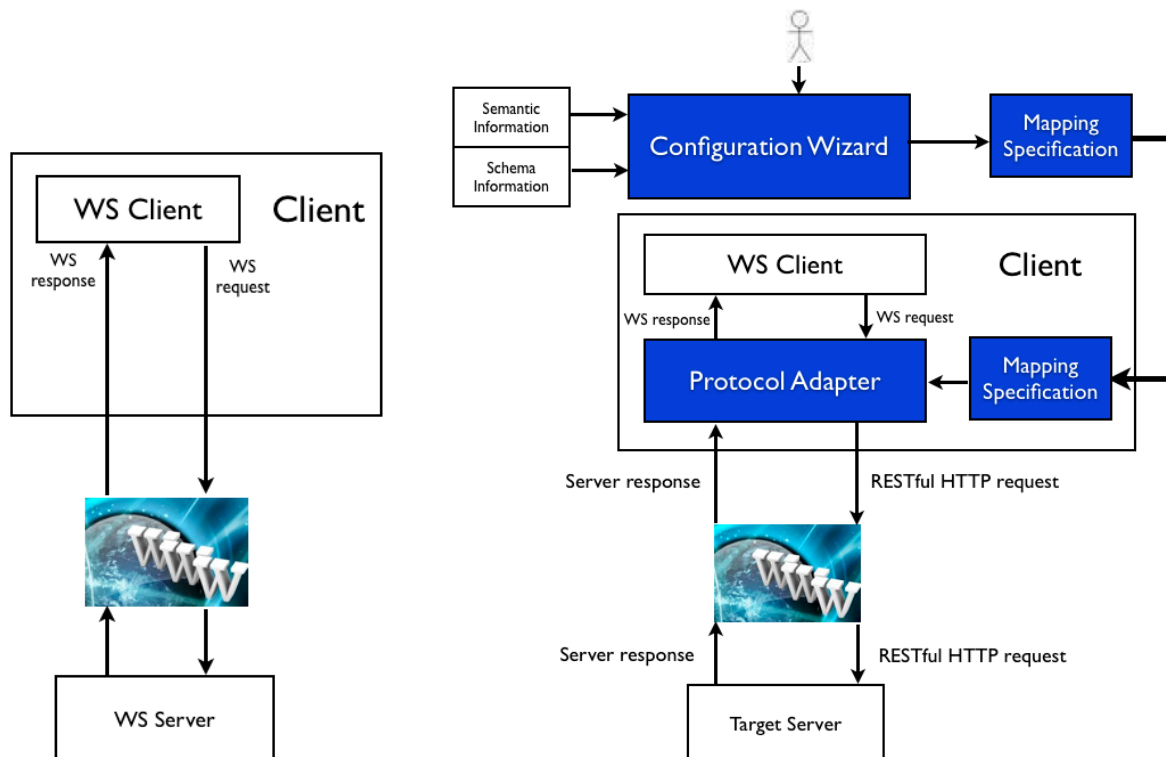


Figure 5-1 (a) XML WS Architecture

Figure 5-1 (b) StoRHm Architecture

Addressing the research objectives in turn:

1. POST Tunnelling - As can be seen from the architectural diagram above, the adapter is located on the client side. The adapter translates XML WS requests to corresponding RESTful HTTP format and hence satisfies objective (1). The alternative was of course to locate the adapter on the server side, as in [8]. However, this would mean that the messages on the network would still be SOAP/POX based; ‘POST tunnelling’ would still exist and thus objective (1) would not be satisfied. For failing this objective (and the objectives outlined in 3a, 3b and 3c – see later), the server side approach was rejected.

2. Gradual migration - Clients can be migrated gradually from SOAP/POX WS to RESTful WS as the client-side adapter ensures that the clients interface is unaffected by a back-end server migration. The clients can therefore be migrated as and when it suits the enterprise [7]. Thus, objective (2) is satisfied.
- 3.(a) As StoRHm is client-based, the messages are immediately visible and thus objective (3a) is satisfied. As the server side approach in [8] only transforms the messages when they reach the server, it fails this objective.
- 3.(b) StoRHm takes into account the interface-specific nature of the client XML WS invocations and thus impact on the client is minimal, thereby satisfying this objective. The framework outlined in [8] requires client code compilation, failing the objective.
3. (c) As Semantic Web technologies are used to automate the mapping stage of the architecture, this objective is satisfied. The framework outlined in [8] does not use Semantic Web technologies and thus fails this objective.

5.3 Project lifecycle

There are two versions of the StoRHm architecture and both versions consist of a static configuration element followed by a dynamic runtime element. Figure 5-2 charts the timelines of the two versions of StoRHm:

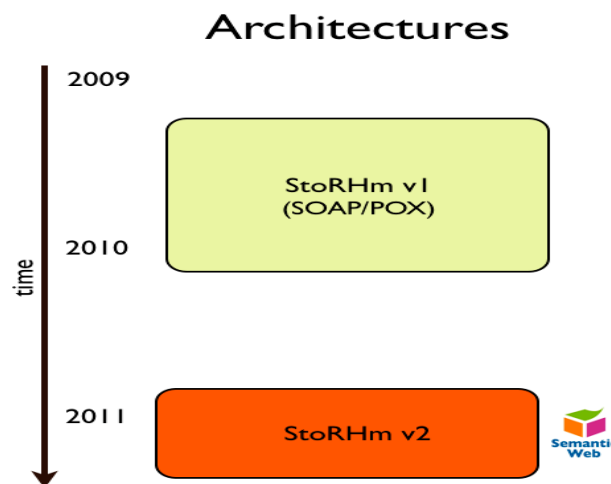


Figure 5-2 Timeline of Architectures

StoRHm v1 (version 1)

StoRHm maps XML Web Service messages (both SOAP and POX) to RESTful HTTP format [138][141][142][143][144][145]. In StoRHm v1, the configuration element is manual. The function of the configuration element is to create a mapping specification that enables the runtime protocol

adapter to transform the messages.

StoRHm v2 (version 2)

The StoRHm architecture has evolved naturally into two distinct versions, namely v1 and v2. The major issue with the architecture used by StoRHm v1 is that the configuration element is heavily reliant on the user. This results in an architecture that requires significant user input. StoRHm v2 addresses this issue by integrating Semantic Web technologies to add intelligence to the configuration element. Thus, what was previously a user-intensive process is now an automated, semantically informed architecture. The critical difference is that StoRHm v2 leverages the Semantic Web to automate the configuration element [139][140]. In addition, v2 integrates the SOAP and POX solutions into a “one-stop-shop”.

The relationship between StoRHm v1 and v2 is shown in Fig. 5-3:

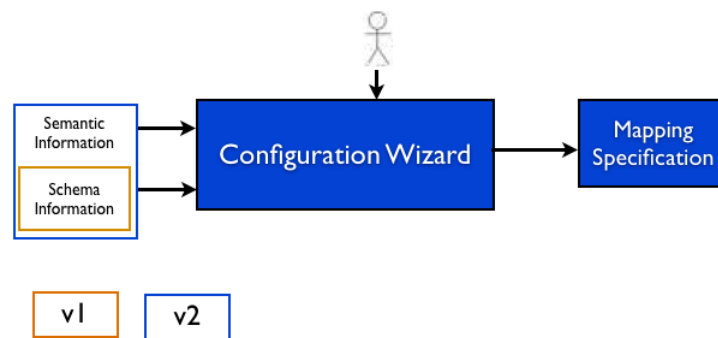


Figure 5-3 Relationship of the two versions

5.4 StoRHm v1

In this section, the Requirements, Architecture and Design of StoRHm v1 is explained in detail.

5.4.1 StoRHm v1 requirements

The requirements of StoRHm are the subject of this section. The requirements outlined at this point are generic for all versions of StoRHm. Fig. 5-4 represents the use case diagram for StoRHm.

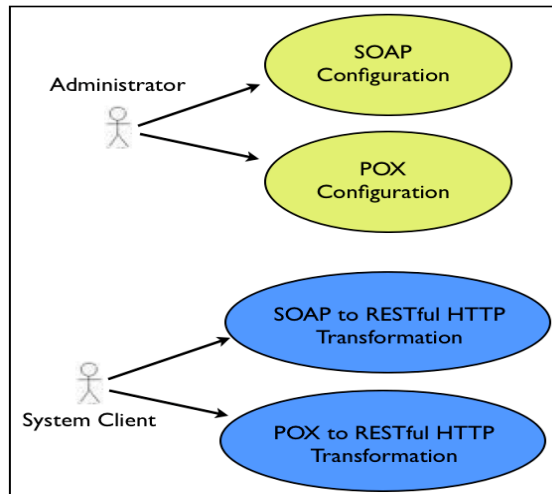


Figure 5-4 StoRHm Use Case Diagram

Web Service interoperability

The overriding goal of StoRHm is to enable existing XML Web Service clients to interact with a RESTful Web Service that is semantically equivalent to the original XML Web Service (Fig. 5-5). In Fig. 5-5, one can see a diagram representing the transformation from a specific interface (i.e. SOAP/POX) to a uniform interface (i.e. RESTful). It is based on an example from [26]. At the top of Fig. 5-5, *PhoneDirectoryService* represents the XML Web service with associated operations. The lower part of Fig. 5-5 is a representation of its RESTful equivalent. The arrow represents the mapping that will be carried out by the architecture.

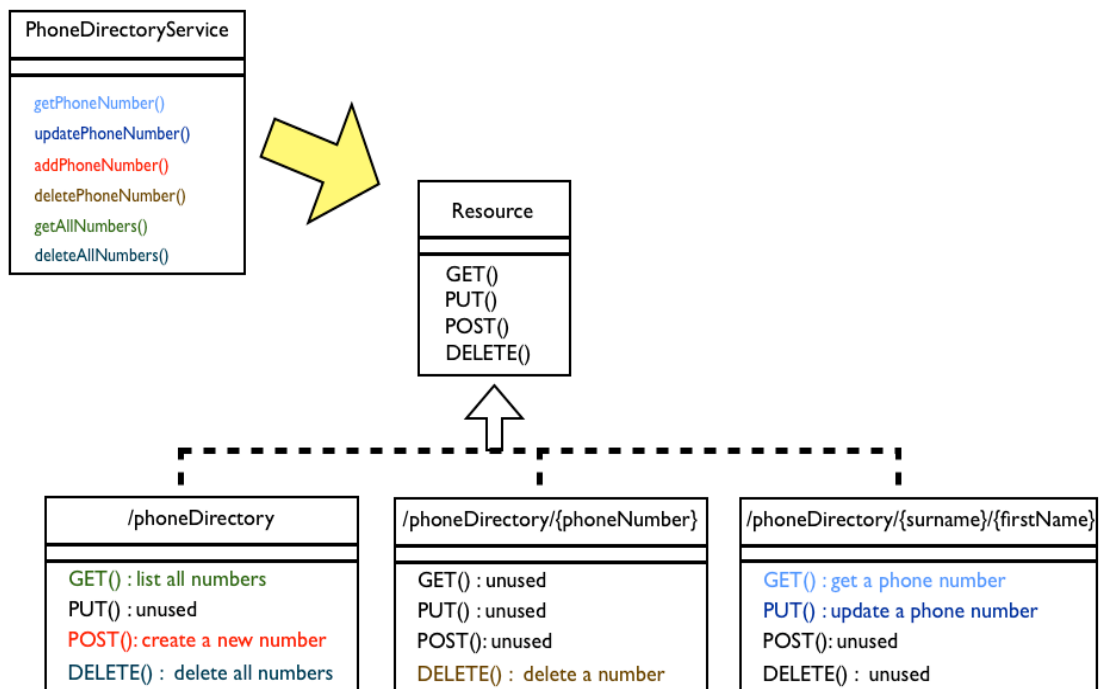


Figure 5-5 XML Web Service to RESTful Interface Mapping [26]

Based on Fig. 5-5, the mappings are shown in Table 5-1:

Specific Interface	Uniform Interface	
	Verb	URI
<code>getPhoneNumber</code>	GET	<code>/phoneDirectory/{surname}/{firstName}</code>
<code>updatePhoneNumber</code>	PUT	<code>/phoneDirectory/{surname}/{firstName}</code>
<code>addPhoneNumber</code>	POST	<code>/phoneDirectory</code>
<code>deletePhoneNumber</code>	DELETE	<code>/phoneDirectory/{phoneNumber}</code>
<code>getAllNumbers</code>	GET	<code>/phoneDirectory</code>
<code>deleteAllNumbers</code>	DELETE	<code>/phoneDirectory</code>

Table 5-1 **Specific Interface mapped to Uniform Interface**

As can be seen from Table 5-1, the operation *getPhoneNumber* is mapped to HTTP GET on the URI */phoneDirectory{surname}/{firstName}*. The parameters are in the URI i.e. *{surname}* and *{firstName}*. The *updatePhoneNumber* mapping follows a similar pattern. However, as per proper HTTP semantics, PUT is the verb used when updating. The *addPhoneNumber* operation, maps to POST on a collection resource */phoneDirectory*. In these situations, the server will expect its data in the entity body of the message and the server will decide upon the URI to be created based on some server-side algorithm. The *deletePhoneNumber* and *deleteAllNumbers* operations map logically to HTTP DELETE. However, depending on whether the URI represents a collection resource */phoneDirectory* or an individual resource */phoneDirectory/{phoneNumber}* determines whether the server will delete all the phone numbers or just one. A DELETE message to the collection resource */phoneDirectory* deletes all the phone numbers i.e. it is the semantic equivalent of the *deleteAllNumbers* operation. A DELETE message to the individual resource */phoneDirectory/{phoneNumber}* results in just one phone number being deleted; the phone number passed in the URI parameter *{phoneNumber}*. This is the equivalent of the *deletePhoneNumber* operation. The GET messages operate in the same fashion. The GET on the collection resource */phoneDirectory* is the equivalent of *getAllNumbers*.

StoRHm must facilitate these mappings. There are two elements to be considered here: the user's configuration of the mapping and the subsequent translation of XML messages as defined by that mapping. StoRHm should provide tool support for the former and should fully automate the latter.

Migration Enabler

StoRHm must enable the seamless migration of clients from XML Web Services to RESTful HTTP Web Services. This requirement means that the server can be migrated without breaking the existing client base.

Enable the Web infrastructure

XML Web Services produce opaque messages disabling advanced Web infrastructure features such as caching. Fielding highlights the importance of HTTP GET and its role in caching: “*The most frequent form of request semantics is retrieving a representation of a resource (e.g. the ‘GET’ method in HTTP), which can often be cached for later reuse*” [112]. Thus, the messages that StoRHm produces must be visible to Web intermediaries, thereby enabling the Web infrastructure.

Minimal client impact

The impact on the client must be minimal. Thus, no re-compilation of the client should be necessary for the client to avail of StoRHm.

5.4.2 StoRHm v1 architecture

Figure 5-6 outlines the architecture of a Web Service application that makes use of this architecture. The StoRHm elements are highlighted with a coloured background. The architecture consists of two elements: a configuration wizard and a runtime protocol adapter. The function of the configuration wizard is to create a mapping file that enables the runtime protocol adapter to transform the messages.

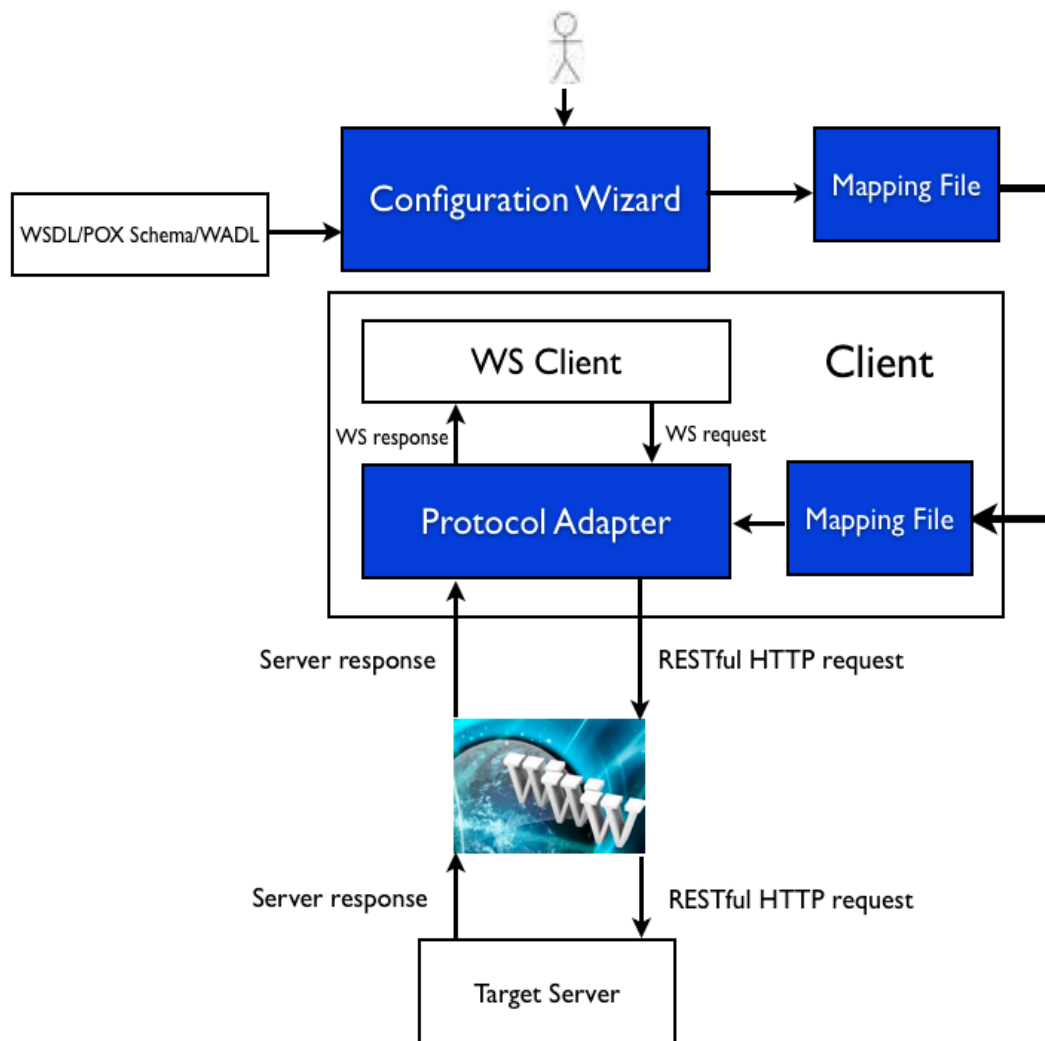


Figure 5-6 StoRHm v1 Architecture

5.4.3 StoRHm v1 Design

In this section, the design of StoRHm v1 is covered in detail. StoRHm v1 is capable of mapping both SOAP and POX Web Service requests to RESTful HTTP format. Despite being architecturally similar, their designs are different. As a result, separate sections are required. To this end, subsections 5.4.3.1 and 5.4.3.2 refer to POX and subsections 5.4.3.3 and 5.4.3.4 refer to SOAP. The principal differences result from the fact that SOAP-based Web services use WSDL to describe themselves whereas POX services use customised XML Schemas. In addition, SOAP messages returning to the client must have a SOAP envelope present; this is not required by POX clients.

5.4.3.1 StoRHm v1 (POX) Configuration Wizard

The function of the Configuration Wizard is to setup the Mapping file which the runtime Adapter later uses when transforming the messages.

GUI Design

The wizard is an event-driven Java GUI based application i.e. it *extends* *JFrame* and *implements* *ActionListener*. Fig. 5-7 outlines the StoRHm v1 (POX) wizard GUI interface. The GUI interface is broken up into three sections; a top, a middle and a bottom section. Each section corresponds to at least one Java JPanel. The top section (in yellow), details the input required from the user regarding the Schema, the optional WADL filename and the Root URI to be used on all the RESTful URI's. The middle section (in brown) outlines the Web Services and XPath expressions parsed from the Schema. The middle panel also outlines the RESTful URI that will be appended to the Root URI; the QoS checkboxes, the HTTP verb and MIME type drop down listboxes. Lastly, the middle panel also contains the Insert buttons that enable the user to insert the XPath expressions into the RESTful URI. The final section is the bottom section, which contains the OK and Cancel buttons.

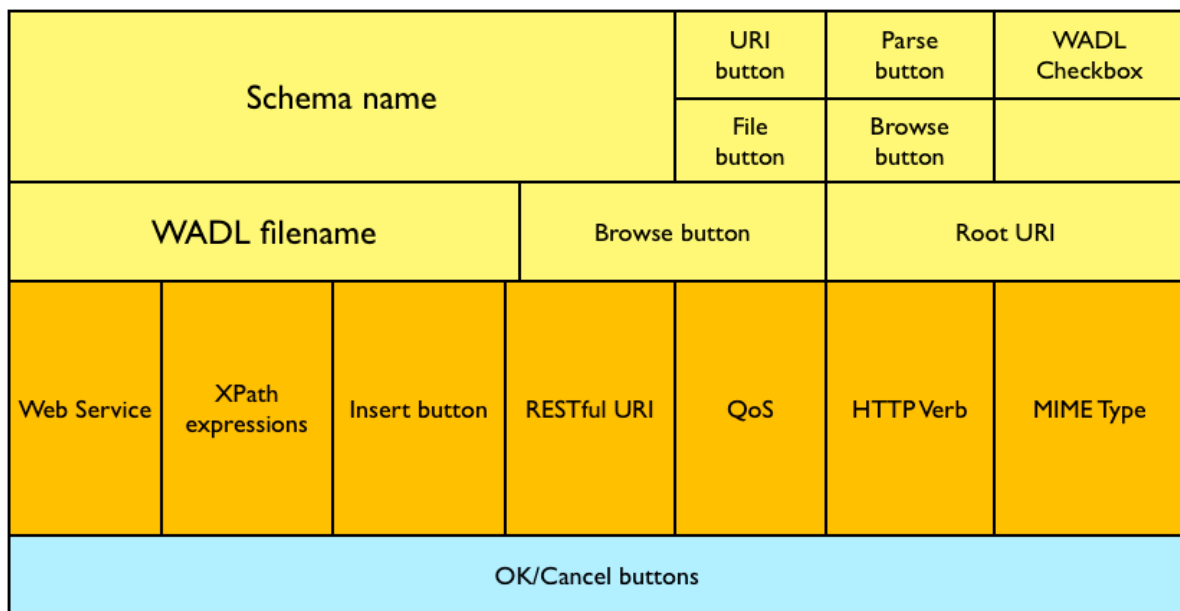


Figure 5-7 StoRHm v1 (POX) Configuration Wizard GUI Layout

Layout Managers

Fig. 5-8 details the layout managers used to organise the GUI interface. The top section (in yellow), is controlled overall by a 2-row, 1-column GridLayout manager. A FlowLayout manager controls each row within the top section. The top Schema row contains two sub-panels: one has a GridLayout of 2 rows, one column (the URI/File radio buttons); the other panel has 2 rows and 2 columns (the Parse and Browse buttons and the WADL checkbox). The second FlowLayout row is the WADL and Root URI section. The middle section, in brown, is managed by a GridLayout with 1 row

consisting of 7 columns. This row matches the brown row in Fig. 5-7. Lastly, the final section on the container is a FlowLayout for the OK and Cancel buttons.

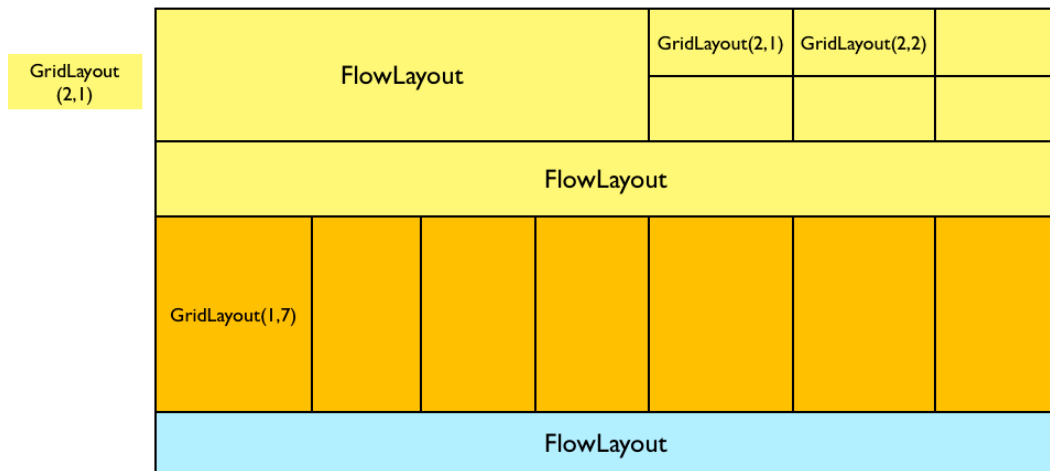


Figure 5-8 StoRHm v1 (POX) GUI Layout Managers

Example

Fig. 5-9 shows the initial screen of the wizard.

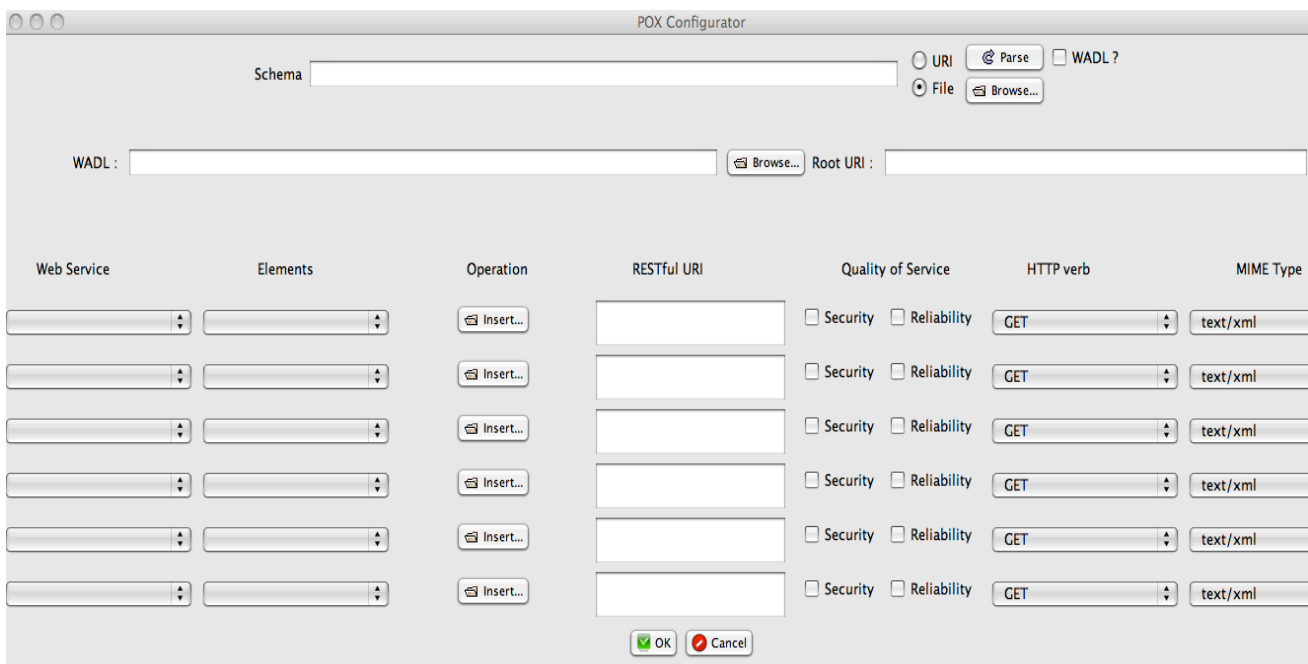


Figure 5-9 Sample StoRHm v1 (POX) Configuration Wizard

Sequence Diagrams

The design of the algorithm used is shown in the sequence diagrams of Fig. 5-10. Fig. 5-10 (a) represents the situation where the user selected the WADL checkbox whereas in Fig. 5-10 (b) the WADL checkbox is not chosen.

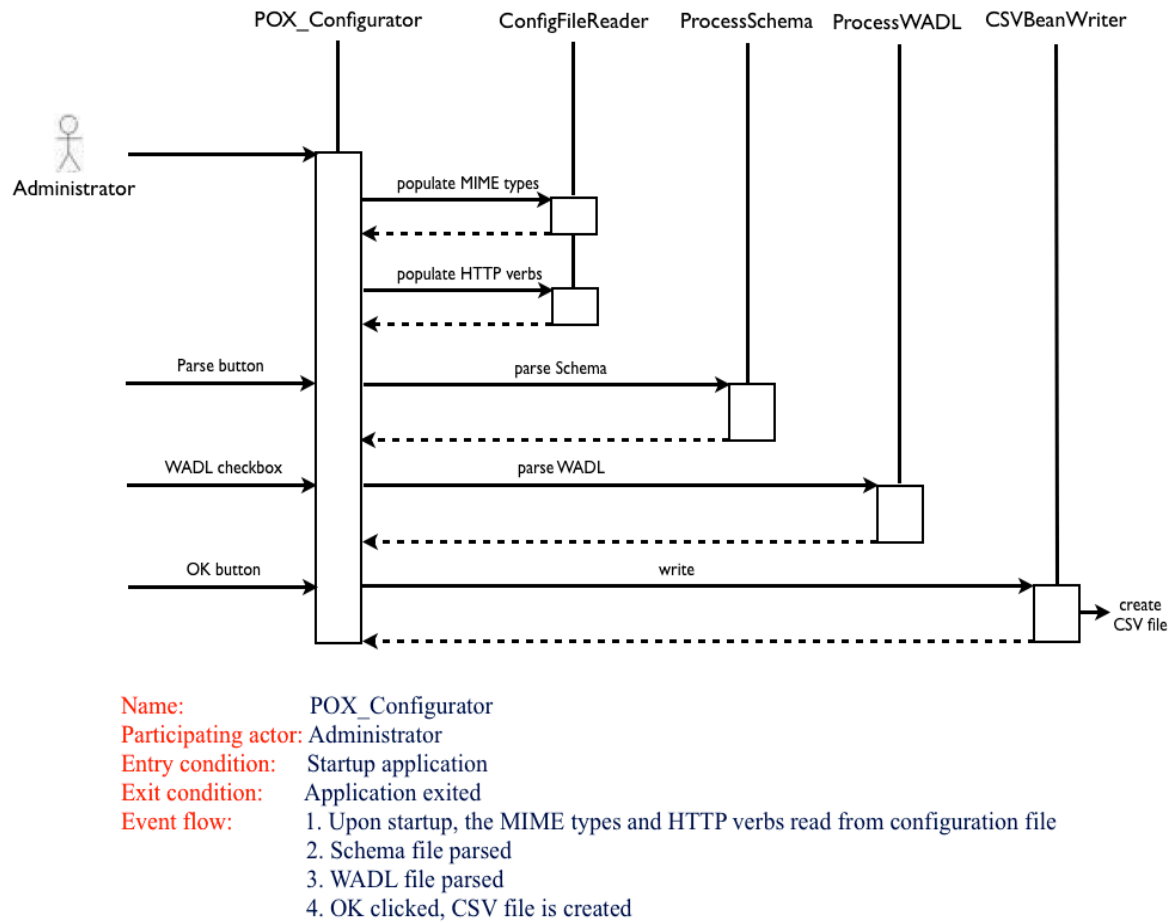


Figure 5-10 (a) StoRHm v1 (POX) Configuration Wizard sequence diagram (with WADL)

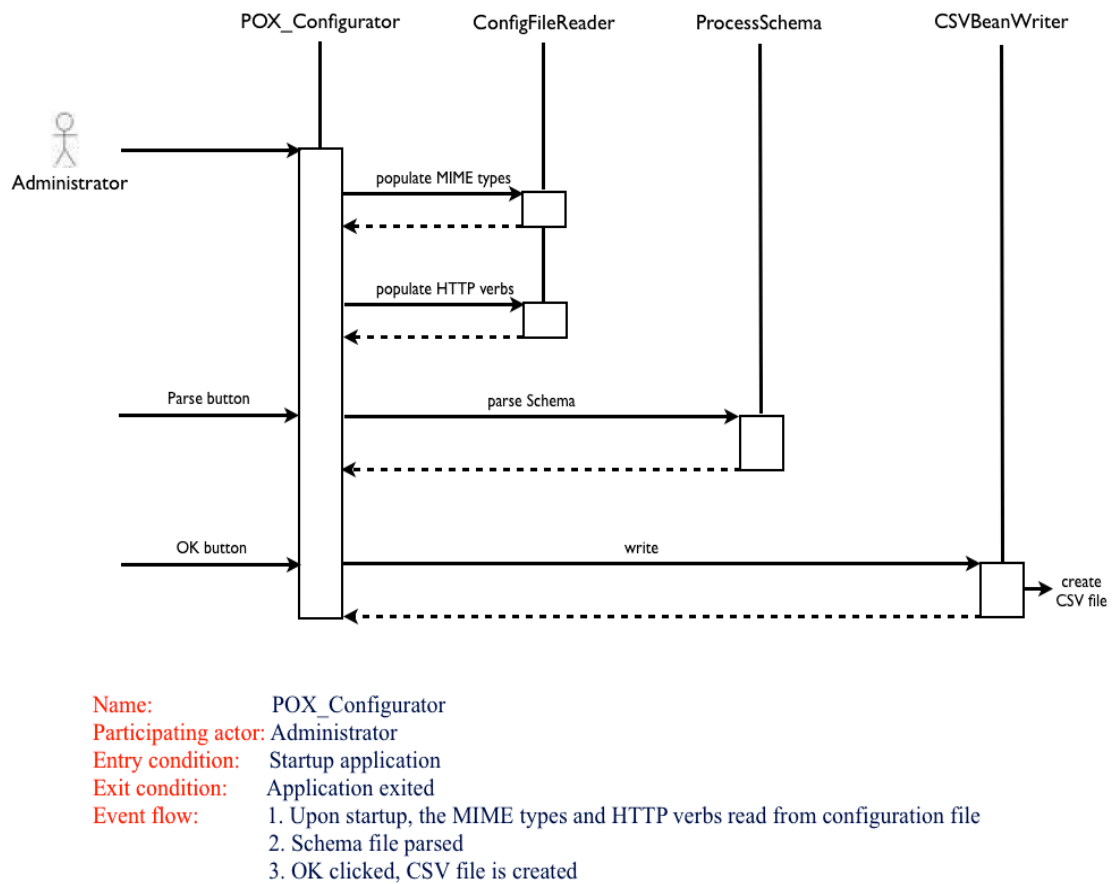


Figure 5-10 (b) StoRHm v1 (POX) Configuration Wizard sequence diagram (without WADL)

Algorithm

The configuration wizard is designed to take the following input: a POX schema and optionally a WADL (Web Application Description Language) file [84]. The user can access these files using both the HTTP and FILE protocols. When the user presses the “Parse” button, the Schema file is parsed. As the Schema is enterprise specific (and not a standard such as SOAP), the wizard is informed via a configuration file as to the XPath expressions to use when parsing the Schema. In addition, the configuration file informs the wizard of the XPath expressions to populate into the “Elements” drop-down listbox. This information is instance document-specific and saves the user time when entering data. Thus, when the Schema is parsed, the Web Services encountered are populated into the “Web Services” listboxes and the XPath expressions required by the runtime adapter are populated into the “Elements” area (see Fig. 5-9). If the user has not selected the WADL checkbox, then the “HTTP Verbs” and “MIME Types” (Fig. 5-9) listboxes are populated from the configuration file also.

If the user has selected the WADL checkbox, then the WADL file is parsed and the “RESTful URF”, “HTTP Verbs” and “MIME Types” sections are populated based on the WADL file. For example,

each specific resource (URI) will support certain verbs and certain media types. By populating these areas based on the WADL file this means that it is impossible to send an incorrect verb or MIME type to that URI.

The user then matches the Web Service with the required XPath expression(s) and, using the “*Insert*” buttons, builds the RESTful URIs. If WADL is being used then the Insert button will append to the RESTful URIs. Note that there is no need to use an XPath expression element for certain operations e.g. if one is issuing a POST to the Root URI, a “/” in the RESTful URI column suffices. The user selects the QoS checkbox buttons to indicate the QoS required. Lastly, the user matches the HTTP verbs and response type with the Web Service (if WADL is selected then this will already be done). The response type is “text/xml” by default. At this point the user presses OK and the mapping file is created.

Package/Class Diagrams

The package and class diagrams representing this design are shown in Fig. 5-11 and Fig. 5-12.

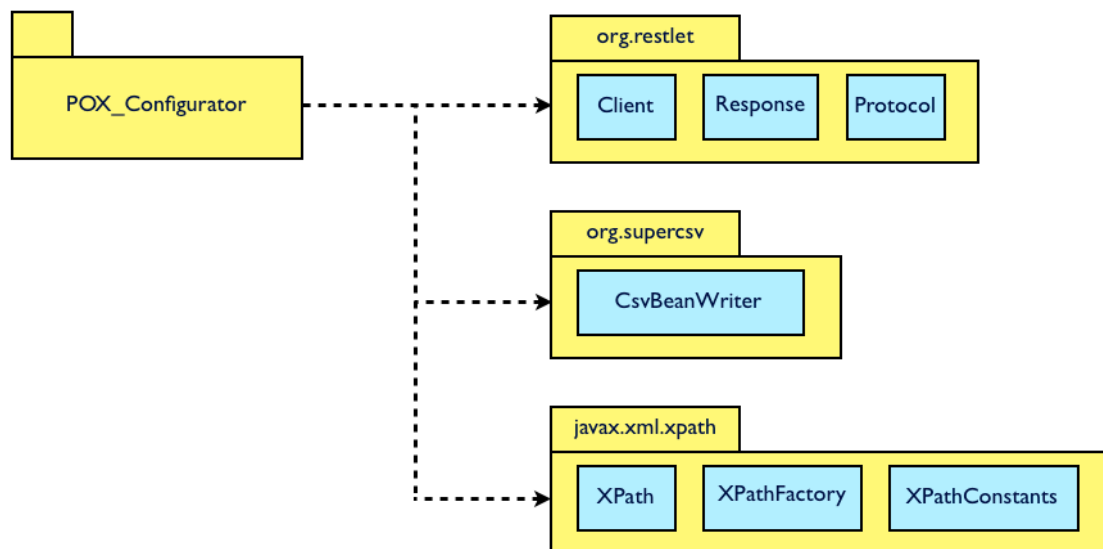


Figure 5-11 StoRHm v1 (POX) Configuration Wizard package diagram

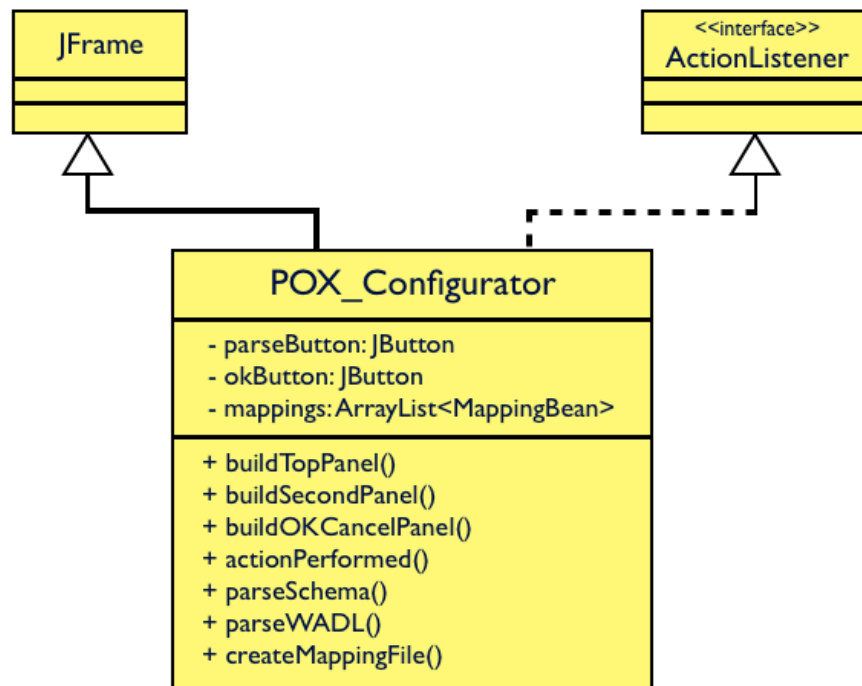


Figure 5-12 StoRHm v1 (POX) Configuration Wizard class Diagram

Design decisions

The Mapping file is implemented as a Comma Separated Value (CSV) file and is created by the configuration wizard (initially a database was used but the CSV file was more efficient in tests and as a result it was decided to use the CSV file). Fig. 5-13 shows a sample CSV file. The columns correspond to:

- Web Service name
- SOAP operation – this is not applicable (N/A) for POX configurations
- HTTP verb to use
- MIME Type
- URI to use (containing the XML XPath expressions to use inserted in curly braces)
- QoS features to apply

Fig. 5-13 is a sample MappingFile created from a POX Web Service. Note that the “*SOAP Operation*” column is not used (“N/A”). This indicates that this column is not applicable to a POX service.

Web Service Name RESTful URI	SOAP Operation	HTTP Verb	MIME type QoS
ServiceViewAll, http://.../BankServices/	N/A,	GET, No Security, Reliability	text/xml
ServiceDeleteAll, http://.../BankServices/	N/A,	DELETE, No Security, No Reliability	text/xml
ServiceAdd, http://.../BankServices/	N/A,	POST, No Security, No Reliability	text/xml
ServiceView, http://.../BankServices/{/.../BRANCH_CODE}/{/.../ACCOUNT_NUMBER},	N/A,	GET, No Security, No Reliability	text/xml
ServiceDelete, http://.../BankServices/{/.../BRANCH_CODE}/{/.../ACCOUNT_NUMBER},	N/A,	DELETE, No Security, No Reliability	text/xml
ServiceUpdate, http://.../BankServices/{/.../BRANCH_CODE}/{/.../ACCOUNT_NUMBER},	N/A,	PUT, Security, No Reliability	text/xml

Figure 5-13 StoRHm v1 (POX) CSV file

5.4.3.2 StoRHm v1 (POX) Protocol Adapter

The POX to RESTful HTTP runtime protocol adapter design is illustrated in Fig. 5-14.

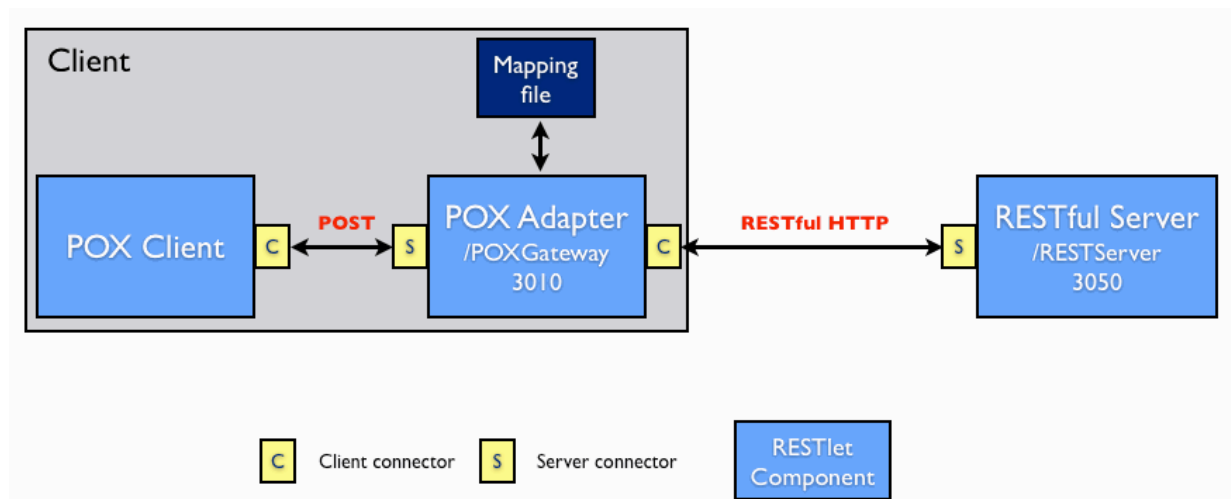


Figure 5-14 StoRHm v1 (POX) Protocol Adapter design

Adapters wrap heterogeneous applications (e.g. different interfaces, protocols and/or data formats) so that they appear as homogeneous and thus easier to integrate [116][118]. One must bear in mind that the work presented in this thesis relates to transforming SOAP/POX message invocations over HTTP. Thus, the source and target protocols and data formats are equivalent (HTTP and XML respectively). However, the adapters in StoRHm address the heterogeneous *usage* of HTTP.

The POX protocol adapter is based on the RESTlet framework. RESTlet is a Java API for developing RESTful applications. The adapter extends the *Resource* base class and overrides the *acceptRepresentation* method (which handles POST requests). In addition to the adapter, the POX client is also RESTlet based. The classes in the RESTlet API reflect the influence of the REST architectural style e.g. *Components*, *Connectors*, *Resources* and *Representations*. In Fig. 5-14, the blue boxes are Components e.g. clients and servers. The small yellow boxes are connectors i.e. software elements that “manage network communication for a component” [3].

Sequence Diagram

The design of the algorithm used is shown in the sequence diagram of Fig. 5-15.

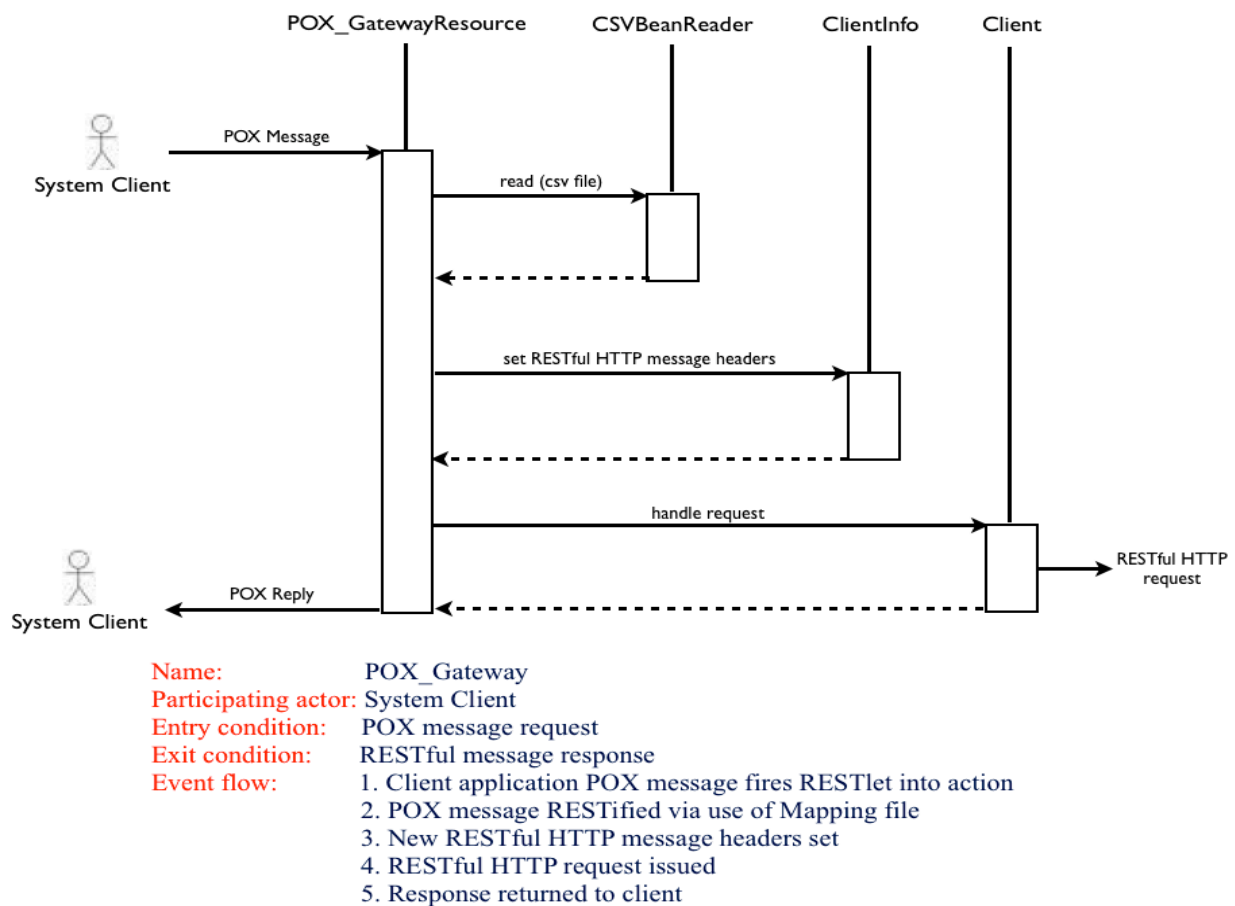


Figure 5-15 StoRHm v1 (POX) Protocol Adapter sequence diagram

Algorithm

The POX client configuration file is modified to point at the local POX adapter. The POX client application creates a client connector through which it issues its POST request. The POST request arrives at the server connector of the POX adapter where the Web Service name, which is appended to the adapter URI by the client, is parsed from the request URI and used to access the specific Web

Service entries in the mapping file. Informed by the mapping file, the adapter transforms the request into RESTful HTTP format. The adapter then creates a client connector to the RESTful server and issues the RESTful HTTP request. The RESTful Server connector receives the request, processes it and returns the response to the adapter from where it is returned to the POX client.

This means that the POX client is abstracted from the mappings i.e. from the point of view of the POX client, it issues a POX request and gets back a POX response. The fact that the POX request has been transformed into a visible RESTful HTTP request is transparent to the POX client. The mapping file plus protocol adapter must be located on the client as one of the requirements is that the benefits of RESTful HTTP e.g. caching is to be availed of. This is only possible in situations where the HTTP message is RESTful as it leaves the client machine. Note that this architecture also avoids the scenario associated with possible bottlenecks in a centralized system.

RESTful Web Services, which are hosted on an origin/target server, can return any MIME type; however, as the original client is a POX client (based on XML), the RESTful Web Services must return XML for requests that pass through the adapter. Note that it is the target server that marks its responses as cacheable and for how long. However, intermediaries such as caches can inspect the request messages to see if they have an up-to-date copy and if so, return it without the need to forward on the request to the target server [5].

Quality of Service

QoS features are supported: security and message reliability. However, as the adapter is concerned with generating RESTful HTTP requests, all the adapter needs to know is whether or not a particular QoS feature is required and then implement that feature in HTTP. For this purpose, a flag in the CSV file is used.

As regards security, the CSV file informs the adapter of a particular Web Service needing security and consequently the adapter opens up a secure connection using HTTPS.

The CSV file also informs the adapter of a particular Web Service requiring message reliability. As regards message reliability, the adapter follows the best practice approach outlined by Gregorio, which remodels POST as PUT [58]. This takes advantage of the idempotence of PUT, whereas POST is not idempotent. Idempotence means that it does not matter how many times a command is executed; the result is always the same. For example, setting a value to 5 will have the same effect regardless of the number of times it is executed. Therefore, whenever a message sent using an idempotent verb gets lost, the message is just re-sent.

POST is only used when a client (the adapter in this instance) does not know the full URI to PUT to. With POST, the client (adapter) uses a known top-level generic “collection” URI. The server, upon

receipt of the POST message to a collection resource, assigns the actual URI that the client (adapter) can subsequently send PUT messages to [45]. Thus, when the adapter outlined in this research receives a POX request where reliability is required, the adapter POSTs to a collection resource with no data in the entity body. The server generates and sends back the URI to be used by the adapter. The adapter then issues a PUT with the data from the POX request inside the entity body [58].

For example, to create an order, the adapter would save the POX order data and POST to, for example, a “/orders” collection URI (with no order data in the entity body of the message). This collection URI “/orders” refers to all the orders. Upon receiving this POST message, the server would generate and return (to the adapter) a unique URI to refer to the new order: for example, “/orders/57676”. This is the URI that the adapter can subsequently send PUT messages to (with the POX data in the entity body of the message).

If either the POST or PUT messages are lost, the adapter re-sends the lost message. In the case of the POST message getting lost, the server re-generates another URI for the adapter to use. This is not an issue as the generated URI’s are logical and not physical i.e. the URI expresses what resource is desired and not a static document [111]. For example, contrast the logical URI *http://www.parts-depot/parts/000000* with the physical URI *http://www.parts-depot/parts/000000.html*. The logical URI abstracts away the underlying implementation from the clients, whereas the physical URI refers to a physical HTML page. The logical URI approach enables the server’s underlying implementation to be changed without impacting the clients [5]. Once the adapter receives a successful response (to the PUT message), the adapter returns the response to the (POX) client.

Package/Class Diagrams

The package and class diagrams representing this design are outlined in Figures 5-16 and 5-17.

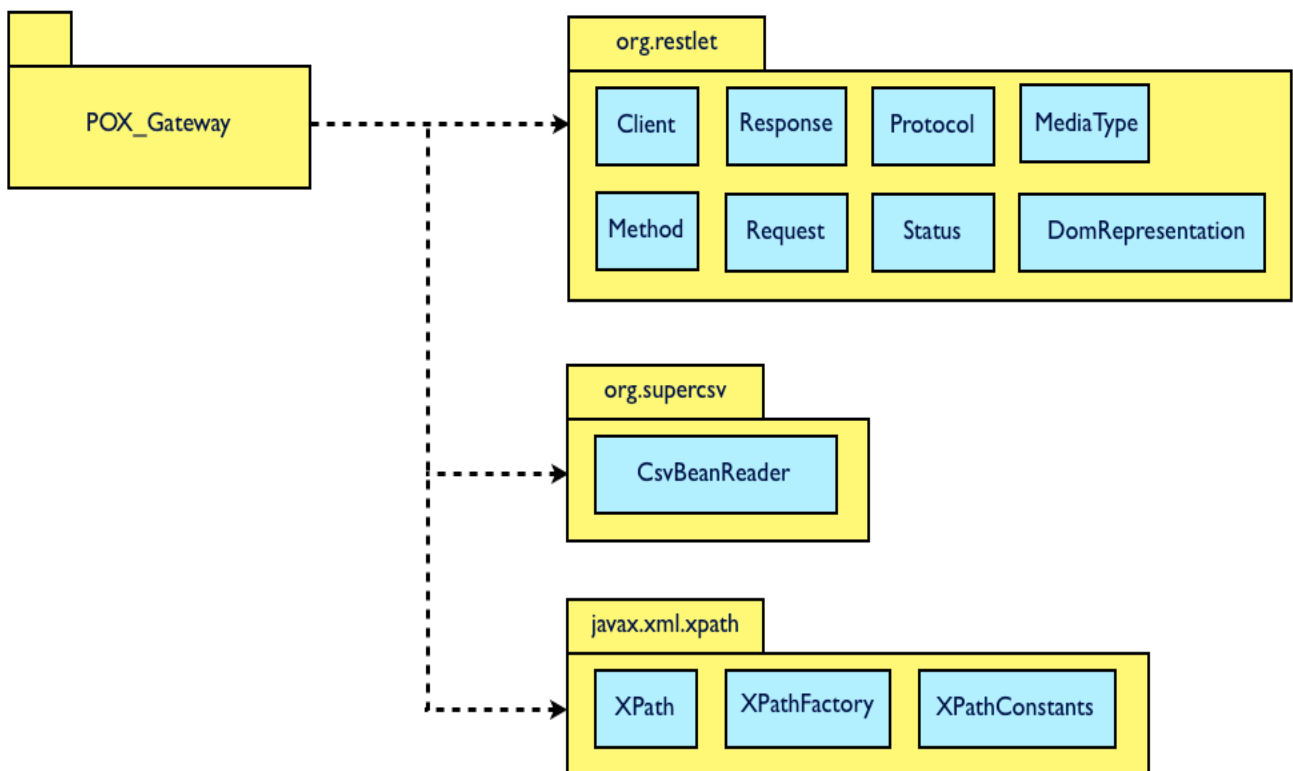


Figure 5-16 StoRHm v1 (POX) Protocol Adapter package diagram

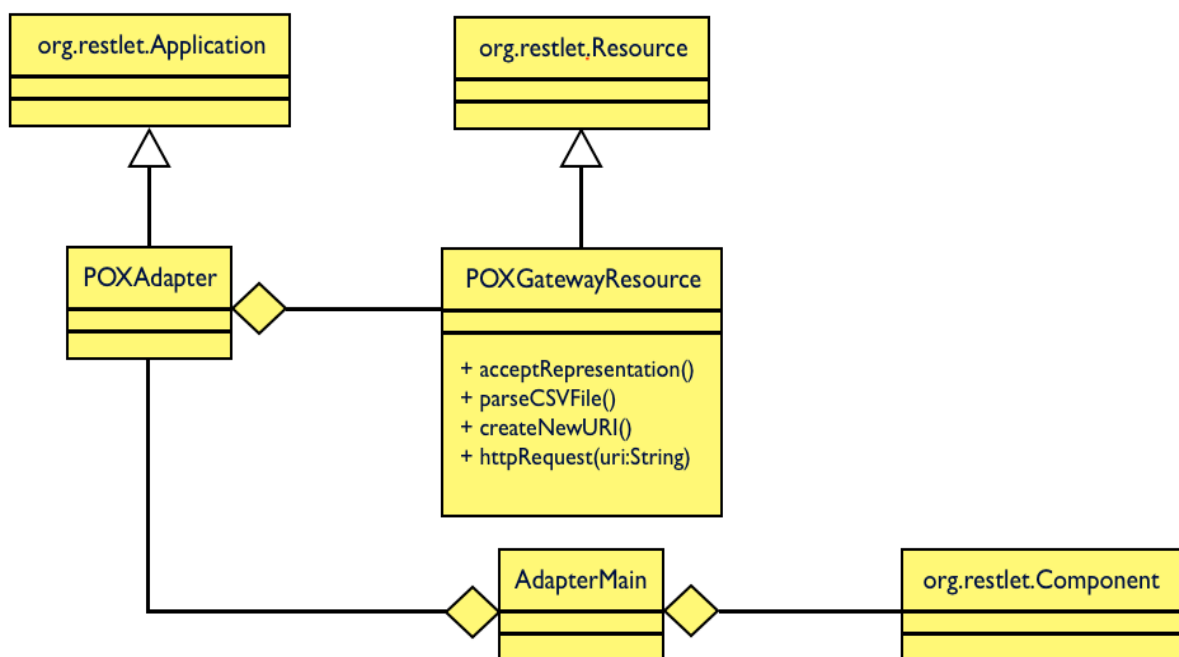


Figure 5-17 StoRHm v1 (POX) Protocol Adapter class diagram

Design decisions

As stated, the protocol adapter is implemented using the RESTlet framework and is deployed on the client side. The URI consisting of the protocol adapter address with the Web Service name appended

is inserted into the POX client configuration file. This ensures that future POX requests are redirected to the protocol adapter where the Web Service name is parsed from the URI and used as a key to the relevant entries in the CSV file.

5.4.3.3 StoRHm v1 (SOAP) Configuration Wizard

As with the POX artefact, the function of the Configuration Wizard is to setup the Mapping file which the runtime Adapter later uses when transforming the messages. Similarly, the wizard is an event-driven Java GUI based application i.e. it *extends JFrame* and *implements ActionListener*.

GUI Design

Fig. 5-18 outlines the StoRHm v1 (SOAP) configuration wizard GUI interface. Similar to the POX wizard interface, there are three sections; a top, a middle and a bottom section. Each section corresponds to at least one Java JPanel. The top section (in yellow), details the input required from the user regarding the WSDL file, the WSDL Schema, the optional WADL filename and the Root URI to be used on all the RESTful URI's. The middle section (in brown) outlines the Web Service operations parsed from the WSDL file. As SOAP is a well-defined message format, XPath expressions can be used for parsing SOAP message [107]. The middle panel also outlines the RESTful URI that will be appended to the Root URI; the QoS checkboxes, the HTTP verb and MIME type drop down listboxes. Lastly, the middle panel also contains the Build buttons that enable the user to insert the WSDL operation parameter names into the RESTful URI. The final section is the bottom section, which contains the OK and Cancel buttons.

WSDL	v1	URI button	Parse button	WADL Checkbox	
	v2	File button	Browse button		
WSDL Schema				Browse button	
WADL				Browse button	
HTTPWS Name			Root URI		
Operations	RESTful URI	QoS	Build buttons	HTTP Verbs	MIME Types
OK/Cancel buttons					

Figure 5-18 StoRHm v1 (SOAP) Configuration Wizard GUI Layout

Layout Managers

Fig. 5-19 details the layout managers used to organise the GUI interface. The top section (in yellow), is controlled overall by a 4-row, 1-column GridLayout manager. A FlowLayout manager controls each row within the top section. The top WSDL row contains three sub-panels: one has a GridLayout of 2-rows, 1-column (WSDL v1 and v2 radio buttons); another has a GridLayout of 2-rows, 1-column (the URI/File radio buttons) and the other panel has 2 rows and 2 columns (the Parse and Browse buttons and the WADL checkbox). The second FlowLayout row is for the WSDL Schema. The third row in the top section is for WADL and the last row is for the Root URI and the read-only HTTP WS Name (which is parsed from the WSDL file). The middle section (in brown) is managed by a GridLayout with 1 row that has 6 columns. This row matches the brown row in Fig. 5-18. The final section on the GUI is a FlowLayout for the OK and Cancel buttons.

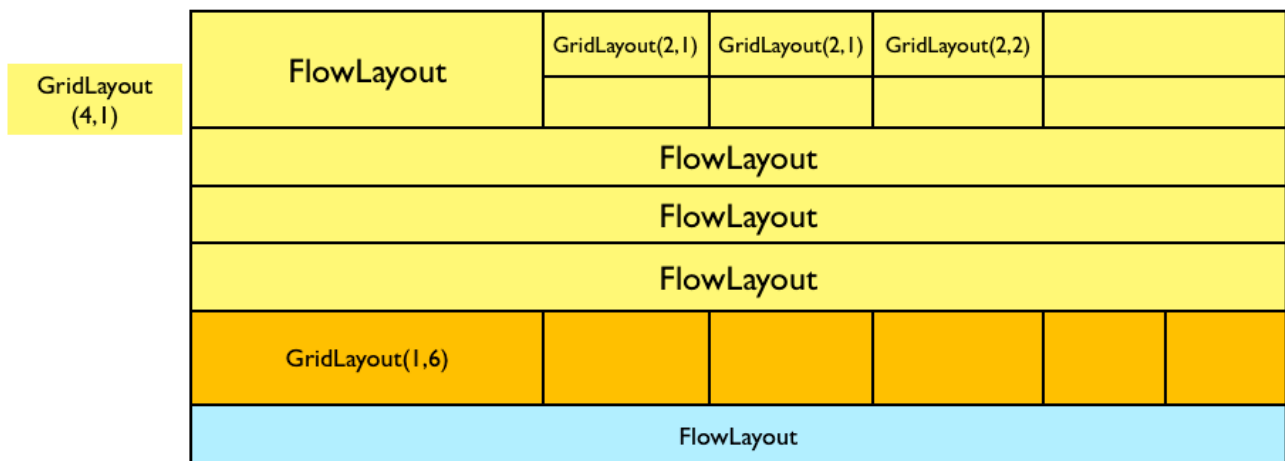


Figure 5-19 StoRHm v1 (SOAP) GUI Layout Managers

Example

Fig. 5-20 shows the initial screen of the wizard.

Mapping

WSDL : ☒ v1.1 ☒ URI ☐ WADL ?
☐ v2.0 ☐ File

Schema

WADL :

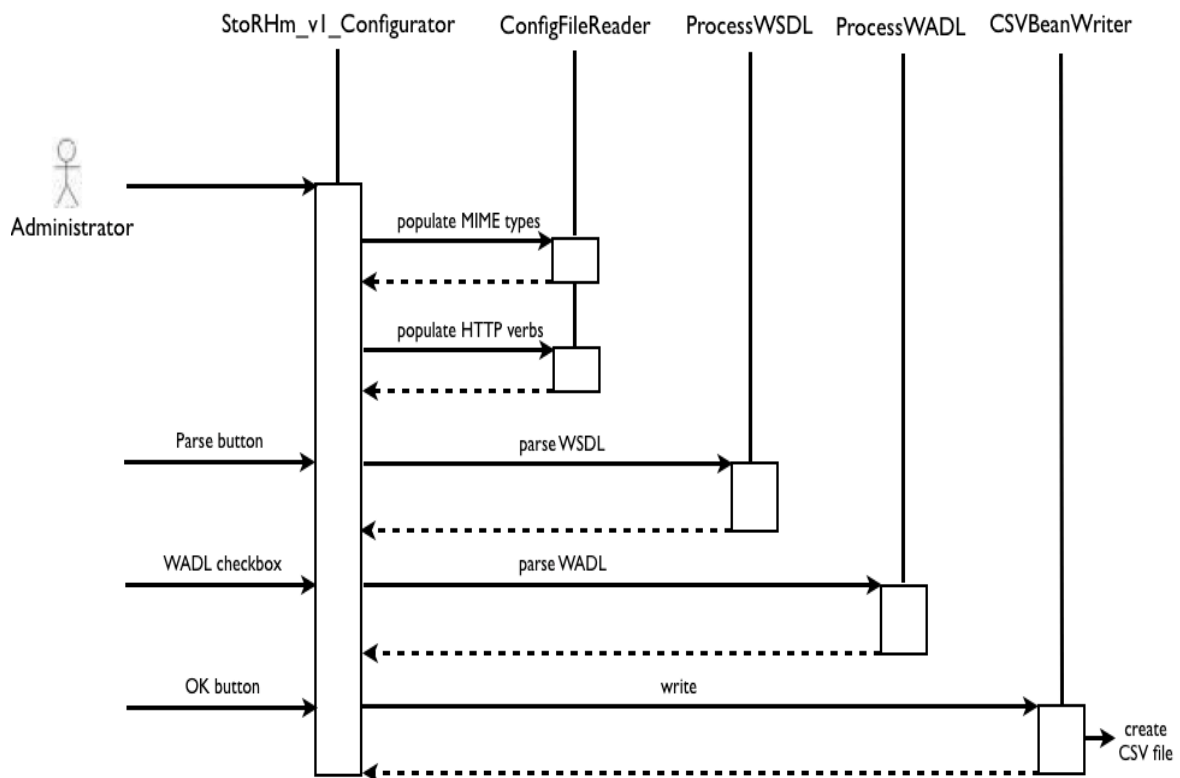
HTTP Web Service Name : Root URI :

Specific Interface	RESTful URI	Quality of Service			Uniform Interface	MIME Type
<input type="text"/>	<input type="text"/>	<input type="checkbox"/> Security	<input type="checkbox"/> Reliability	<input <="" td="" type="button" value="Build..."/> <td><input type="text"/></td> <td><input type="text"/></td>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="checkbox"/> Security	<input type="checkbox"/> Reliability	<input <="" td="" type="button" value="Build..."/> <td><input type="text"/></td> <td><input type="text"/></td>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="checkbox"/> Security	<input type="checkbox"/> Reliability	<input <="" td="" type="button" value="Build..."/> <td><input type="text"/></td> <td><input type="text"/></td>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="checkbox"/> Security	<input type="checkbox"/> Reliability	<input <="" td="" type="button" value="Build..."/> <td><input type="text"/></td> <td><input type="text"/></td>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="checkbox"/> Security	<input type="checkbox"/> Reliability	<input <="" td="" type="button" value="Build..."/> <td><input type="text"/></td> <td><input type="text"/></td>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="checkbox"/> Security	<input type="checkbox"/> Reliability	<input <="" td="" type="button" value="Build..."/> <td><input type="text"/></td> <td><input type="text"/></td>	<input type="text"/>	<input type="text"/>

Figure 5-20 Sample StoRHm v1 (SOAP) Configuration Wizard

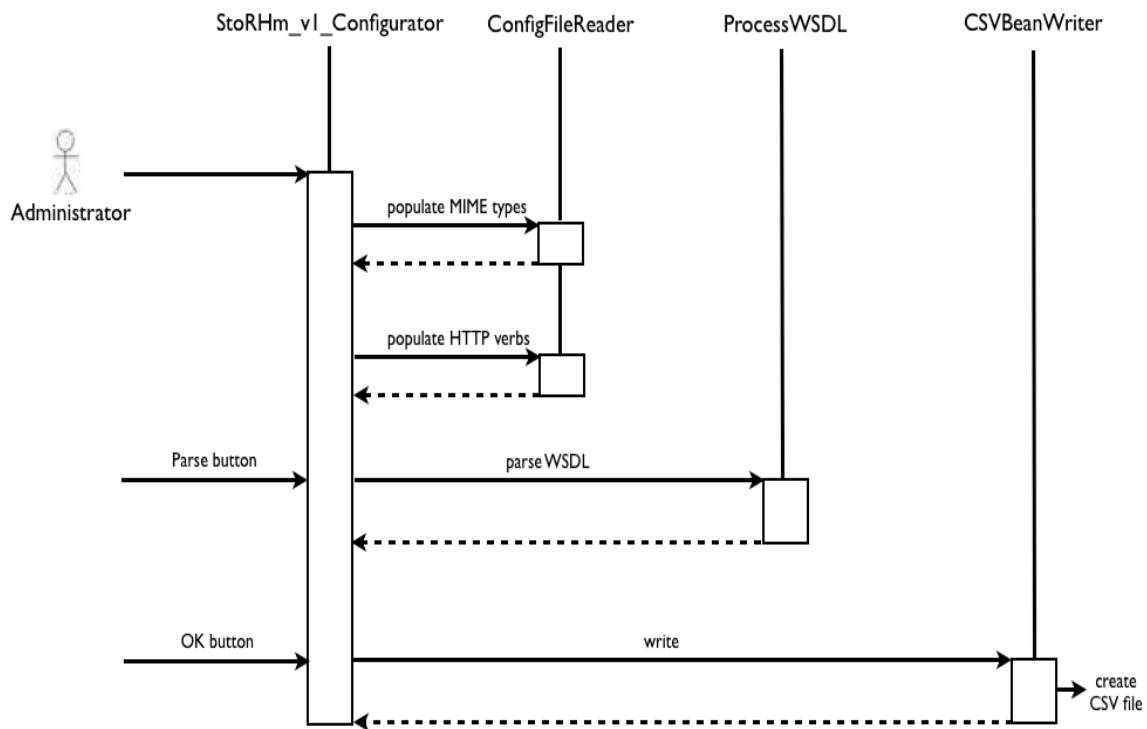
Sequence Diagrams

The design of the algorithm used is shown in the sequence diagrams of Fig. 5-21 and Fig. 5-22. Fig. 5-21 represents the situation where the user selects the WADL checkbox whereas in Fig. 5-22 the WADL checkbox is not chosen.



- Name:** StoRHm_v1_Configurator
- Participating actor:** Administrator
- Entry condition:** Startup application
- Exit condition:** Application exited
- Event flow:**
1. Upon startup, the MIME types and HTTP verbs read from configuration file
 2. WSDL file parsed
 3. WADL file parsed
 4. OK clicked, CSV file is created

Figure 5-21 StoRHm v1 (SOAP) Configuration Wizard sequence diagram (with WADL)



Name: StoRHm_v1_Configurator
Participating actor: Administrator
Entry condition: Startup application
Exit condition: Application exited
Event flow:

1. Upon startup, the MIME types and HTTP verbs read from configuration file
2. WSDL file parsed
3. OK clicked, CSV file is created

Figure 5-22 StoRHm v1 (SOAP) Configuration Wizard sequence diagram (without WADL)

Algorithm

The configuration wizard takes as input: a WSDL file, the WSDL schema (Netbeans splits them into two files) and the optional WADL file [84]. The user can access these files using both the HTTP and FILE protocols. When the user presses the “Parse” button, the WSDL file and its associated schema are parsed. The Web Service parsed from the WSDL file is checked to ensure that it has a HTTP binding. If so, the Web Service name is populated into the “*HTTP Web Service Name*” field. The operations supported by the Web Service are populated into each of the “*Specific Interface*” drop down listboxes. If the user has not selected the WADL checkbox, then the “*HTTP Verbs*” and “*MIME Types*” dropdown listboxes are populated from a configuration file. The user can click the “*Build*” button and a wizard (detailed in Chapter 6) enables the user to build a parameterised URI based on the WSDL operation parameter names. This feature enables the insertion of XPath

expressions into the RESTful URI that the protocol adapter can use when parsing the SOAP message at runtime.

As with the POX configuration wizard, if the user has selected the WADL checkbox, then the WADL file is parsed and the “*RESTful URI*”, “*HTTP Verbs*” and “*MIME Types*” sections are populated based on the WADL file. For example, each specific resource (URI) will support certain verbs and certain media types. By populating these areas based on the WADL file this means that it is impossible to send an incorrect verb or MIME type to that URI.

The user then creates a mapping between the operations, the RESTful URIs and the HTTP verb. The user also selects the QoS checkboxes buttons to indicate the QoS required. Lastly, the user selects the response type required (if WADL is selected then this will already be done). The response type is “text/xml” by default. At this point the user presses OK and the mapping file is created.

Package/Class Diagrams

The package and class diagrams representing this design are shown in Fig. 5-23 and Fig. 5-24.

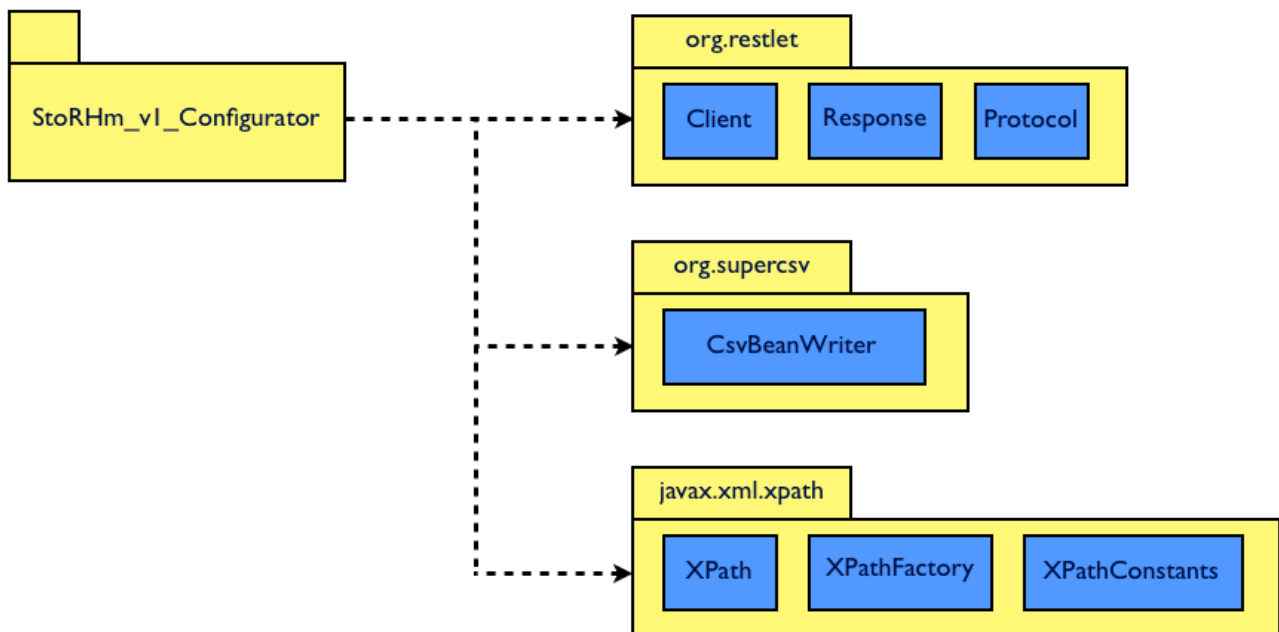


Figure 5-23 StoRHm v1 (SOAP) Configuration Wizard package diagram

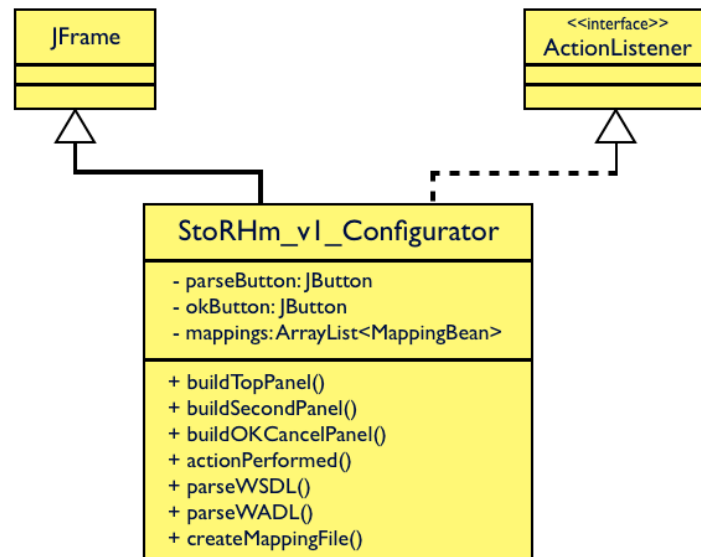


Figure 5-24 StoRHm v1 (SOAP) Configuration Wizard class diagram

Design decisions

The Mapping file is implemented as a Comma Separated Value (CSV) file and is created by the configuration wizard. Fig. 5-25 shows a sample CSV file. The CSV file is identical to the POX configuration CSV file detailed earlier (Fig. 5-13) except that the SOAP Operation column is now populated. The columns correspond to:

- Web Service name
- SOAP operation
- HTTP verb
- MIME Type
- URI to use (containing the XML XPath expressions to use inserted in curly braces)
- QoS features to apply

Web Service Name RESTful URI	SOAP Operation	HTTP Verb	MIME type QoS
PhoneDirectory, http://.../phoneDirectory/,	addPhoneNumber,	POST, No Security, Reliability	text/xml
PhoneDirectory, http://.../phoneDirectory/,	deleteAllNumbers,	DELETE, No Security, No Reliability	text/xml
PhoneDirectory, http://.../phoneDirectory/,	getAllNumbers,	GET, No Security, No Reliability	text/xml
PhoneDirectory, http://.../phoneDirectory/{phoneNo},	deletePhoneNumber,	DELETE, No Security, No Reliability	text/xml
PhoneDirectory, http://.../phoneDirectory/{surname}/{firstName},	getPhoneNumber,	GET, No Security, No Reliability	text/xml
PhoneDirectory, http://.../phoneDirectory/{surname}/{firstName},	updatePhoneNumber,	PUT, Security, No Reliability	text/xml

Figure 5-25 StoRHm v1 (SOAP) CSV file

5.4.3.4 StoRHm v1 (SOAP) Protocol Adapter

The runtime protocol adapter design is illustrated in Fig. 5-26.

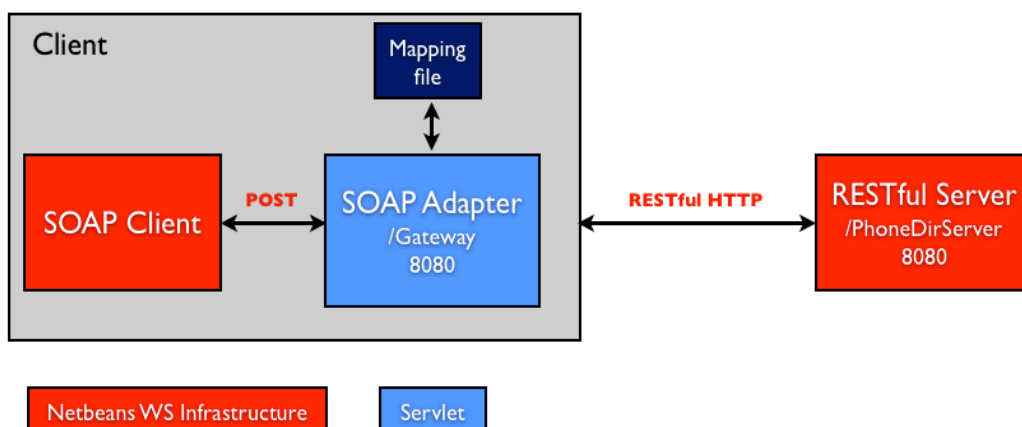


Figure 5-26 StoRHm v1 (SOAP) Protocol Adapter design

The design is based on the both servlets and Netbeans native SOAP WS infrastructure. In this instance, the backend RESTful Server hosts both SOAP and RESTful Web Services. The SOAP WS are developed initially so that the Netbeans infrastructure can generate a WSDL automatically. Once the WSDL is generated, Netbeans wizards can generate the SOAP client code automatically. The backend RESTful WS services are generated using Netbeans wizards and are the target of transformed SOAP client requests.

Sequence Diagram

The design of the algorithm used is shown in the sequence diagram of Fig. 5-27.

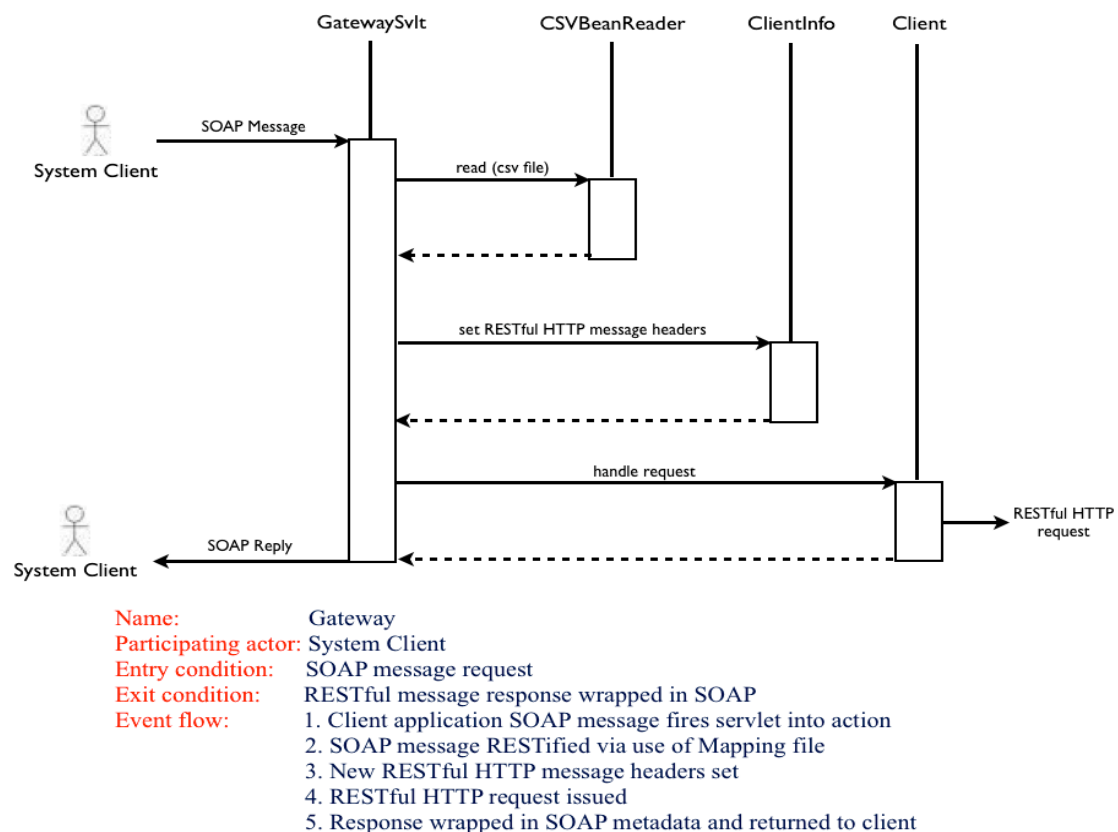


Figure 5-27 StoRHm v1 (SOAP) Protocol Adapter sequence diagram

Algorithm

The SOAP client WSDL file is modified to point at the adapter. The WS name is appended to this URI also. The client code, which is automatically generated by Netbeans in this version of the architecture, is executed. The POST request arrives at the StoRHm v1 adapter where the Web Service name is parsed from the request URI. The SOAP operation is also parsed from the incoming SOAP request. Both the Web Service name and SOAP operation and then used as a key to the relevant entries in the mapping file. Informed by the mapping file, the adapter transforms the request into RESTful HTTP format. The adapter then issues the RESTful HTTP request. The RESTful Server receives the request, processes it and returns the response to the adapter. The adapter wraps the response in SOAP metadata and returns it to the SOAP client.

The same advantages that hold for the POX architecture apply here also: the client is abstracted from the mappings; the mapping file plus protocol adapter must be located on the client so as to avail of Web infrastructure efficiencies and avoid the issues associated with bottlenecks in a centralized system. In addition, as the original client is SOAP based and therefore XML based, the RESTful WS requests emanating from the adapter will always request XML as the MIME type. However, this

does not mean that the RESTful WS in this architecture cannot return other media types. This is one of the advantages of RESTful WS – their flexibility in dealing with various client types.

Quality of Service

QoS features are supported as per the POX artefact i.e. the CSV file informs the adapter of the QoS required, not SOAP headers. However, all responses going back to the client must be wrapped in a SOAP envelope in order for the SOAP client to understand the response.

Package/Class Diagrams

The package and class diagrams representing this design are outlined in Figures 5-28 and 5-29.

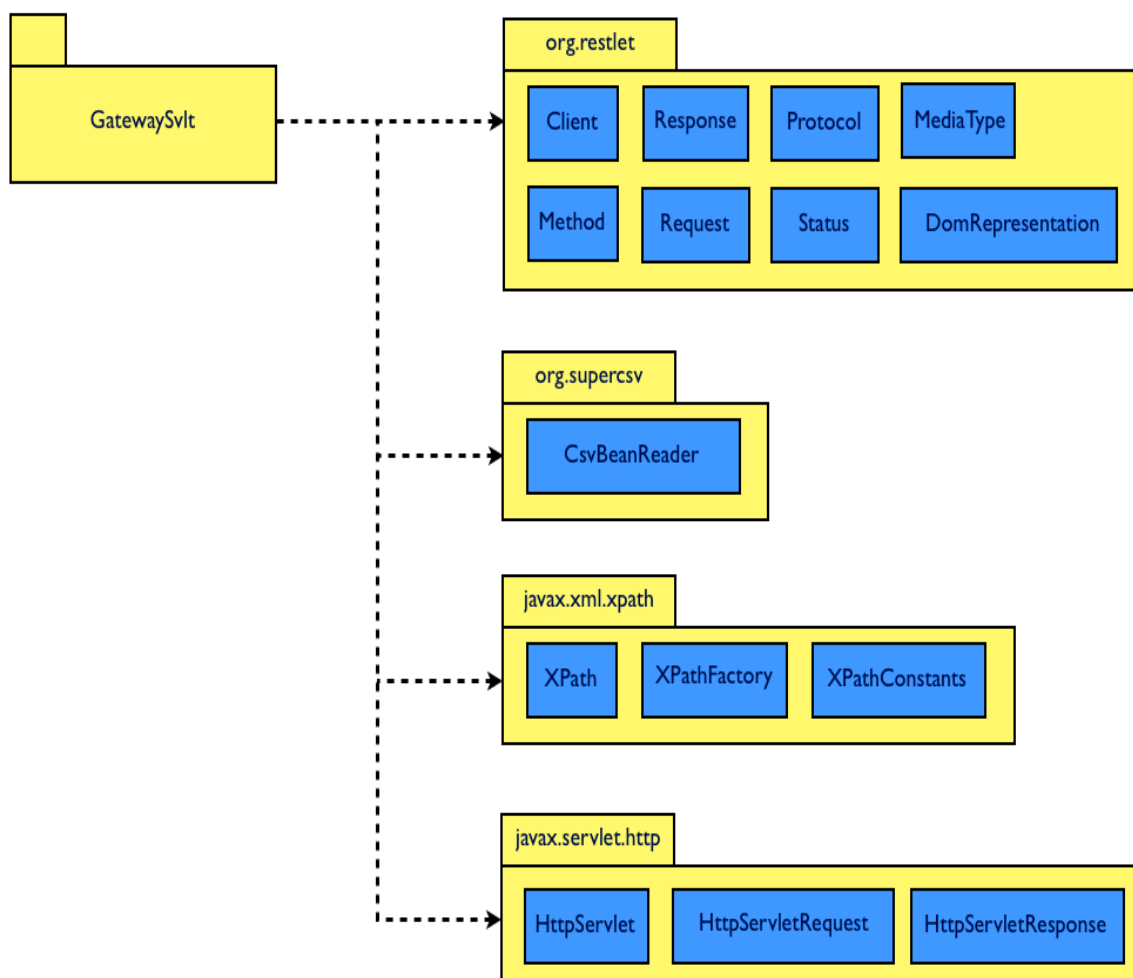


Figure 5-28 StoRHm v1 (SOAP) Protocol Adapter package diagram

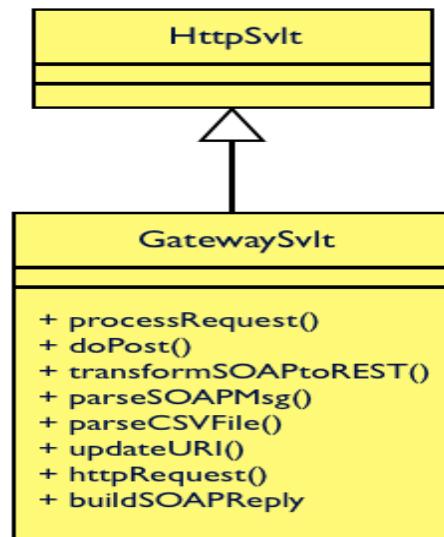


Figure 5-29 StoRHm v1 (SOAP) Protocol Adapter class diagram

Design decisions

The protocol adapter is implemented as a servlet i.e. the adapter extends *HttpServlet* and overrides *doPost*. The adapter is deployed on the client side. The URI consisting of the protocol adapter address with the Web Service name appended is inserted into the *soap:address* element in the WSDL file. The SOAP requests are thus redirected to the protocol adapter where the Web Service name is parsed from the URI.

5.5 StoRHm v2

In this section, the Requirements, Architecture and Design of StoRHm v2 is explained in detail.

5.5.1 StoRHm v2 requirements

The requirements for v2 are the same as in v1 (section 5.4.1) with the additional requirement of Semantic Web intelligence being used to automate the configuration stage.

5.5.2 StoRHm v2 architecture

StoRHm v2 is architecturally similar in many respects to the architecture of StoRHm v1. However, where v2 is different has a significant impact. As outlined earlier, the architectures are split into two distinct phases: the initial configuration wizard that produces the mapping file and the runtime adapter engine which uses this mapping file to transform the messages.

StoRHm v2's architecture (Fig. 5-30) is similar to that of the StoRHm v1 architecture outlined in Fig. 5-6. The difference, which is highlighted in red, notes the use of Semantic Web technologies to automate the configuration element.

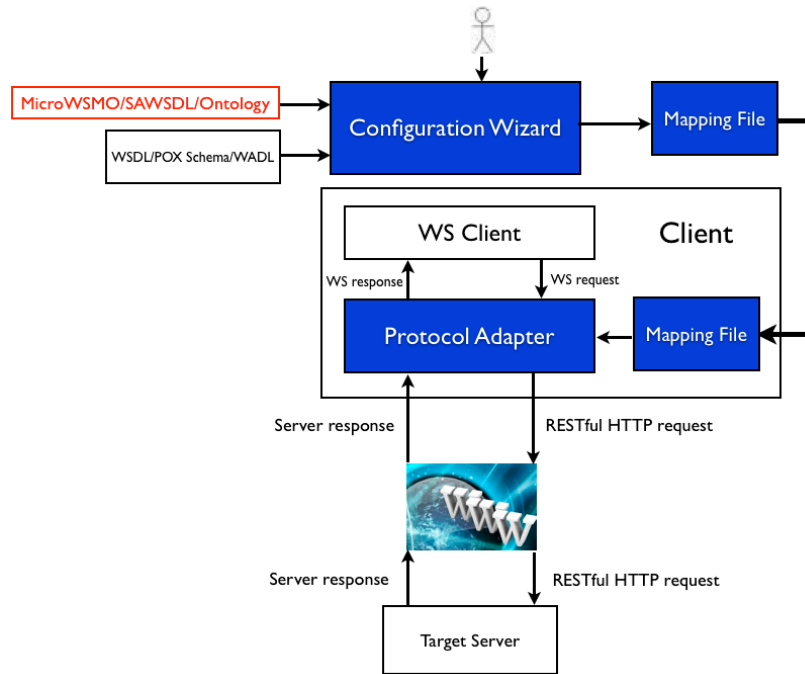


Figure 5-30 StoRHm v2 Architecture

5.5.3 StoRHm v2 Design

The critical and only difference architecturally between the two versions of StoRHm is that version 2 is semantically informed and thus automation of the mapping is possible. This is due to the fact that semantically annotating disparate elements to point to the same real world concept enables automated matching [128]. Obviously this has an impact on the design of the Configuration Wizard. The design of the Protocol Adapter was influenced by a desire to move to an integrated technology solution. Sections 5.5.3.1 and 5.5.3.2 outline the design of the configuration wizard and protocol adapter respectively in StoRHm v2.

5.5.3.1 StoRHm v2 Configuration Wizard

The introduction of the Semantic Web technologies to automate the configuration element was the primary reason for the evolution of StoRHm to a second version. As per StoRHm v1, the function of the Configuration Wizard is to setup the Mapping file for subsequent use by the runtime Adapter when transforming XML Web Service messages. Similarly, the wizard is an event-driven Java GUI based application.

GUI Design

Fig. 5-31 outlines the StoRHm v2 configuration wizard GUI interface. There are three (colour-coded) sections. Each section corresponds to at least one Java JPanel. The top section (in yellow), is where the radio buttons reside, enabling the user to specify whether this is a SOAP or POX configuration. The middle section (in brown) details the input required from the user regarding the semantically annotated files i.e. the SAWSDL, MicroWSMO and Ontology filenames. The final section is the bottom section, which contains the OK and Cancel buttons.

Fig. 5-31 shows the screen layout used.

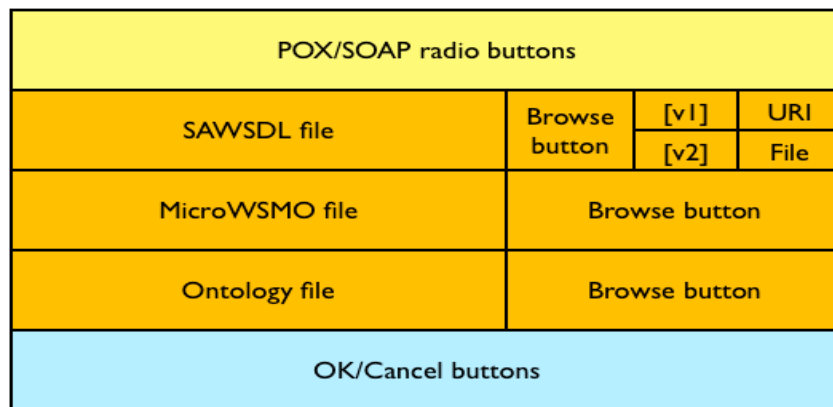


Figure 5-31 StoRHm v2 Configuration Wizard GUI Layout

Layout Managers

Fig. 5-32 details the layout managers used to organise the GUI interface. The JFrame container is organised overall as a BorderLayout on the y-axis. The screen is then divided into a 4-row, 1-column GridLayout (yellow portion of Fig. 5-32). FlowLayout managers control all rows on the grid. The first row is for the SOAP/POX radio buttons; the second row is for the SAWSDL file; the next row is for the MicroWSMO file and the last row is for the ontology. The final section on the GUI (after the grid) is a FlowLayout for the OK and Cancel buttons.

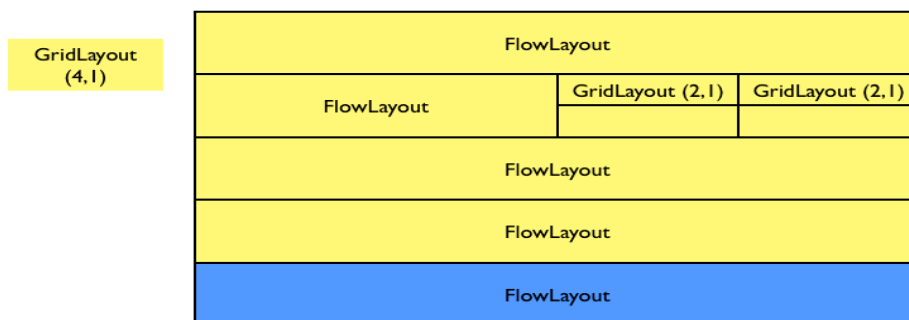


Figure 5-32 StoRHm v2 GUI Layout Managers

Example

Fig. 5-33 shows the initial screen of the wizard. Part (a) is when POX is selected; part (b) is when SOAP is selected.

The screenshot shows a dialog box titled "Configuration Wizard". At the top, there are two radio buttons: "POX" (which is selected) and "SOAP". Below this, there are three input fields, each with a "Browse..." button to its right: "SAWSDL:", "MicroWSMO:", and "Ontology:". To the right of these fields, there are two radio buttons: "URI" and "File", with "File" being selected. At the bottom of the dialog, there are two buttons: "OK" (with a green checkmark icon) and "Cancel" (with a red X icon).

Figure 5-33 (a) StoRHm v2 Configuration Wizard (POX)

The screenshot shows a dialog box titled "Configuration Wizard". At the top, there are two radio buttons: "POX" and "SOAP" (which is selected). Below this, there are three input fields, each with a "Browse..." button to its right: "SAWSDL:", "MicroWSMO:", and "Ontology:". To the right of these fields, there are four radio buttons: "v1.1", "v2.0", "URI", and "File". "v1.1" and "File" are selected. At the bottom of the dialog, there are two buttons: "OK" (with a green checkmark icon) and "Cancel" (with a red X icon).

Figure 5-33 (b) StoRHm v2 Configuration Wizard (SOAP)

Sequence Diagrams

The design of the algorithm used is shown in the sequence diagrams of Fig. 5-34. Fig. 5-34 (a) reflects when the user has selected POX and Fig. 5-34 (b) reflects when the user has selected SOAP:

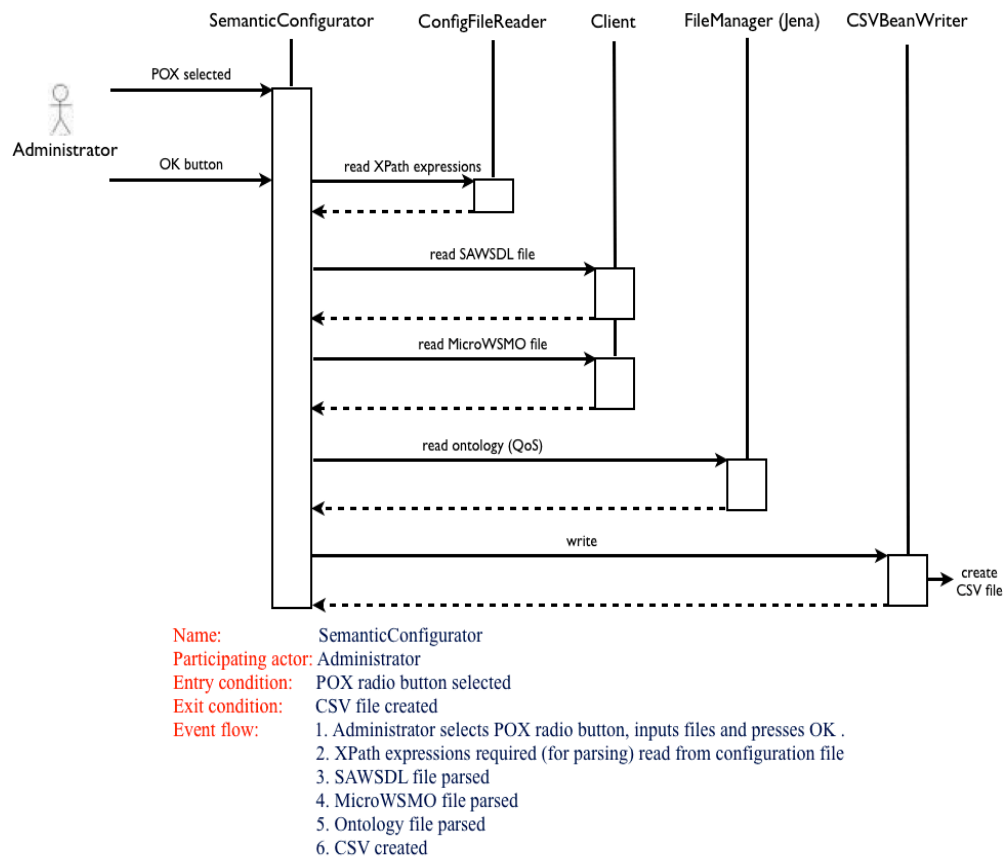


Figure 5-34 (a) StoRHm v2 (POX) Configuration Wizard Sequence Diagram

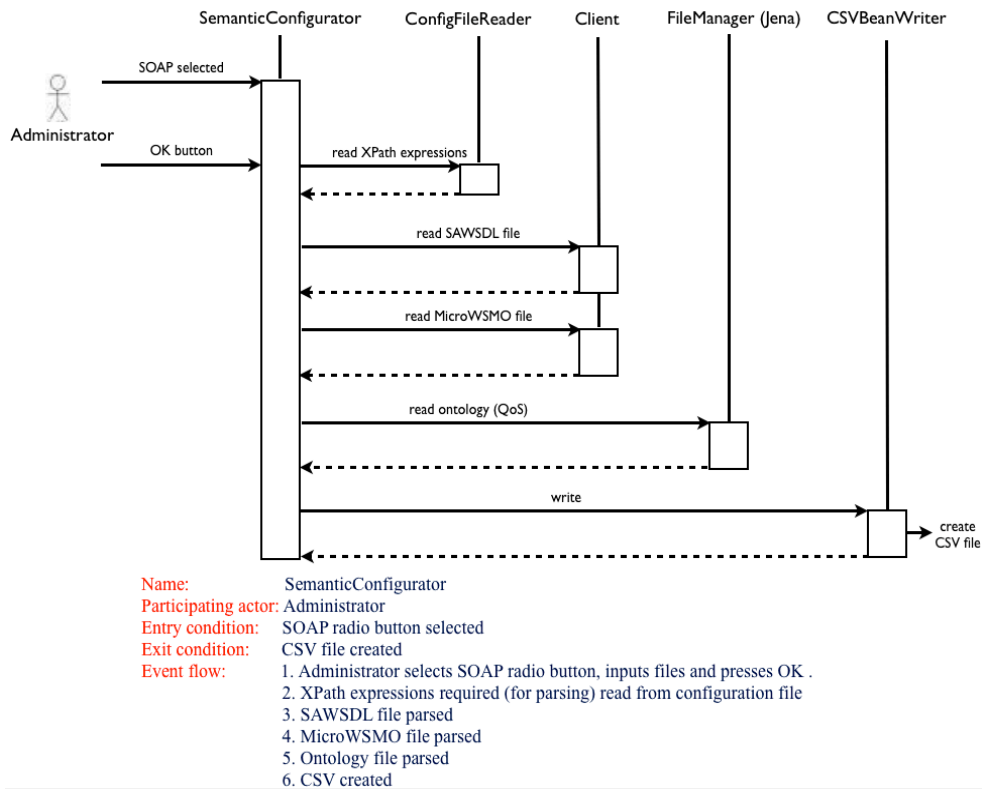


Figure 5-34 (b) StoRHm v2 (SOAP) Configuration Wizard Sequence Diagram

Algorithm

The configuration wizard takes the following semantically annotated input:

- a SAWSDL file representing the SOAP/POX WS description
- a MicroWSMO file representing the equivalent RESTful WS description
- the shared/common ontology outlining the QoS required

The flow diagram as per Fig. 5-35 outlines the algorithm when the user clicks OK.

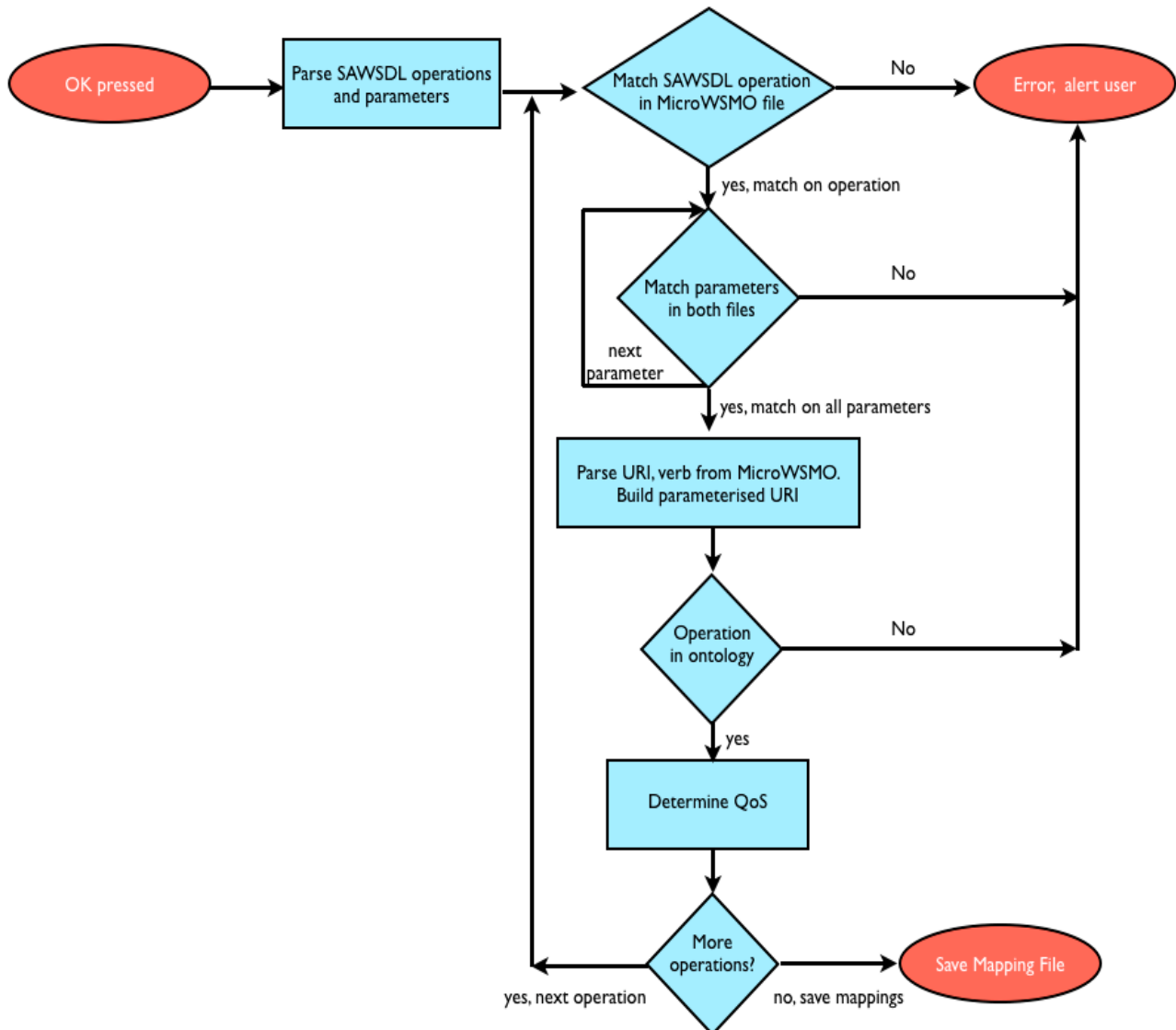


Figure 5-35 StoRHm v2 Configuration Wizard flow diagram

This is explained as follows:

- the semantically annotated operations and parameters are parsed from the SAWSDL file
- iterate through the SAWSDL operations:
 - is there a semantic match with an operation in the MicroWSMO file?
 - N: alert the user

- Y: loop for the operation parameters; is there a semantic match on all the parameters, regardless of order?
 - N: alert the user
 - Y: that operation matches, parse the URI and verb from the MicroWSMO file; build the parameterized URI using the element names from the SAWSDL file (as this reflects the structure of the incoming message to the adapter)
- does the operation exist in the ontology?
 - N: alert the user
 - Y: determine the QoS (if any) for that operation; save the mapping (in memory)
- if no error occurred, create the Mapping file.

The requirement to integrate both POX and SOAP configuration elements is achieved by the user selecting their request message format (SOAP or POX). This will determine whether or not the SOAP operation column in the mapping file is populated. Note that the “v1.1” and “v2.0” options refer to SAWSDL v1.1 and 2.0 respectively.

Package/Class Diagrams

The package and class diagrams representing this design are shown in Fig. 5-36 and Fig. 5-37.

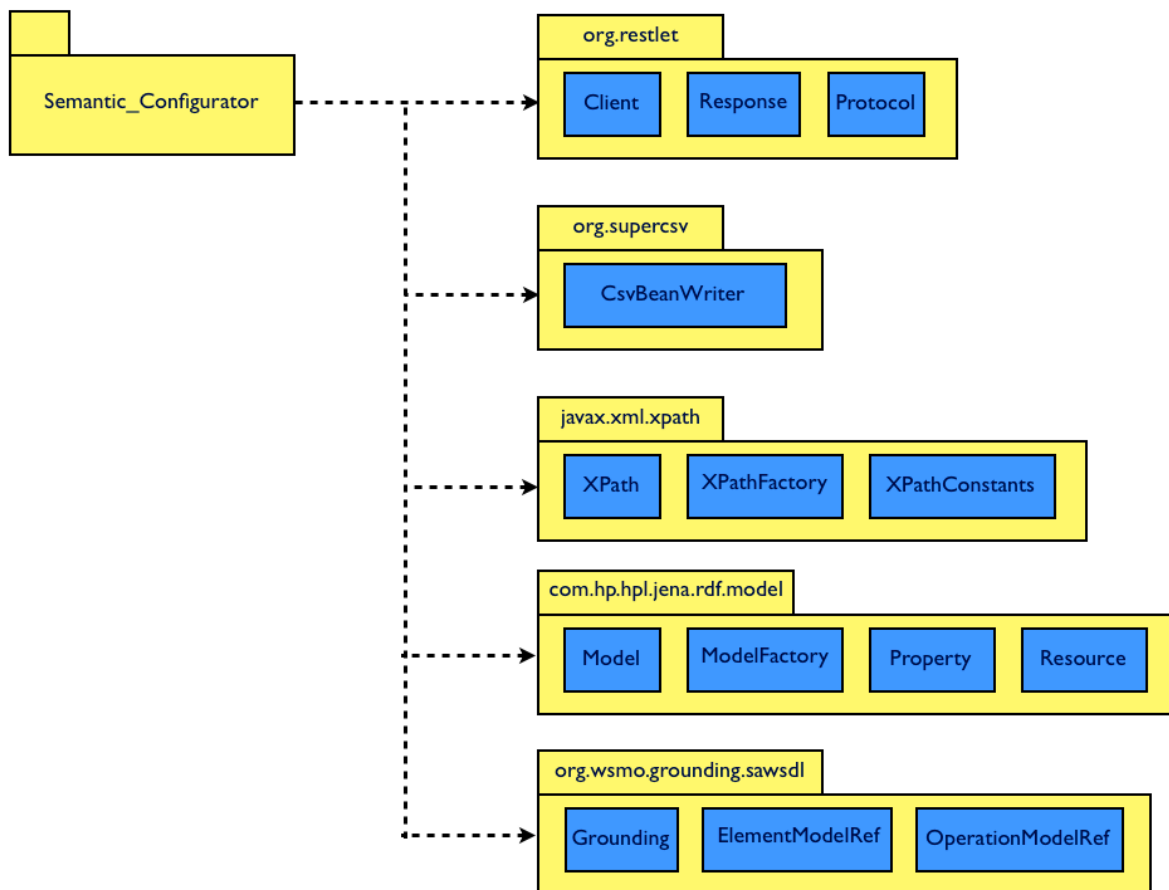


Figure 5-36 StoRHm v2 Configuration Wizard package diagram

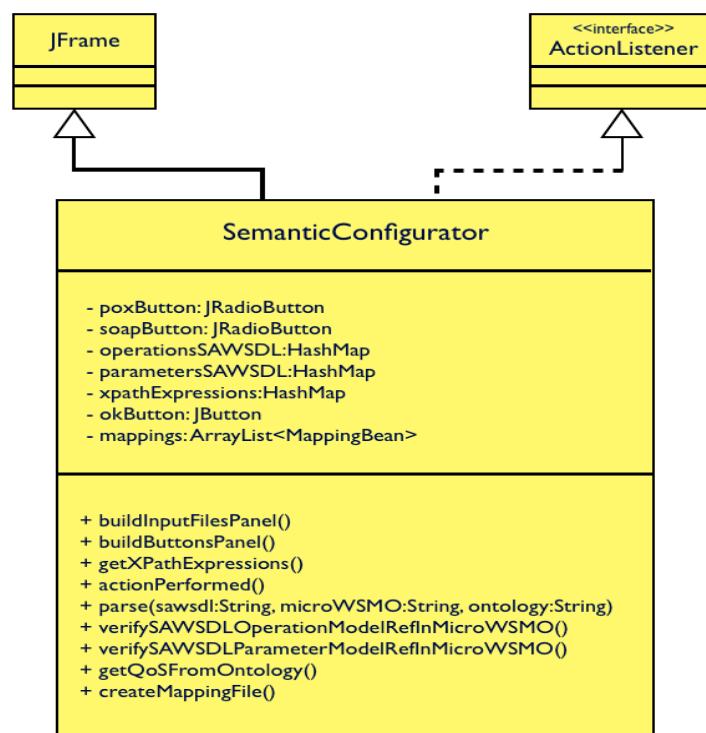


Figure 5-37 StoRHm v2 Configuration Wizard class diagram

Design decisions

As per StoRHm v1.

5.5.3.2 StoRHm v2 Protocol Adapter

The design of the StoRHm v2 protocol adapter (Fig. 5-38) is similar in many respects to the StoRHm v1 (POX) adapter (see Fig. 5-14). These similarities result from the design decision to use the RESTlet framework. However, the desire to design a solution that integrated both SOAP and POX into one application artefact is a noteworthy difference. The integrated adapter responds to SOAP requests via the URI “/StoRHm_Adapter” and POX requests via the URI “/POX_Adapter”. The client in StoRHm v2 is RESTlet based. The client is also integrated i.e. whether the client issues SOAP or POX request depends on configuration file settings.

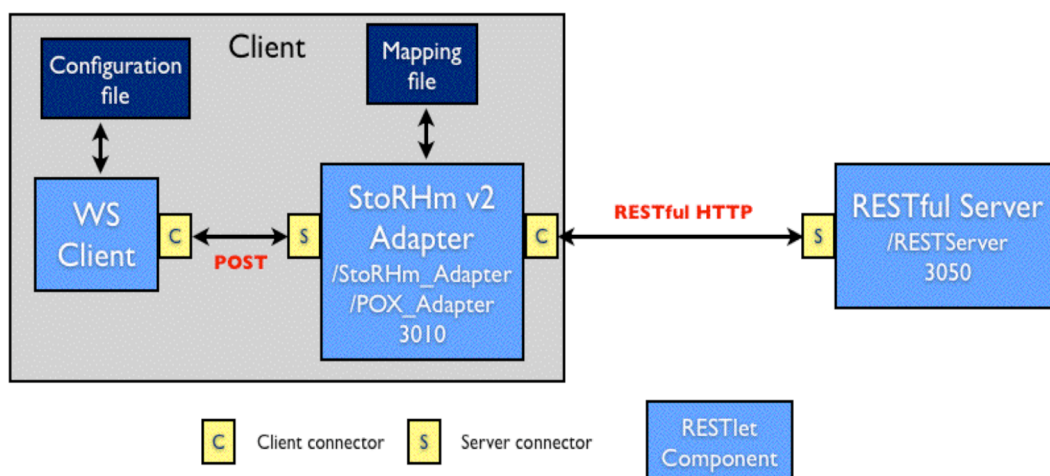


Figure 5-38 StoRHm v2 Protocol Adapter design

Sequence Diagrams

The design of the algorithm used is shown in the sequence diagram of Fig. 5-39. Fig. 5-39 (a) reflects the sequence when a SOAP message arrives at the adapter; Fig. 5-39 (b) reflects the sequence when a POX message arrives at the adapter. Point 5 in the event flow is the only difference i.e. where the RESTful response is wrapped in SOAP metadata for SOAP clients.

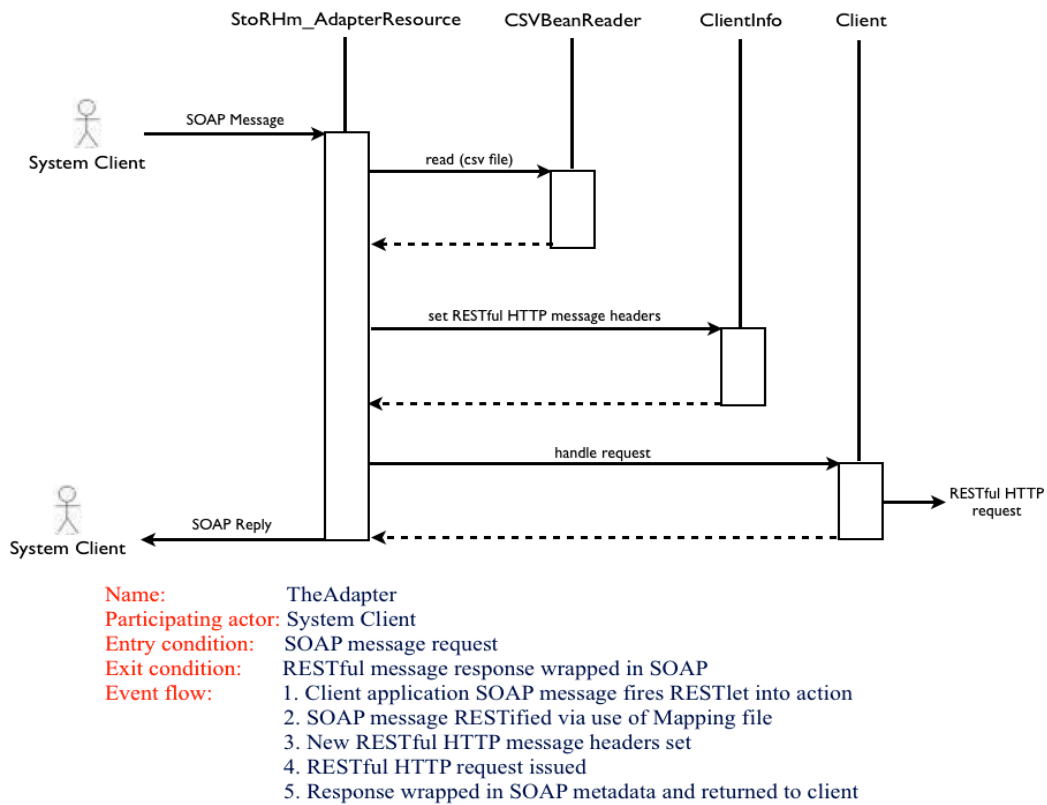


Figure 5-39 (a) StoRHm v2 Protocol Adapter (SOAP) sequence diagram

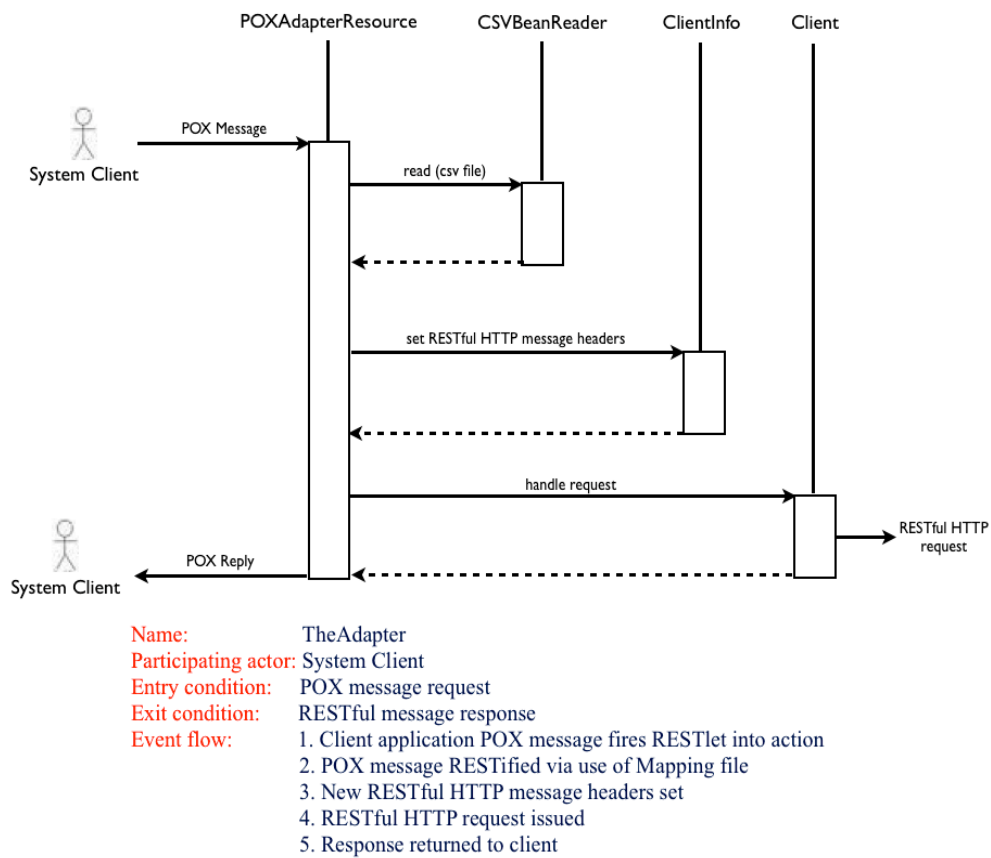


Figure 5-39 (b) StoRHm v2 Protocol Adapter (POX) sequence diagram

The algorithms used are as outlined previously in StoRHm_v1. This includes QoS.

Package/Class Diagrams

The package and class diagrams representing the design are outlined in Figures 5-40 and 5-41. Note that the design is similar to the POX design in StoRHm_v1 i.e. full use of RESTlet technology.

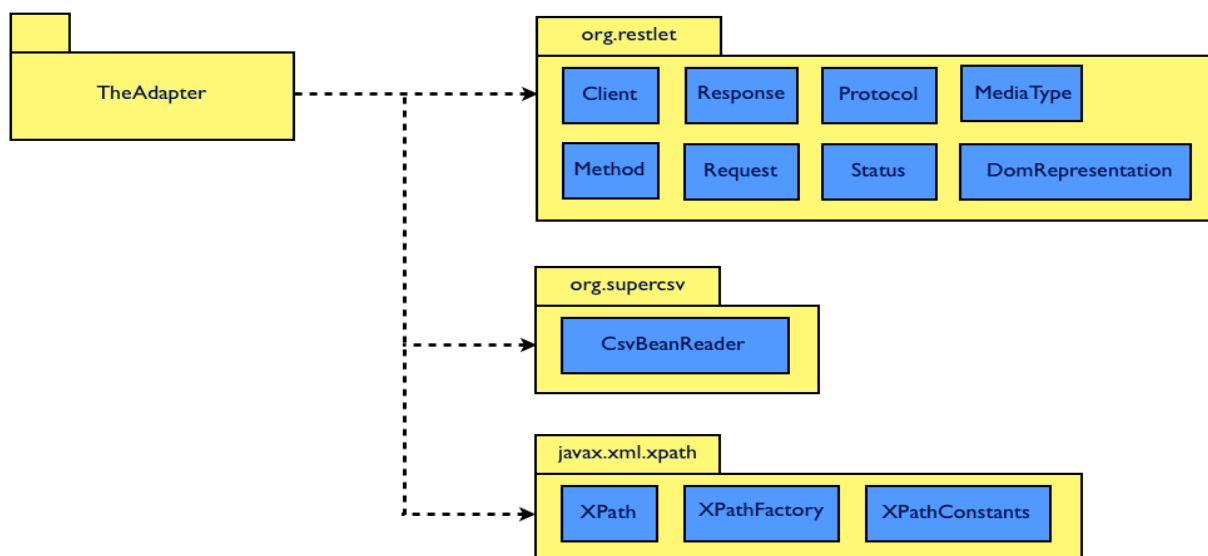


Figure 5-40 StoRHm v2 Protocol Adapter package diagram

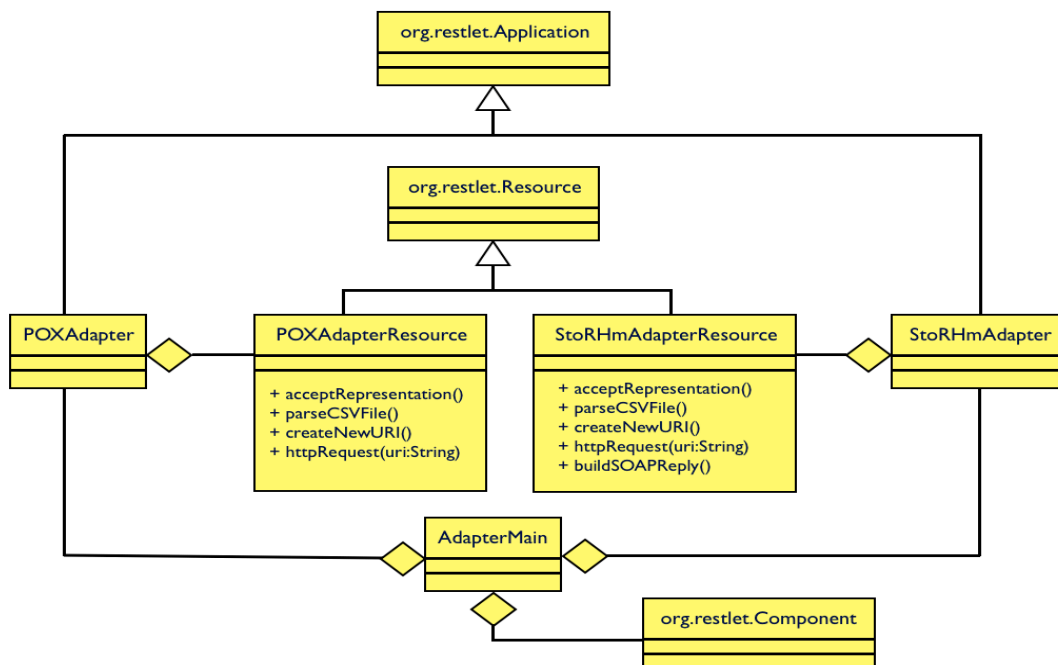


Figure 5-41 StoRHm v2 Protocol Adapter class diagram

Design decisions

As stated previously, the adapter is designed using the RESTlet framework and integrates both the SOAP and POX adapters.

5.6 Summary

This chapter outlined the StoRHm architecture underpinning the research contributions of this thesis. The research questions that led to the architecture were initially restated. Given that StoRHm has two distinct architectures, a project lifecycle outlining timelines and rationale for each version was presented. Both versions of the StoRHm architecture were then discussed. Each version supports a configuration element that generates a mapping file and a runtime adapter that, informed by the mapping file, transforms the incoming request message to RESTful format. It was noted that StoRHm v1 contains a manual, user-intensive configuration element. StoRHm v2 is a semantically-informed version of StoRHm v1. The configuration element of StoRHm v2 avails of semantic technologies to automate the mappings. The next chapter covers the implementation, test cases and evaluation of StoRHm.

Chapter 6

Implementation and Evaluation

6.1 Introduction

Chapter 5 outlined the architecture central to this thesis. This chapter will examine the implementation instances, test cases, performance and evaluation criteria used as part of this research. Table 6-1 lists the software artefacts:

Architecture	XML WS	Configurator	Adapter	Client	Server
StoRHm v1	POX	POX_Configurator	POX_Gateway (RESTlet)	POX_Client (RESTlet)	POXServer (RESTlet)
	SOAP	StoRHm_v1_Configurator	Gateway (Servlet)	Parts_Client PhoneDirClient (Netbeans SOAP)	PartsServer PhoneDirServer (Netbeans SOAP)
StoRHm v2	SOAP	SemanticConfigurator	TheAdapter (RESTlet)	StoRHm_Client (RESTlet)	StoRHm_Web_Services (Netbeans SOAP)
					RESTful_WS_Server (RESTlet)
	POX				POXServer (RESTlet)

Table 6-1 Software Artefacts

These artefacts total 10,790 lines of code approximately (excluding PhoneDirClient and PhoneDirServer). The breakdown is as follows: StoRHm_v1 (POX): 2855 lines of code;

StoRHm_v1 (SOAP): 3365 lines of code and StoRHm_v2: 4570 lines of code.

6.2 StoRHm v1 implementation

As outlined in Chapter 5, StoRHm v1 consists of two elements: a configuration wizard and a runtime protocol adapter. The implementation instances are now discussed. As stated previously, StoRHm v1 is capable of mapping both SOAP and POX Web Service requests to RESTful HTTP format. Despite being architecturally similar, their implementations are different. As in Chapter 5, in the interests of clarity, separate sections are required. To this end, subsections 6.2.1 and 6.2.2 refer to POX and subsections 6.2.3 and 6.2.4 refer to SOAP. The principal difference with regard to the protocol adapter is the fact that when implementing the SOAP adapter, the RESTlet framework was in its early stages. This was not the case when implementing the POX adapter some months later.

6.2.1 StoRHm v1 (POX) Configuration Wizard

The configuration function is supported by a wizard as shown in Figure 6-1.

Web Service	Elements	Operation	RESTful URI	Quality of Service	HTTP verb	MIME Type
ServiceViewAll	/Request/ServiceVie...	Insert...	/	<input type="checkbox"/> Security <input checked="" type="checkbox"/> Reliability	GET	text/xml
ServiceDeleteAll	/Request/ServiceVie...	Insert...	/	<input type="checkbox"/> Security <input type="checkbox"/> Reliability	DELETE	text/xml
ServiceAdd	/Request/ServiceVie...	Insert...	/	<input type="checkbox"/> Security <input type="checkbox"/> Reliability	POST	text/xml
ServiceView	/Request/ServiceVie...	Insert...	riceView/ACCOUNT_NUMBER}	<input type="checkbox"/> Security <input type="checkbox"/> Reliability	GET	text/xml
ServiceDelete	/Request/ServiceVie...	Insert...	riceView/ACCOUNT_NUMBER}	<input type="checkbox"/> Security <input type="checkbox"/> Reliability	DELETE	text/xml
ServiceUpdate	/Request/ServiceVie...	Insert...	riceView/ACCOUNT_NUMBER}	<input checked="" type="checkbox"/> Security <input type="checkbox"/> Reliability	PUT	text/xml

Figure 6-1 POX to RESTful HTTP Configuration Wizard

The structure of the GUI is a front-end for the CSV with some support. XML Schema informs the POX side of the mapping. The wizard prompts the user for the name of the Schema file. The file name can be URI based or file based. Using a configuration file, the wizard populates the Web Service names (column 1) and the Elements combo boxes (column 2). The Element combo boxes contain XPath expressions which are based on the element names and their structure in the Schema. The HTTP verb and MIME type columns (the last two columns in Fig. 6-1) are populated from configuration files. The user then selects a (POX) Web service and builds the semantically equivalent RESTful Web Service by using the “*Insert*” button. This builds up a RESTful URI with XPath expressions which are later used to parse the parameters from incoming POX requests.

The user selects the HTTP verb to use from the HTTP verb drop down combo box. The MIME Type drop down enables users to select different response types from the RESTful Web services. However, as the client is a POX client, this setting will always be XML. The user selects the QoS requirements for that RESTful HTTP service by selecting the relevant QoS checkbox(s) (column 5). The user then selects OK and the CSV file is populated. The CSV file is subsequently used by the protocol adapter, when transforming the POX request to RESTful HTTP format.

WADL, a description language for HTTP-based Web applications can be used to support the RESTful HTTP side of the mapping [84]. However, it is an optional feature as many RESTful HTTP services use HTML (with/without microformats) to describe their interface [86]. As opposed to building the URI manually, the WADL checkbox at the top right of Fig. 6-1 can be selected. If the WADL checkbox is selected, the WADL file is parsed enabling automatic population of the Root URI. The RESTful URI’s are parsed sequentially from the WADL file into the text fields in the RESTful URI column (column 4). Depending on the URI being parsed from the WADL file, the verbs supported by that URI are populated into the “HTTP verb” column and the MIME Types supported by that URI are populated into the “MIME Type” column. This means that the last two columns (HTTP verb and MIME Type) are populated from the WADL file and are URI-dependent e.g. if a URI does not support POST then POST will not appear in the drop down listbox. This will help prevent a URI from receiving a request for an unsupported verb/MIME type.

The WADL process is best explained with the aid of an example. Listing 6-1 is a sample WADL file reflecting a RESTful Banking Web Service.

```

<?xml version="1.0" encoding="UTF-8"?>
<application>
  <resources base="http://127.0.0.1:3050/RESTServer/">
    <resource path="/BankServices">
      <method name="POST" id="postXml">
        <response>
          <representation mediaType="text/xml"/>
        </response>
      </method>
      <method name="DELETE" id="deleteXml">
        <response>
          <representation mediaType="text/xml"/>
        </response>
      </method>
      <method name="GET" id="getXml">
        <response>
          <representation mediaType="text/xml"/>
        </response>
      </method>
      <resource path="/{branchCode}/{accountNo}">
        <param type="xs:string" style="template" name="branchCode"/>
        <param type="xs:string" style="template" name="accountNo"/>
        <method name="DELETE" id="deleteXml">
          <response>
            <representation mediaType="text/xml"/>
          </response>
        </method>
        <method name="GET" id="getXml">
          <response>
            <representation mediaType="text/xml"/>
          </response>
        </method>
        <method name="PUT" id="putXml">
          <response>
            <representation mediaType="text/xml"/>
          </response>
        </method>
      </resource>
    </resources>
  </application>

```

Listing 6-1 Sample WADL file

This Web Service is hosted at “*http://127.0.0.1:3050/RESTServer*” and it supports the following URI’s:

- “*/BankServices*” which supports the verbs POST, DELETE and GET. The response representation from each verb is the media type “text/xml” i.e. XML.
- “*/BankServices/{branchCode}/{accountNo}*” where {*branchCode*} and {*accountNo*} are string placeholders for incoming requests. This URI supports DELETE, GET and PUT and the response representation from each verb is also XML.

Knowing the URI’s, the verbs they support and the returned representation enables the semi-automation of the RESTful side of the mapping (as outlined earlier). This leads to the automatic population of certain fields, namely, the RESTful URI, the Root URI, the HTTP verb and MIME

type. Figure 6-2 reflects the screen display upon selecting the WADL checkbox from Figure 6-1. Note that the SOAP/POX elements still have to be manually input.

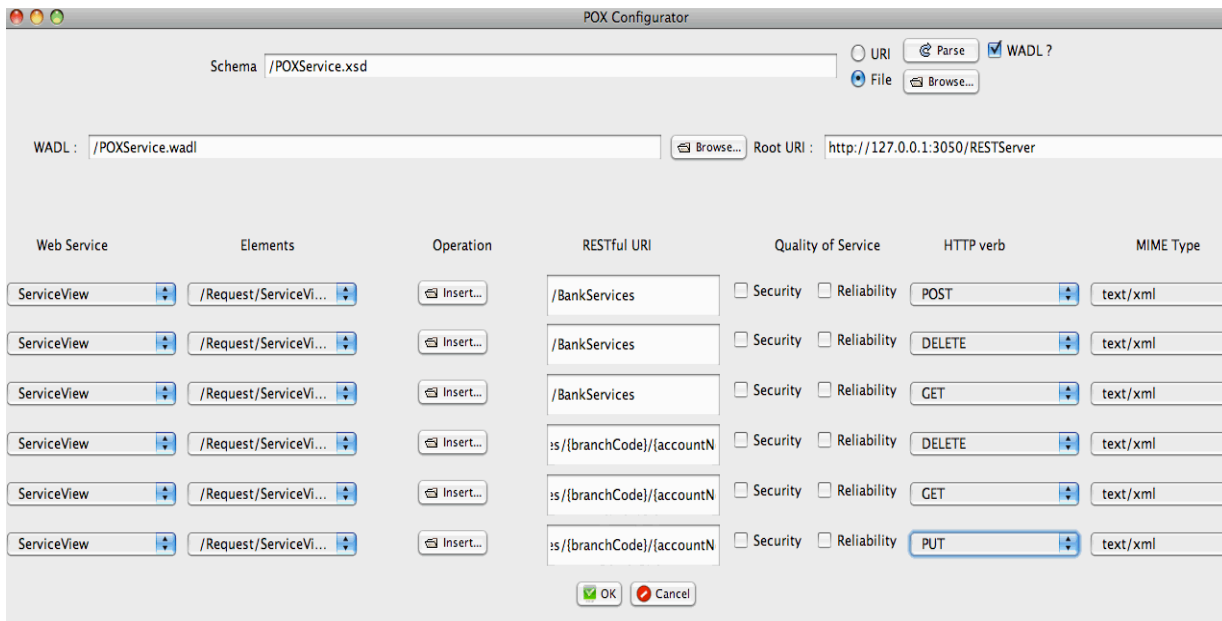


Figure 6-2 Configuration Wizard with WADL selected

6.2.2 StoRHm v1 (POX) Protocol Adapter

The protocol adapter is implemented using the RESTlet framework and deployed on the client. The RESTlet framework is a simple yet powerful Java-based API for developing RESTful applications. RESTlet enables the development of protocol-specific clients and servers (HTTP for the purposes of this thesis). The adapter, which acts as both a server (to the POX client) and a client (to the RESTful WS), is setup to listen on port 3010 at URI “*http://127.0.0.1/POXGateway*”. Note that this setup is shown in Fig. 5-14. The client-side configuration file is modified to re-direct POX service invocations via the adapter. This means that there is no re-compilation of the POX client necessary i.e. the impact on the POX client is minimal.

The POX client is a RESTlet based HTTP client. The client executes a POX request as normal and this request arrives at the adapter. The adapter parses the Web service name from the incoming URI. The Web Service name is then used to index the CSV file. The adapter, using the other entries for that row in the CSV file, maps the POX request to a RESTful HTTP request. This may involve the extracting of data from the POX request and inserting that data into the RESTful URI.

This process can be seen from Fig. 6-3:

(1) - The incoming URI (from the POX client) is pointing at the adapter, namely “POXGateway”. The Web Service name, namely “ServiceView” is appended to the URI.

(2) - The adapter parses “ServiceView” from the URI and uses the Web Service name to index the CSV file. The CSV file informs the adapter that the resultant HTTP request is a GET with a response type of “text/xml”. The second column in the CSV row is set to “N/A” for all POX requests. This column is the SOAP operation and consequently is only populated and required by SOAP requests. There are no QoS features with this Web Service i.e. no Security or Reliability.

(3) - Lastly, the CSV provides a “RESTful URI” column that enables the adapter to construct the URI for the HTTP message. This column contains the XPath expressions that are used on the POX request to elicit the values to use in the subsequent HTTP message. For example, “{/Request/ServiceView/BRANCH_CODE}” is an XPath expression that when applied to the original POX request results in the value “123456” being extracted. This value is inserted into the URI in the XPath expression’s location. Thus the BRANCH_CODE of “123456” and the ACCOUNT_NUMBER of “12345678” are extracted from the POX request and inserted into the RESTful URI request. Note that the mapping creates a relationship between the data encoded in the POX request and the resource structure of the RESTful URI, i.e. the data in the POX request is encoded in the URI.

Note: The POX request in Fig. 6-3 refers to port 3011 (as opposed to 3010). This was done so that samples messages could be viewed. An HTTP monitor was inserted between the POX client and the adapter. The monitor listened on port 3011 and re-directed the request to port 3010 (the POX adapter). The monitor was also inserted between the POX Adapter and the RESTful WS. The monitor listened on port 3051 and re-directed the request to port 3050 (the RESTful WS server). This pattern was used throughout the test period.

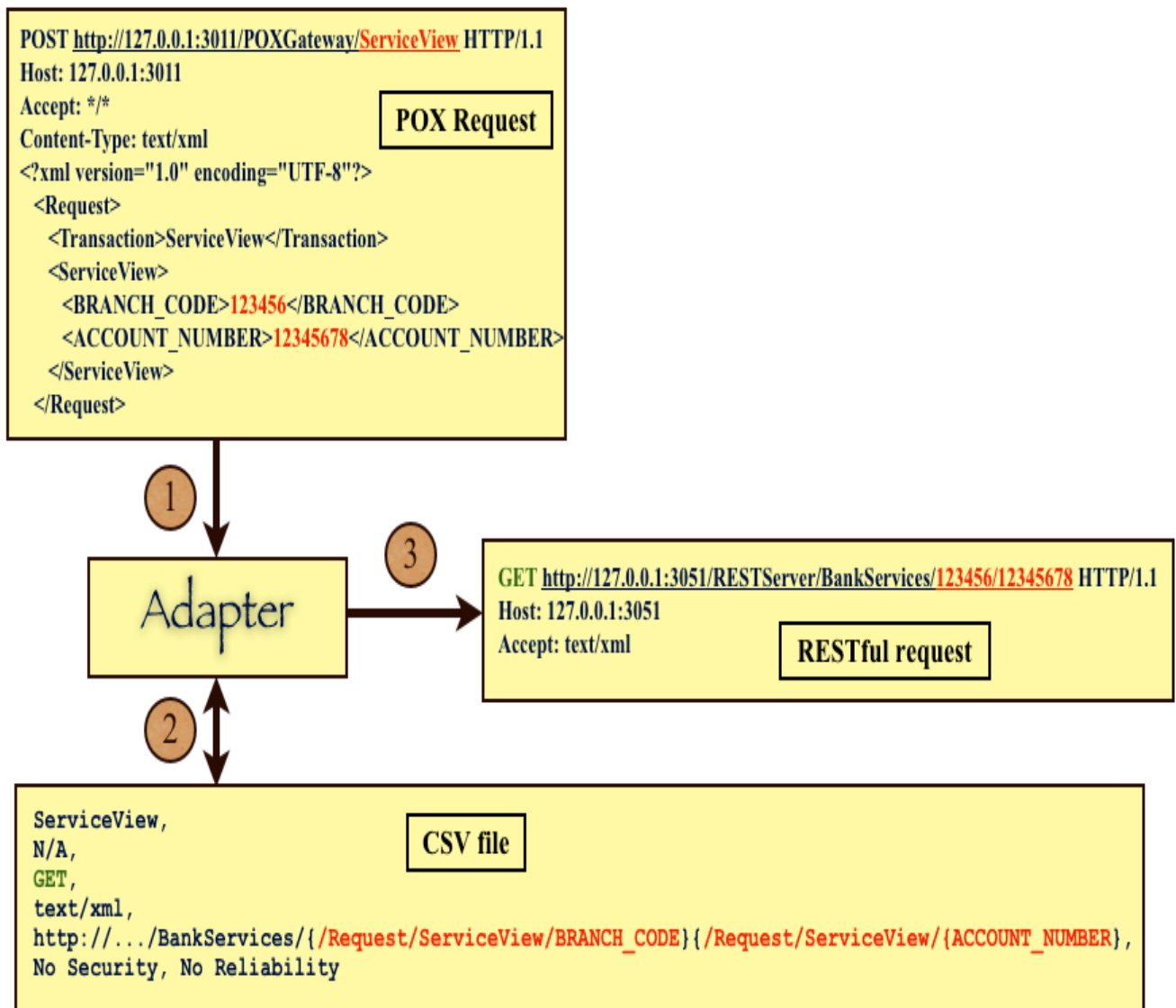


Figure 6-3 Adapter mapping a POX message to RESTful format

6.2.3 StoRHm v1 (SOAP) Configuration Wizard

The configuration function is supported by a wizard as shown in Fig. 6-4. The structure of the GUI is a front-end for the CSV with some support. WSDL informs the SOAP side of the mapping. Versions 1.1 and 2.0 of WSDL are supported. The wizard prompts the user for the names of the WSDL and schema files. The file names can be URI based or file based. As the target mapping is from SOAP over HTTP to RESTful HTTP, the wizard ensures that the Web service has a HTTP binding. The wizard obtains the Web service name and operations supported by parsing the WSDL file. The operations arguments are retrieved from the schema and are used to create signatures for the Specific Interface column (column 1 in Fig. 6-4). The user can build the URI using the “Build URI” window

(Fig. 6-5). In Fig. 6-5, the arguments of the WSDL operation are populated into the drop down listbox as an aid to the user in building the URI.

The user selects the QoS requirements for that RESTful HTTP service by selecting the relevant checkbox (3rd column in Fig. 6-4). The Uniform Interface and MIME type columns (5th and 6th columns respectively in Fig. 6-4) are populated from configuration files. The user then selects OK and the CSV file is populated.

The CSV file is subsequently used by the protocol adapter when transforming the SOAP request to RESTful HTTP format. As with the POX to RESTful HTTP configuration wizard, WADL is an optional feature. So rather than the “HTTP Verb” and “MIME Type” columns being populated from configuration files, the WADL checkbox can be selected and the WADL facility provided by NetbeansTM used to populate those columns.

Mapping

WSDL : ☒ v1.1 ☐ v2.0 ☐ URI ☒ WADL ?

Schema

WADL :

HTTP Web Service Name : Root URI :

Specific Interface	RESTful URI	Quality of Service			Uniform Interface	MIME Type
addPhoneNumber (phoneN...	/phoneDirectory	<input type="checkbox"/> Security	<input checked="" type="checkbox"/> Reliability	<input type="button" value="Build..."/>	POST	text/xml
deleteAllNumbers ()	/phoneDirectory	<input type="checkbox"/> Security	<input type="checkbox"/> Reliability	<input type="button" value="Build..."/>	DELETE	text/xml
getAllNumbers ()	/phoneDirectory	<input type="checkbox"/> Security	<input type="checkbox"/> Reliability	<input type="button" value="Build..."/>	GET	text/xml
deletePhoneNumber (phone...	/phoneDirectory/{phoneNo}	<input type="checkbox"/> Security	<input type="checkbox"/> Reliability	<input type="button" value="Build..."/>	DELETE	text/xml
getPhoneNumber (firstNam...	neDirectory/{surname}/{firstName}	<input type="checkbox"/> Security	<input type="checkbox"/> Reliability	<input type="button" value="Build..."/>	GET	text/xml
updatePhoneNumber (phon...	neDirectory/{surname}/{firstName}	<input checked="" type="checkbox"/> Security	<input type="checkbox"/> Reliability	<input type="button" value="Build..."/>	PUT	text/xml

Figure 6-4 StoRHm v1 Configuration Wizard



Figure 6-5 Building the URI Wizard

6.2.4 StoRHm v1 (SOAP) Protocol Adapter

The protocol adapter of StoRHm_v1 is implemented as a servlet and is deployed on the client. The adapter, which acts as both a server (to the SOAP client) and a client (to the RESTful WS), is setup to listen on port 8080 at URI “http://127.0.0.1/Gateway”. The “*location*” attribute of the “*service/port/address*” element of the client-side WSDL file is modified to re-direct SOAP service invocations via the adapter. There is no re-compilation of the SOAP client necessary. However, The Web Services Reference must be re-created for this WSDL change to take effect. The StoRHm_v1 client utilises the Netbeans infrastructure to call the Web Service. This means that the developer is abstracted from the client call and much of the code generation is automated. The client executes a SOAP request as normal and this request arrives at the adapter. The adapter parses the Web service name from the incoming URI. The Web Service name is then used to index the CSV file. The adapter, using the other entries for that row in the CSV file, maps the SOAP request to a RESTful HTTP request. This may involve the extracting of data from the SOAP request and inserting that data into the RESTful URI.

This process can be seen from Fig. 6-6:

- (1) - The incoming URI (from the SOAP client) is pointing at the adapter, namely “Gateway”. The Web Service “PartsInventory” is appended to the URI.
- (2) - The adapter parses “PartsInventory” from the URI and the SOAP operation “getPartNumber” from the SOAP request. The adapter then uses both the Web Service name and SOAP operation name to index the CSV file. The CSV file informs the adapter that the resultant HTTP request is a

GET with a response type of “text/xml”. There are no QoS features with this Web Service i.e. no Security or Reliability.

(3) - Lastly, the CSV provides a “RESTful URI” column that enables the adapter to construct the URI for the HTTP message. This column contains the XPath expressions that are used on the SOAP request to elicit the values to use in the subsequent HTTP message. For example, “{/partNo}” is an XPath expression that when applied to the original SOAP request results in the value “200” being extracted. This value is inserted into the URI in the XPath expression’s location. Thus the *partNo* of 200 is extracted from the SOAP request and inserted into the RESTful URI request.

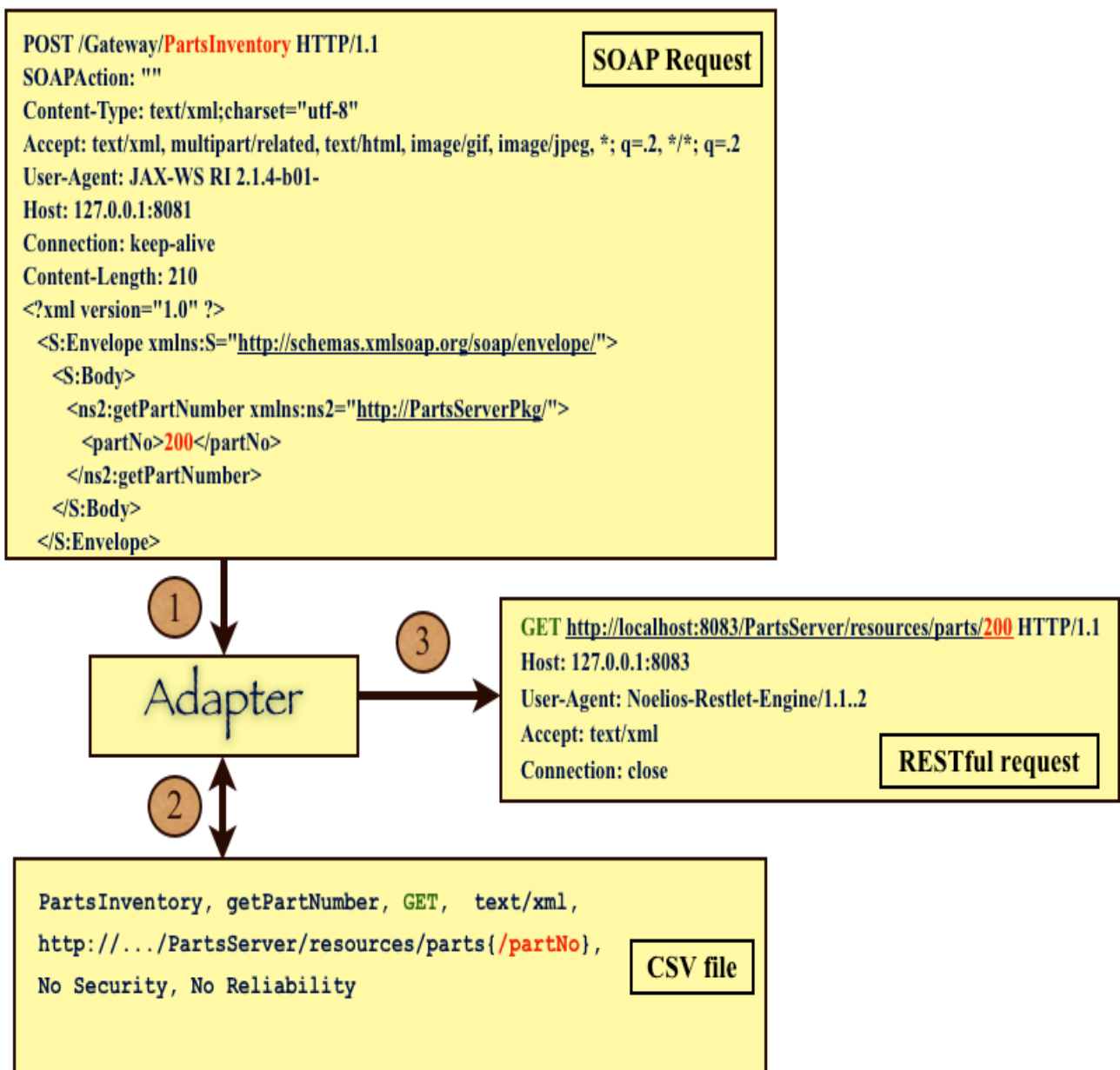


Figure 6-6 Adapter mapping a SOAP message to RESTful format

6.4 StoRHm v2 implementation

As with StoRHm v1, there are configuration and protocol adapter elements. However, the implementation of both in StoRHm v2 is quite different. The core requirement differentiating StoRHm v2 is the integration of the Semantic Web in order to automate the configuration element. Thus, the configuration wizard of StoRHm v2 reflects Semantic Web intelligence. With regards to the protocol adapter, StoRHm v2 integrates both SOAP and POX into one instance whereas the protocol adapter of StoRHm v1 resulted in two separate instances (one for SOAP and one for POX).

6.4.1 Configuration Wizard

A desktop-based wizard (Fig. 6-7) supports the configuration function. The structure of the GUI is a front-end for the CSV with some support. The wizard provides two radio buttons at the top, named POX and SOAP respectively. Depending on which the user selects, the user will be prompted for a Schema (POX) or a SAWSDL file (SOAP). The MicroWSMO and ontology files are common to both SOAP and POX. The file names can be URI based or file based.

The Semantic Web is a promising approach for Web Service selection [106] because RDF triples can link concepts defined in different vocabularies thereby establishing relationships between vocabularies [104]. As a result, the SAWSDL file, representing the SOAP/POX Web Service; the MicroWSMO file, representing the RESTful Web service and the ontology, stating the QoS supported by the Web Service are all semantically annotated a priori by Core dashboard [81], SWEET [13] and Protege [98] respectively.

The wizard obtains the model references of the operations supported and their arguments, by parsing the SAWSDL file. The model references represent the semantic concepts. For example, Listing 6-2 is a sample SAWSDL segment outlining a SOAP Banking Web Service:

```
30 <xsd:complexType name="ServiceView">
31 <xsd:sequence>
32 <xsd:element minOccurs="0" name="branchCode"
33 sawsdl:modelReference="http://www.leitrimills.ie/ontologies/BankService#BranchCode"
34 type="xsd:string"/>
35 <xsd:element minOccurs="0" name="accountNo"
36 sawsdl:modelReference="http://www.leitrimills.ie/ontologies/BankService#AccountNo"
37 type="xsd:string"/>
38 </xsd:sequence>
39 </xsd:complexType>
40 </xsd:schema>
41 </types>
42 <message name="ServiceView">
43 <part element="tns:ServiceView" name="parameters"/>
44 </message>
45
46 <portType name="BankService">
47 <operation name="ServiceView">
48 <input message="tns:ServiceView"/>
49 <output message="tns:ServiceViewResponse"/>
50 <sawsdl:attrExtensions sawsdl:modelReference="http://www.leitrimills.ie/ontologies/BankService#ViewService"/>
51 </operation>
52 </portType>
```

Listing 6-2 SAWSDL example (segment)

Note the semantic annotations of the operation itself :

"<http://www.leitrimmills.ie/ontologies/BankService#ViewService>" (line 50) and its input parameters:
"<http://www.leitrimmills.ie/ontologies/BankService#BranchCode>" (lines 33) and
"<http://www.leitrimmills.ie/ontologies/BankService#AccountNo>" (lines 36). Listing 6-3 shows the MicroWSMO (segment) equivalent of Listing 6-2:

```
3  <msm:Service rdf:ID="BankService">
4    <rdfs:isDefinedBy rdf:resource=""/>
5    <rdfs:label>Bank Service API</rdfs:label>
6    <msm:hasOperation>
7      <msm:Operation rdf:ID="ViewOperation">
8        <rdfs:label>ServiceView</rdfs:label>
9        <hr:hasMethod>GET</hr:hasMethod>
10       <hr:hasAddress rdf:datatype="http://www.wsmo.org/ns/hrests#URITemplate">
11         http://127.0.0.1:3051/RESTServer/BankServices/{branchcode}/{accountnumber}
12       </hr:hasAddress>
13       <sawSDL:modelReference rdf:resource="http://www.leitrimmills.ie/ontologies/BankService#ViewService"/>
14       <msm:hasInput>
15         <msm:MessageContent rdf:ID="ViewOperationInput">
16           <msm:hasOptionalPart>
17             <msm:MessagePart rdf:ID="branchcode">
18               <sawSDL:modelReference rdf:resource="http://www.leitrimmills.ie/ontologies/BankService#BranchCode"/>
19             </msm:MessagePart>
20           </msm:hasOptionalPart>
21           <msm:hasOptionalPart>
22             <msm:MessagePart rdf:ID="accountnumber">
23               <sawSDL:modelReference rdf:resource="http://www.leitrimmills.ie/ontologies/BankService#AccountNo"/>
24             </msm:MessagePart>
25           </msm:hasOptionalPart>
26         </msm:MessageContent>
27       </msm:hasInput>
28     </msm:Operation>
```

Listing 6-3 MicroWSMO example (segment)

Note that the semantic concept for the operation is the same (line 13) as in the SAWSDL file (Listing 6-2). The semantic annotations match for the inputs to the service also (lines 18 and 23). These matching semantic concepts (i.e. same URIRef's) are located in the MicroWSMO file in order to ascertain the URI (line 12 of Listing 6-3) and HTTP verb (line 9 of Listing 6-3) used by the equivalent RESTful Web Service.

The ontology is used to specify the QoS required and consequently, the service is located in the ontology to determine whether Security and/or Reliability are required. Listing 6-4 is the ontology for the service described in Listing 6-2 and 6-3. The *ViewService* concept is outlined between lines 73-77. The URIRef "<http://www.leitrimmills.ie/ontologies/BankService#ViewService>" (line 73) denotes the semantic concept. This is the same semantic concept identified on line 50 of Listing 6-2 (SAWSDL file) and on line 13 of Listing 6-3 (MicroWSMO file). The ontology states on line 75-76 that this service does not support Security or Reliability. The branch code and account number concepts of the ontology are outlined on lines 81 and 83 of Listing 6-4 respectively. The full URIRef for the branch code is "<http://www.leitrimmills.ie/ontologies/BankService#branchCode>" (line 79).

This matches with line 33 of Listing 6-2 (SAWSDL) and line 18 of Listing 6-3 (MicroWSMO). The URIRef for the account number is “*http://www.leitrimmills.ie/ontologies/BankService#accountNo*”. This matches with line 36 of Listing 6-2 and line 23 of Listing 6-3. Once the QoS for the service has been ascertained, the CSV file can be populated for that service.

```
70
71      <!-- http://www.leitrimmills.ie/ontologies/BankService#ViewService -->
72
73      <BankService rdf:about="#ViewService">
74          <rdf:type rdf:resource="#owl:Thing"/>
75          <hasReliability rdf:datatype="&xsd:boolean">false</hasReliability>
76          <hasSecurity rdf:datatype="&xsd:boolean">false</hasSecurity>
77      </BankService>
78
79      <!-- http://www.leitrimmills.ie/ontologies/BankService#accountNo -->
80
81      <owl:Thing rdf:about="#accountNo">
82          <rdf:type rdf:resource="#BankService"/>
83      </owl:Thing>
84
85      <!-- http://www.leitrimmills.ie/ontologies/BankService#branchCode -->
86
87      <owl:Thing rdf:about="#branchCode">
88          <rdf:type rdf:resource="#BankService"/>
89      </owl:Thing>
90  </rdf:RDF>
```

Listing 6-4 Ontology example (segment)

The CSV file is then populated as follows: Web Service Name and SOAP Operation from the SAWSDL/Schema file; RESTful URI and HTTP verb from the MicroWSMO file and QoS from the ontology. However, if any of the SAWSDL/Schema concepts are not located in either the MicroWSMO and/or ontology files, an error is reported to the user and the configuration process exits without creating the CSV file.

Lifting and lowering schemas are used to mediate data between services but as we are completely replacing the SOAP request with a RESTful request, there is no need for mediation and thus no need to use lifting and lowering schemas.

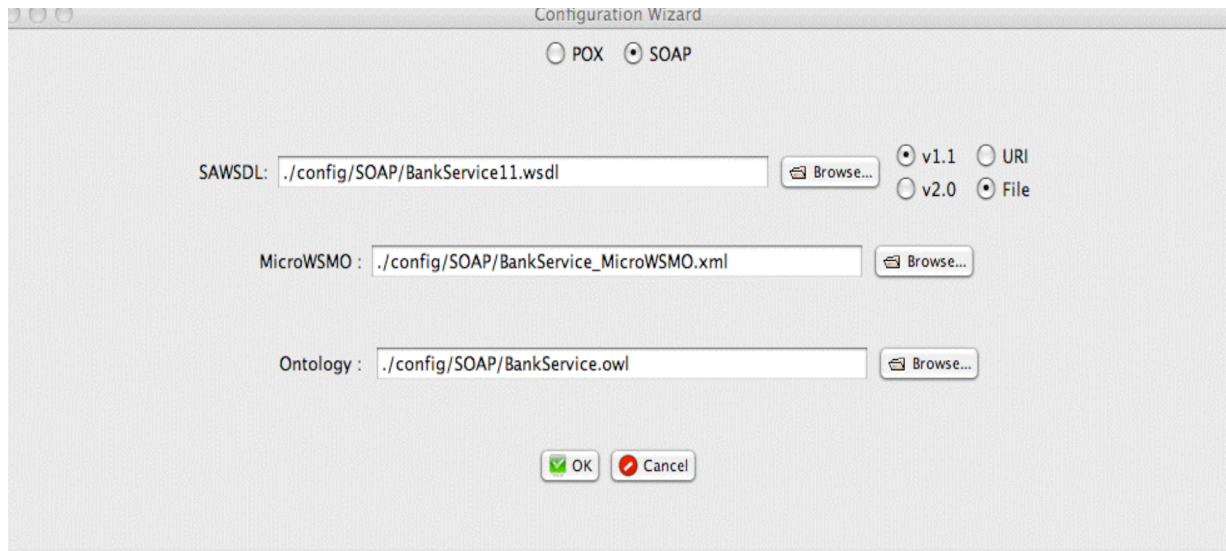


Figure 6-7 (a) StoRHm v2 Configuration Wizard - SOAP

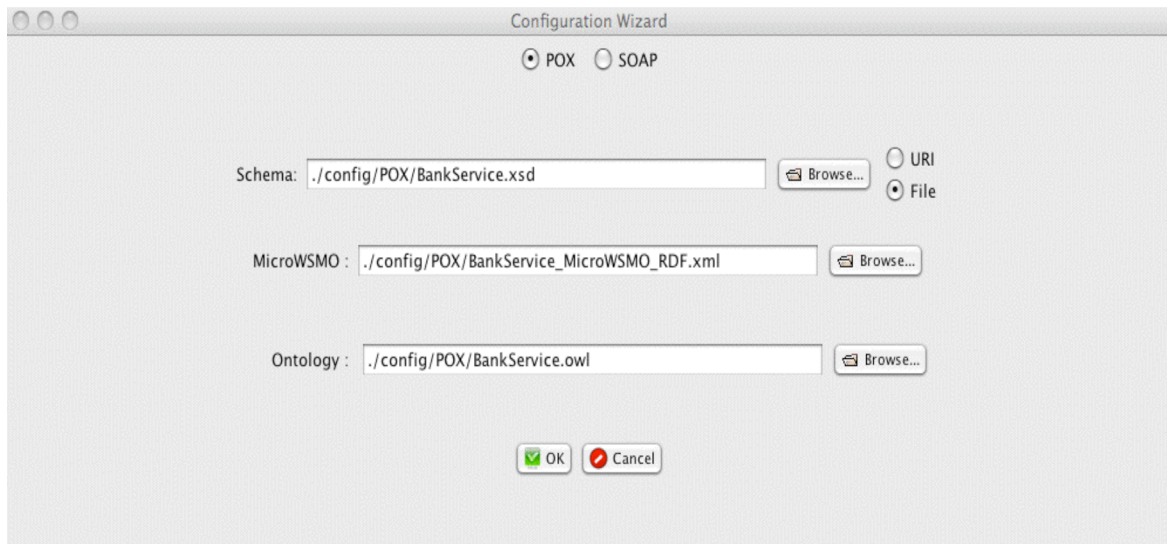


Figure 6-7 (b) StoRHm v2 Configuration Wizard - POX

Fig. 6-7 (a) shows the configuration wizard based on the SOAP radio button being selected (note the SAWSDL input prompt). Fig. 6-7 (b) shows the configuration wizard based on the POX radio button being selected (note the Schema input prompt).

6.4.2 Protocol Adapter

The StoRHm_v2 adapter is an integrated adapter based on the RESTlet framework. As the StoRHm_v1 adapter is servlet based and the POX to RESTful HTTP adapter is RESTlet based, change was unavoidable. A decision was made to proceed with the RESTlet approach. The reasons for this were as follows:

- a) Familiarity with the RESTlet pattern was limited when developing the original StoRHm_v1

adapter (which was the very first adapter)

- b) RESTlet provides excellent granularity of control within the code over the URL patterns i.e. there is no need for editing of an external web.xml file as with the servlet pattern
- c) After changing the WSDL file to point the SOAP client at the adapter, in order to get the client to “dynamically” read the WSDL file, it was necessary to delete and re-create the Web Service Reference in Netbeans. This is not the case when the client is implemented using RESTlet.

This has little effect on migrating the POX adapter (of StoRHm v1) as it was already based on RESTlet. Changes were required when integrating the SOAP adapter (of StoRHm_v1), as it was servlet based.

As the StoRHm_v2 adapter is an integrated solution, it can handle both SOAP and POX requests. It achieves this by setting up two separate URI's; one for POX requests (“/POXAdapter”) and one for SOAP requests (“/StoRHm_Adapter”). Both URI's are on port 3010. There are two separate Java classes responding to these URI's. Their operation follows the pattern outlined previously:

- extract the Web Service name from the URI.
- use the Web Service name (and SOAP operation name if a SOAP request) to index the CSV file.
- the adapter now has the HTTP verb to use; the RESTful URI; the content type to request and the QoS required.
- if the URI retrieved from the CSV file is for a top-level collection i.e. there are no placeholders in the URI, then use that URI in the subsequent RESTful HTTP request. If there are placeholders in the CSV URI, the request is parsed to elicit these values and the data is inserted into the URI in the locations of the placeholders.

Once the HTTP request is built, the adapter issues the request. The RESTful Web Service is executed and returns its response to the adapter. The adapter returns this response to the original client. If the original client was a SOAP client then the adapter wraps the response message in SOAP metadata before returning it.

6.5 Testing

In this section, the test environment is outlined and sample messages using the adapter are shown. The performance of the adapter is also analysed. The sample messages are locally based tests (over the LAN) but the performance tests are based on a distributed environment (over a LAN initially and

subsequently over the Internet).

6.5.1 Test Environment

A test environment has been implemented both in a controlled LAN environment and in an Internet based environment. The client and server machines in the LAN environment are Dell Optiplex 780 desktop machines running Windows 7 with 4GB RAM and Intel Core2 Duo CPU processors. One of the Dell machines hosts the client and adapter software while the other machine hosts the server software. Glassfish is the application server used by the server. In the Internet environment, the client is as outlined above and the server is a Dell Dimension 8400 running Windows XP with 3GB RAM. The server machine is connected to the Internet via a 3Mb/sec Asymmetric Digital Subscriber Line (ADSL) line. Figure 6-8 (a) and (b) reflect both the LAN and Internet-based setup respectively.

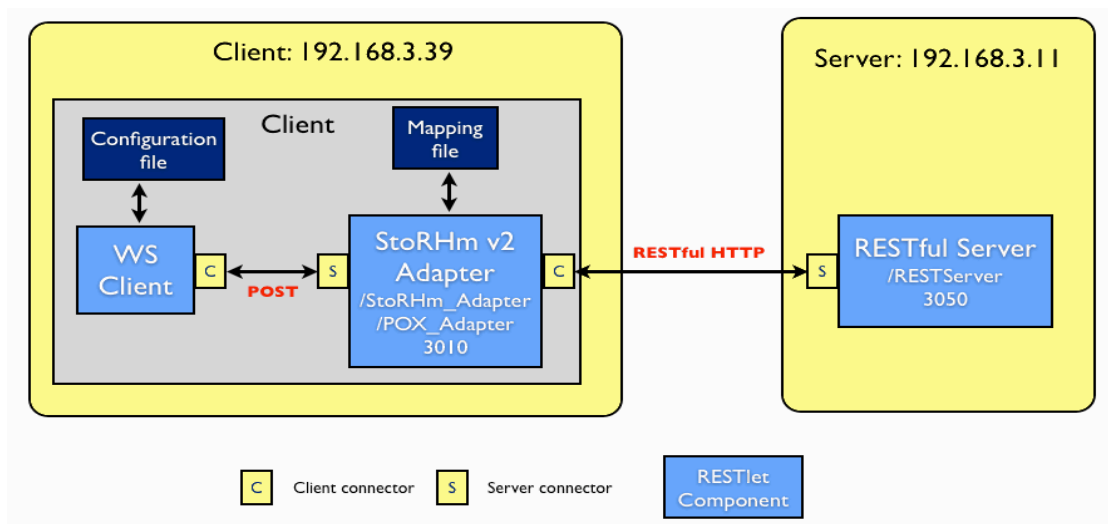


Figure 6-8 (a) LAN-based Test environment

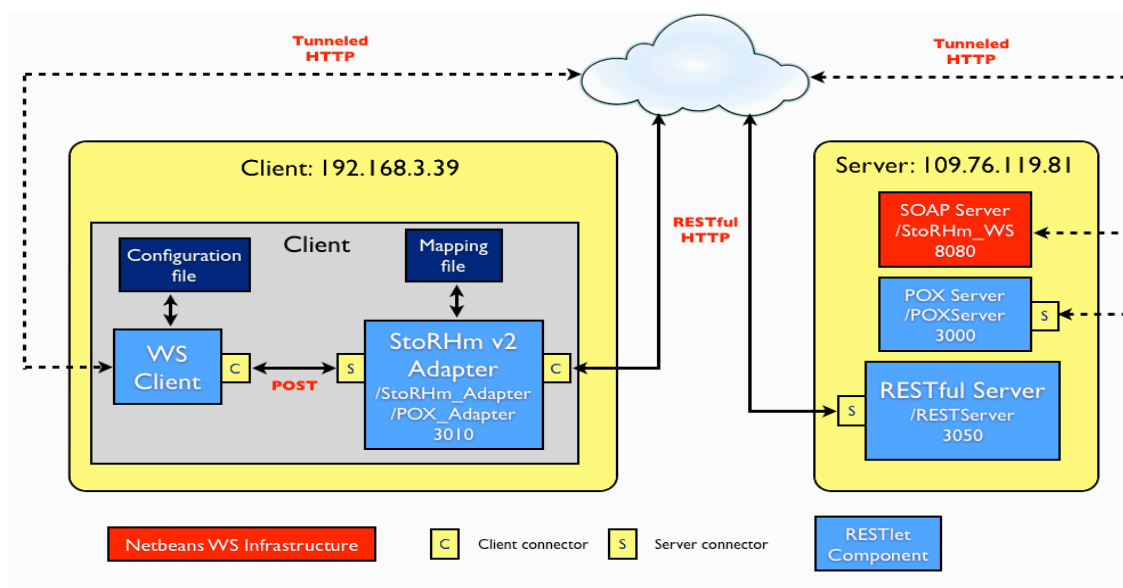


Figure 6-8 (b) Internet-based Test environment

The IDE is NetBeans v6.8 which has integrated JAX-WS and Jersey 1.0 (JAX-RS reference implementation) libraries. The configuration wizards use the “?wsdl” and “application.wadl” facilities provided by Netbeans. The RESTlet framework was used by the adapters for responding to SOAP/POX requests and for issuing subsequent RESTful HTTP messages to the backend RESTful server. The RESTful server was also developed using RESTlet. An HTTP monitor, TCPMon [99] was inserted between the Web service client, the adapter and the Web services server so as to monitor the messages on the network. Subsections 6.5.2 and 6.5.3 detail POX and SOAP examples respectively. Each of the examples work with both versions of StoRHm.

6.5.2 POX example

The POX client is developed normally i.e. to communicate with a POX web service. The POX web service in this instance is called “*ServiceView*” and it’s equivalent RESTful Web Service (*GET /RESTServer/BankServices/{branchCode}/{accountNo}*) was also developed. The wizard in Figure 6-1 was executed and the CSV file created. The client-side configuration file, namely *StoRHm_Client.xml*, which the client dynamically reads, was modified to point to the protocol adapter. At this point, the RESTful WS server and the protocol adapter are up and running and the client configuration file has been modified.

The client executes its POX request, which is directed to the adapter. Using the CSV file the adapter transforms the POX message into a RESTful HTTP request. If the URI in the CSV file has placeholders for POX elements then those elements are extracted from the POX message and are inserted into the URI. The RESTful HTTP request is then issued to the updated URI. Once the RESTful HTTP request returns, the reply is returned to the POX client.

Figure 6-9 shows the client POX request being mapped to RESTful HTTP. Note that the POST verb has now been mapped to a GET and thus if the message is routed through a cache, the cache can now inspect the message and check if it has an up-to-date representation of the resource.

Figure 6-9 also shows the HTTP *Accept* header which informs the RESTful service of the content type the client understands (*text/xml* in the example). This is referred to as “*Content Negotiation*” and enables RESTful Web Services greater flexibility with regard to the clients they can interact with than their POX counterparts i.e. the RESTful services can serve other representation types e.g. JSON. However, as the client is POX, the response type will always be XML.

The *Accept* header has changed from the POX request to the RESTful request because the user selected *text/xml* specifically when creating the mapping. However, had the user selected “*Use POX*

Header” from the MIME Type drop down combo box during the configuration stage, the *Accept* header on the RESTful request would be identical to the original POX request.

The “*Content-Type*” header on the POX request is not applicable in the RESTful HTTP request because this is a GET request i.e. there is no entity body and therefore no “content”.

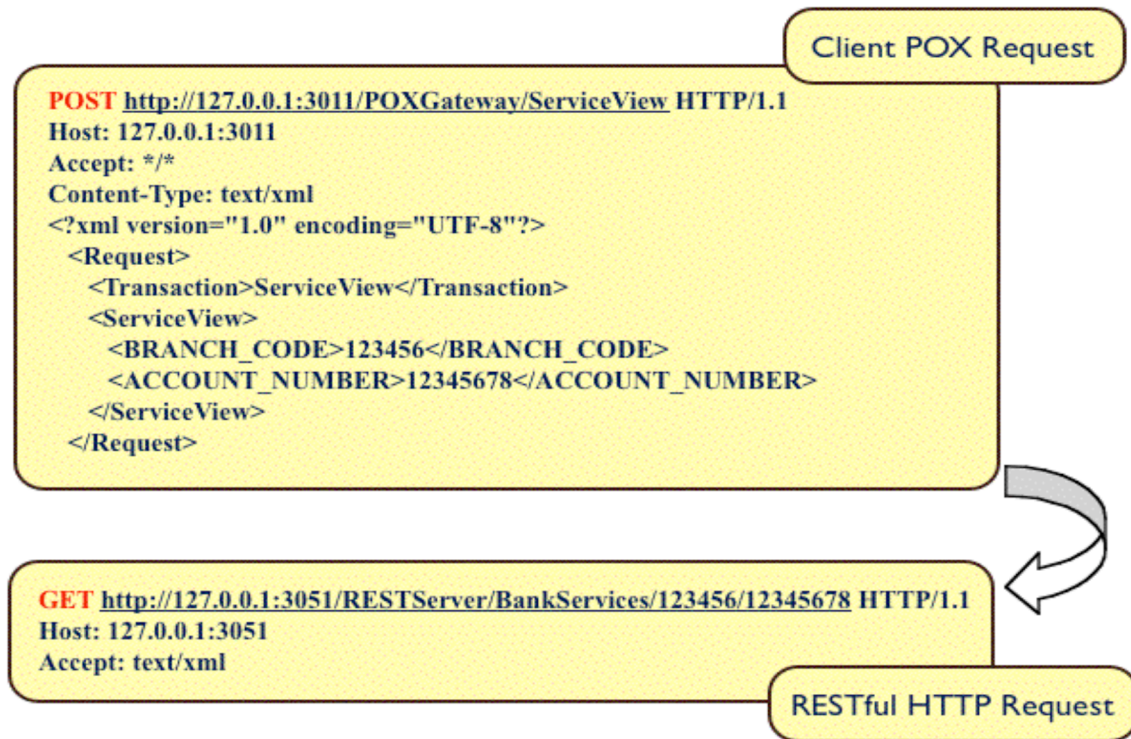


Figure 6-9 POX to RESTful HTTP Transformation

Figure 6-10 shows the RESTful Web Service response, which passes through the adapter where it is returned to the POX client.

RESTful Response

HTTP/1.1 200 The request has succeeded
Content-Type: text/xml
Server: Noelios-Restlet-Engine/1.1..2

```
<?xml version="1.0" encoding="UTF-8"?>
<AccountDetails>
  <BranchCode>
    <Value>123456</Value>
    <link href="http://127.0.0.1:3051/RESTServer/BankServices/123456/12345678" rel="src"/>
    <link href="http://127.0.0.1:3051/RESTServer/BankServices/123456" rel="target"/>
  </BranchCode>
  <AccountNumber>
    <Value>12345678</Value>
    <link href="http://127.0.0.1:3051/RESTServer/BankServices/123456/12345678" rel="src"/>
    <link href="http://127.0.0.1:3051/RESTServer/BankServices/123456/12345678" rel="target"/>
  </AccountNumber>
  <CustName>Sean Kennedy</CustName>
  <CustAddress>Athlone</CustAddress>
  <CustType>Personal</CustType>
  <CustRating>3</CustRating>
  <Balance>100</Balance>
</AccountDetails>
```

Figure 6-10 RESTful response

6.5.3 SOAP example

The SOAP client is developed normally i.e. to communicate with a SOAP web service. The SOAP web service is called “*PartsInventory*” and it supports several operations one of which is *getPartNumber*. An equivalent RESTful Web Service was developed. The wizard in Fig. 6-4 was executed and the CSV file created. The web service WSDL file (which the client dynamically reads) was modified to point to the protocol adapter. At this point, the RESTful WS server and the protocol adapter are up and running and the client WSDL file has been modified.

The client executes its SOAP request which is directed to the adapter. Using the CSV file the adapter transforms the SOAP message into a RESTful HTTP request. If the URI in the CSV file has placeholders for SOAP elements then those elements are extracted from the SOAP message and are inserted into the URI. The RESTful HTTP request is then issued to the updated URI. Once the RESTful HTTP request returns, the reply is wrapped in SOAP metadata extracted from the original SOAP request and returned to the SOAP client. Fig. 6-11 shows the client SOAP request being mapped to RESTful HTTP.

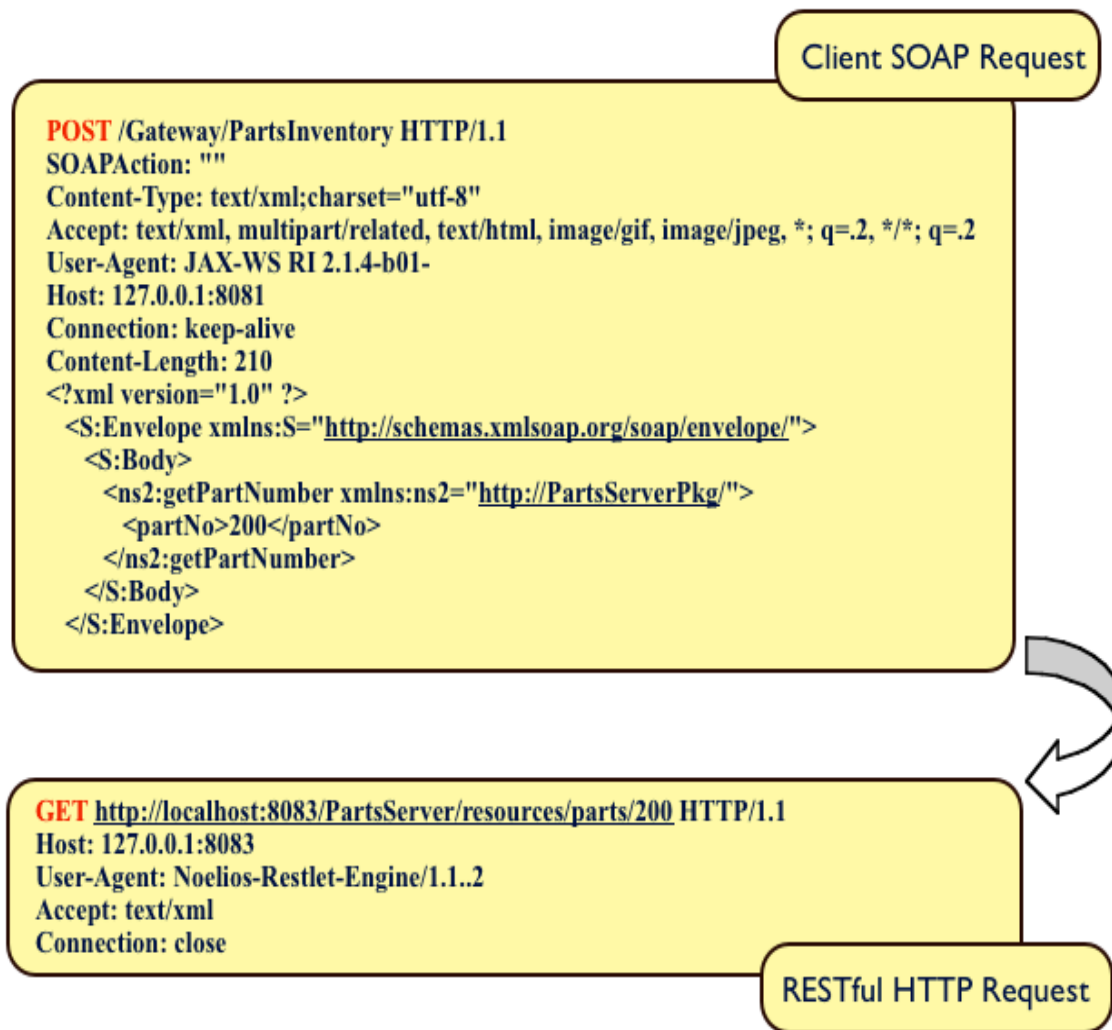


Figure 6-11 SOAP to RESTful HTTP Transformation

Fig. 6-12 shows the RESTful Web Service response being wrapped with SOAP elements. The RESTful Web Service response from the server passes through the adapter where it is wrapped in SOAP and returned to the SOAP client.

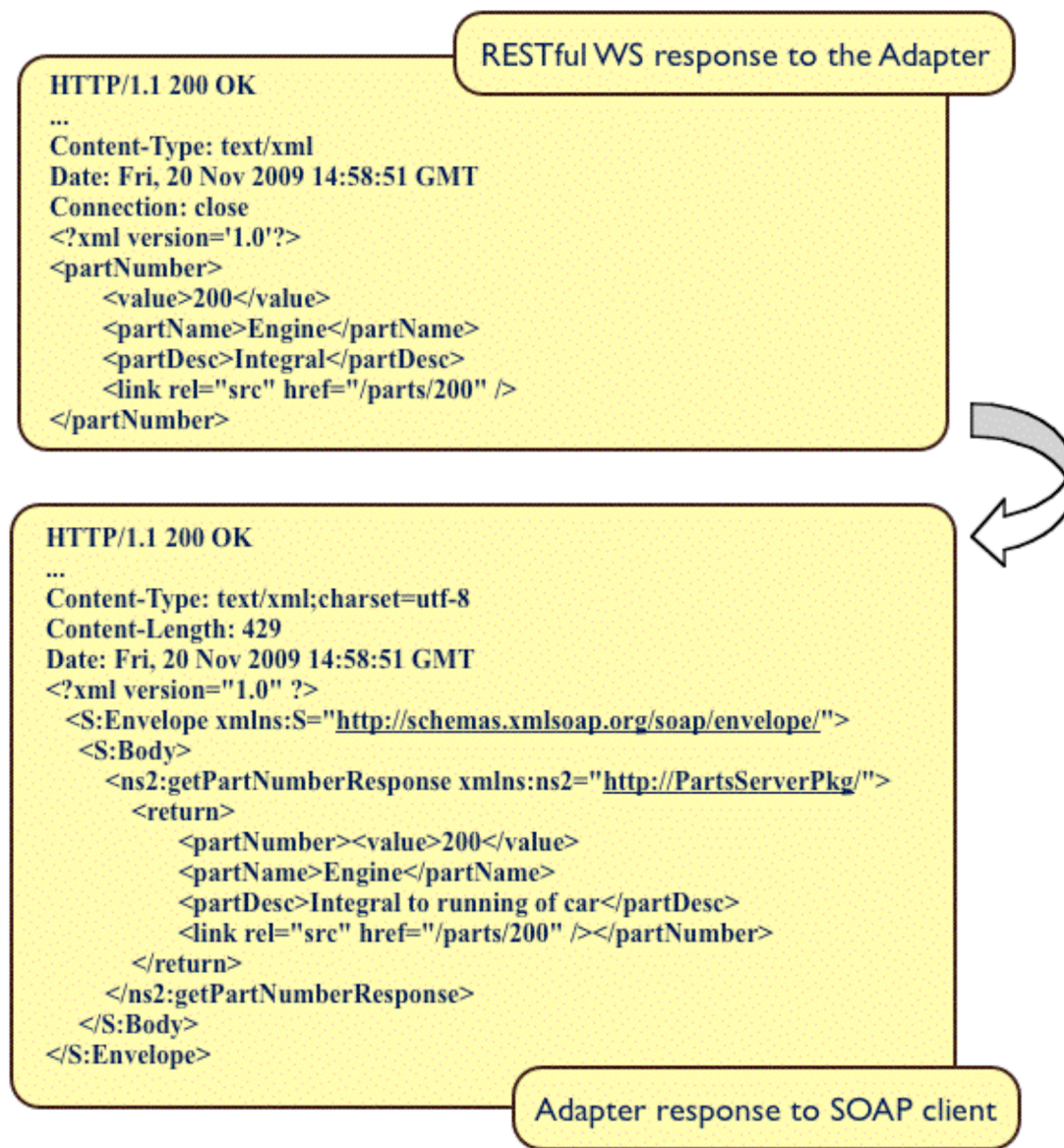


Figure 6-12 RESTful HTTP to SOAP Transformation

6.6 Evaluation

Our hypothesis was that XML based Web Services could be represented RESTfully when using HTTP. This was especially important for retrieval-type Web Services as they would map to HTTP GET and thus Web intermediaries such as caches would be enabled. We also stated our intention to build a seamless migration tool to facilitate enterprises wishing to migrate from POX Web Services to RESTful HTTP Web Services. Therefore StoRHm is evaluated based on the following criteria:

- Configuration Wizard
 - Operations that affect one resource and operations that affect more than one resource
 - URI length restrictions
- Protocol Adapter

- Performance implications
- Message visibility
- Ease of Migration

6.6.1 Configuration Wizards

This subsection relates to the following areas: an overview of the support in the wizards; the XML operations supported (SOAP or POX) and a discussion on the limits imposed on a URI.

6.6.1.1 XML operations supported

Given the interface-specific nature of XML operations, the operations are categorized into the following areas: Create, Read, Update and Delete (CRUD) requests. These do not equate to POST, GET, PUT and DELETE respectively, as PUT can also create and POST can append [45]. The XML operations are categorized as follows:

XML operations that affect one resource:

- Read: Support is available to insert all simple data types from the WSDL/SAWSDL/POX schema into the URI and as complex types are a hierarchy of simple types, these simple type values must make their way into the URI. If a SOAP client, the XML response returned from the RESTful Web Service is wrapped in a SOAP envelope so that the SOAP client will understand it.
- Delete: Same as Read above.
- Create: Support exists to map the XML payload into the URI. However, there is a difference when forwarding the request. The operation to be executed is located inside the XML request message. This is not needed in a RESTful implementation where the URI, coupled with the verb, will identify the Web Service to be executed. Currently the implementation passes on the XML payload as is. Thus we have imposed a constraint that the RESTful Web Service is developed based on existing XML payloads. In order to remove the unnecessary XML Web Service operation (outer element in SOAP messages) before passing on the XML payload, an XSLT transformation will be required (see Future Work section). As with Read/Delete above, if a SOAP client, the RESTful response is wrapped in SOAP metadata.
- Update: Same as Create above.

XML operations that affect more than one resource:

To improve efficiency in distributed systems, coarse-grained operations are preferred to fine grained operations. This pattern is known as the “Remote Facade” pattern [100]. Thus, instead of the client making two or more fine-grained operation calls returning many data items, the client makes one coarse-grained operation call which returns all the data at once. For example, if a SOAP bookstore operation requested information about a *user* and a *book* at the same time, both identifiers would be passed in the request. This would map to two GET requests on two different URI’s (the *book* resource and the *user* resource). The returned data from the fine-grained resources would need to be integrated into a coarse grained response. To handle complex requests that map logically to more than one URI, we propose to leave the POST in place to carry the data but RESTify the URI i.e. we constrain the RESTful WS to implement the logic as in the SOAP model. This means that the data is passed on untouched via POST to a RESTful Web Service URI. This usage of POST is often “*the easiest way to express complex operations that span multiple resources*” [45].

6.6.1.2 URI Size limit

The HTTP specification does not put a limit on URI length but servers do e.g. Apache imposes an 8Kb limit [45]. As GET and DELETE have no entity bodies to carry data, the arguments to be passed are encoded in the URI. If a requirement arises where a large amount of data must be passed, the URI (for a GET or DELETE) may become too long and lead to malformed URIs [101]. In this scenario, it is suggested to leave the POST in place to carry the data but RESTify the URI. This is a non-controversial use of POST [45]. Note however, that a 9-month study was conducted in 2004 to gather statistics on publicly available Web Services [130]. It was noted that 92% of SOAP messages were smaller than 2KB [130].

However, we believe that the URI is more efficient in representing data than XML, simply because the URI contains no XML metadata i.e. no XML start and end tags involved. Even if one chooses to use HTTP parameters (which roughly involves the same number of characters as an XML start tag), there is still the gain of not having any XML end tags. For example, to encapsulate a person’s name in XML requires the start tag `<name>` followed by the end tag `</name>`. In a parameterized URI this would be encoded as `?name=`. Thus, there is the saving on the XML end tag (`</name>` in this example).

In the Introduction chapter, the goal of seamlessly enabling SOAP client’s access to RESTful WS was outlined. The implication here is that the server side has already been migrated i.e. the SOAP WS and their RESTful equivalents already exist and therefore the issue of the URI size limit will already have been addressed. Thus, whether the mapping is from a specific SOAP operation to GET,

where the URI is capable of accommodating the data; or to POST, where the URI is unable to accommodate the data, will already have been decided. The objective of the configuration wizards presented in this thesis, was of addressing XML client-side migration. The URI size limit issue is of direct relevance to our Future Work where the server-side migration from SOAP WS to their RESTful counterparts will be researched.

6.6.2 Protocol Adapter

While performance is not central to the research in this thesis, it is an important consideration for distributed technologies. Therefore, to measure the time impact of the adapter was necessary. One of the standard measurements taken is the “*round-trip method invocation time*” which is the time difference between the client invoking the remote method and the remote method returning its results [102].

In this section, we present performance tests comparing the round-trip method invocation times (in msecs) of a normal XML Web Service request with that of the same Web Service request routed via the adapter to the equivalent RESTful Web Service. The StoRHm v2 adapter is the adapter used here as it is the most recent and is an integrated solution. Note that the adapter-based figures represent a worst-case scenario i.e. a full round trip on each occasion with no efficiencies such as caching implemented. Two test environments were used: initially, a 1Gbit/sec LAN and subsequently the Internet. The LAN based tests are POX based whereas the Internet based tests are both SOAP and POX based i.e. the Internet based tests are a superset of the LAN based tests. The subsequent statistics are informed by the Internet results.

In addition, the requirement that the transformed messages are visible (and therefore RESTful) is also evaluated.

6.6.2.1 Software artefacts

The following software artefacts were designed and developed:

Client: A generic client, namely *StoRHm_Client*, was developed. *StoRHm_Client* is dynamically informed by a configuration file, namely *StoRHm_Client.xml*. This configuration file enables the dynamic re-configuration of the client test data e.g. what type of request (SOAP or POX) to generate and the number of tests to perform. Listing 6-5 shows the *StoRHm_Client.xml* set to generate 100 POX requests that will be routed via the adapter:


```

<?xml version="1.0" encoding="UTF-8"?>
<Client>
  <performanceTests viaAdapter="yes" wsurl="http://109.76.119.81:3000/POXServer/ServiceView" />
  <type value="POX" numTests="100" outputToScreen="no" />
  <SOAP>
    <wsdl path="./config/SOAP/BankServiceService.wsdl" />
    <request path="./config/SOAP/SOAPRequestServiceView_DOM1637.txt" />
  </SOAP>
  <POX>
    <adapter url="http://127.0.0.1:3010/POX_Adapter/ServiceViewDetailedParse" />
    <request path="./config/POX/POXRequestServiceViewDetailedParse.txt" />
  </POX>
</Client>

```

Listing 6-5 StoRHm_Client.xml

RESTful Server: A RESTlet-based backend RESTful Web Server, namely *RESTful_WS_Server*, responding to two URI's on port 3050 was developed. The URI's are:

- *"/RESTServer/BankServices/"* and
- *"/RESTServer/BankServices/{branchCode}/{accountNo}"*

The two URI's are handled by two different Java classes. The *"/{branchCode}/{accountNo}"* are placeholders for actual values passed in the URI. For example, if the URI

"/RESTServer/BankServices/930010/12344321" is used in a HTTP request (to port 3050) then the Java class assigned to that URI is executed. The *branchCode* variable takes on the value 930010 and the *accountNo* variable takes on the value 12344321. The method executed in the class depends on the HTTP verb used in the request:

- POST results in *"acceptRepresentation"* method being executed
- GET results in *"represent"* method being executed
- DELETE results in *"removeRepresentations"* method being executed
- PUT results in *"storeRepresentation"* method being executed

POX Server: A RESTlet-based backend POX Web Server, namely *POXServer*, responding to one URI *"/POXServer"* on port 3000 was developed. This server overrides the *acceptRepresentation* method i.e. it handles POST requests. The server parses the XML document using non-standard XPath expressions. The server returns a standard XML response document to the client.

SOAP Server: A JAX-WS based SOAP Web Server, namely *StoRHm_Web_Services*, responding to one URI *"/StoRHm_Web_Services"* on port 8080 was developed. This server utilises the Netbeans and JAX-WS annotations e.g. *@WebService* before a class to specify that this class is a Web Service

and *@WebMethod* before a method to specify that this is an operation hosted by the Web Service. The server returns a standard XML response document to the client.

6.6.2.2 LAN performance – StoRHm v2 adapter

The XML Web Services used were called *ServiceView* (Listing 6-6a) and *ServiceAdd* (Listing 6-6b). Both are POX requests and the files sizes are identical (except for the Web Service name). When routed via the adapter, these requests translated into HTTP GET and POST requests respectively. In order to isolate the adapter, backend database activity was removed and identical test XML data (Listing 6-6c) returned.

```
<?xml version="1.0" encoding="UTF-8"?>
<Request>
  <Transaction>ServiceView</Transaction>
  <ServiceView>
    <BRANCH_CODE>123456</BRANCH_CODE>
    <ACCOUNT_NUMBER>12345678</ACCOUNT_NUMBER>
  </ServiceView>
</Request>
```

Listing 6-6 (a) POX ServiceView.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Request>
  <Transaction>ServiceAdd</Transaction>
  <ServiceAdd>
    <BRANCH_CODE>123455</BRANCH_CODE>
    <ACCOUNT_NUMBER>12345678</ACCOUNT_NUMBER>
  </ServiceAdd>
</Request>
```

Listing 6-6 (b) POX ServiceAdd.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<DummyResponse>
  <return>nothing</return>
</DummyResponse>
```

Listing 6-6 (c) Test data returned

LAN testing indicates that the adapter imposes a 48% performance penalty when mapping to a GET and a 96% penalty when leaving the POST in place. To better understand these figures, three suites of tests were carried out:

- a) the difference between HTTP GET and POST.
- b) examples where the adapter replaces the client POST request with a GET
- c) examples where the adapter leaves the POST in place

Examining these in turn:

a) HTTP GET versus POST

As stated previously, there is no database activity on the server and both responses contain identical test data. The difference is that the POX client POSTs an XML document and the RESTful client GETs an equivalent URI i.e. there is no XML entity body in a RESTful GET.

POST – Using the POX Web Service *ServiceView*, 100 tests were carried out in the environment outlined in section 6.5.1. To achieve this, the *StoRHm_Client* configuration file (*StoRHm_Client.xml*) was modified to generate *ServiceView* POX requests, point at the POX server (*POXServer*) and set the number of requests to 100. The client was executed and the average round-trip method invocation time for the 100 Web Service executions was recorded. The average figure was 9.7 msec.

GET – A separate RESTful HTTP client, namely *RESTFUL_WS_Client*, was developed to generate GET requests to the RESTful Web Service. The URI used by the client was the equivalent RESTful URI to the *ServiceView* requests i.e. */RESTServer/BankServices/123456/12345678*. The client loops for 100 Web Service executions and the average round-trip method invocation time was recorded. The average figure was 6.6 msec. Thus, in this example, the GET was 47% faster.

Fig. 6-13 demonstrates the comparison.

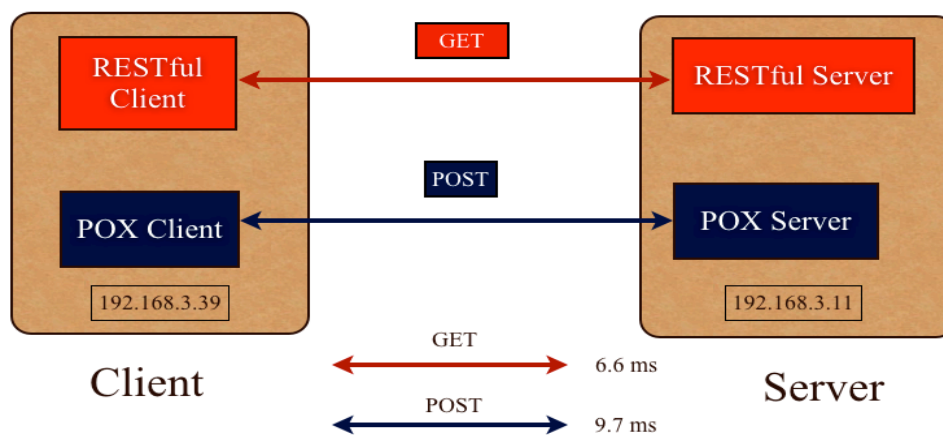


Figure 6-13 LAN: GET versus POST performance

b) POST replaced with a GET by the adapter

The POX request *ServiceView* was applicable in this scenario also. *StoRHm_Client.xml* was modified so that the same POX requests would now first travel to the adapter. In this instance, the adapter, informed by the CSV file setup earlier, converts the requests from POST to GET, maps the XML to URI format and sends them to the URI in the CSV file. The same tests were repeated 100 times and a new average round-trip method invocation time recorded. This new average figure was 14.4 msecs. Thus, in this example, the adapter imposes a 48% penalty when mapping to a GET.

Fig. 6-14 shows the comparison.

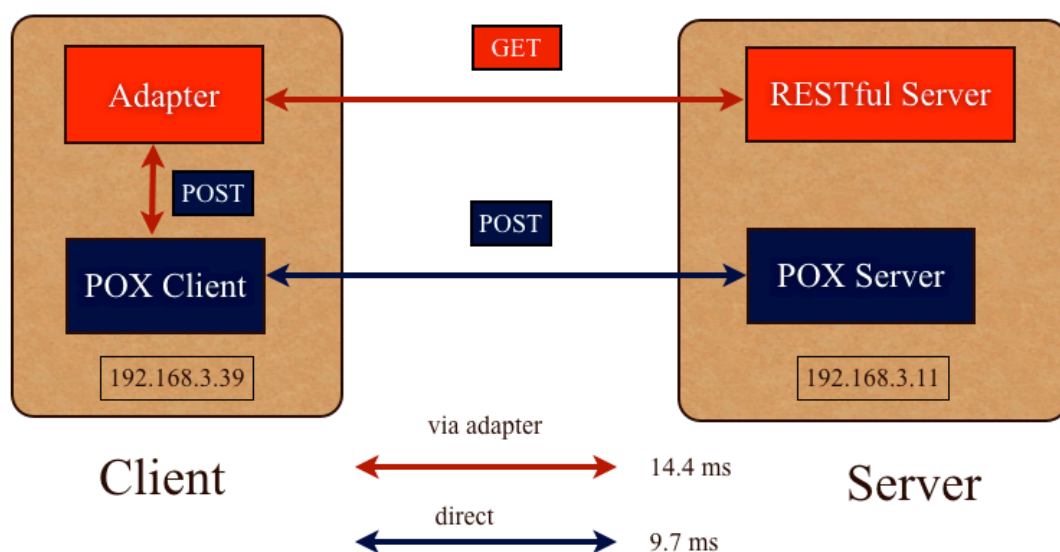


Figure 6-14 LAN: *ServiceView* performance

c) POST left untouched by the adapter

The POX request *ServiceAdd* was used in this scenario. As the message size is the same as *ServiceView*, the average round-trip method invocation time for *ServiceView* is applicable.

StoRHm_Client.xml was modified so that the same POX requests would now travel via the adapter. Whenever the URI in the CSV file is a top-level resource i.e. there are no XPath expressions in the URI path, the adapter does not parse the request in an attempt to insert values into the URI. As the CSV verb in this example is POST, the CSV URI is a collection resource. Thus, the URI from the CSV file is used as is and there is no time expended on building up a new URI based on the contents of the incoming request. The adapter leaves the POST in place and copies the entity body of the POX client request into the client adapter request to the RESTful server. The same request was repeated

100 times and a new average round-trip method invocation time recorded. This new average figure was 19 msec. Thus, the adapter imposes a 96% cost to the average round-trip invocation time.

Fig. 6-15 demonstrates the time penalty imposed by the adapter.

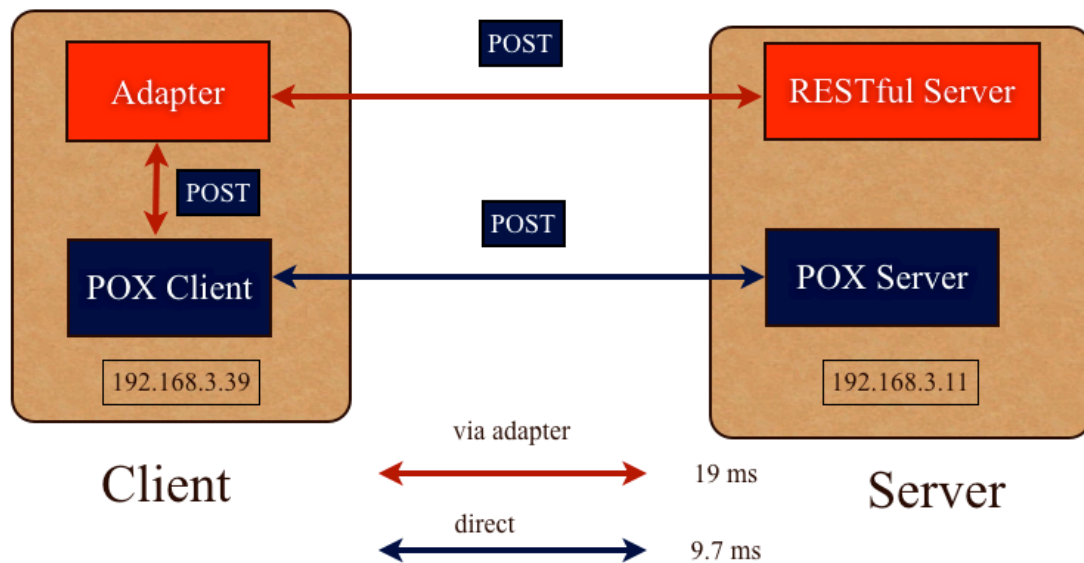


Figure 6-15 LAN: *ServiceAdd* performance

The tests outlined above, demonstrate via specific Web Service examples, the performance implications of introducing the adapter into a LAN infrastructure. These LAN-based tests are encapsulated in Table 6-2:

GET/POST	GET	POST	difference
msecs	6.6	9.7	47%

Table 6-2 (a) LAN Performance Tests – GET versus POST

LAN Performance	no Adapter	via Adapter	adapter penalty
ServiceView	9.7 msec	14.4 msec	48%
ServiceAdd	9.7 msec	19 msec	96%

Table 6-2 (b) LAN Performance Tests – *ServiceView* versus *ServiceAdd*

As can be seen from Table 6-2, the adapter imposes a significant penalty in a LAN environment. It was suspected that the penalty was excessive because any impact in a (high speed) LAN will impact negatively, relatively speaking. To test this hypothesis, it was decided that a suite of tests based on the Internet was required. It was also decided that the LAN based request files were very simple and that more realistic request files would be appropriate (see Appendix Listings 1-4). In addition, the Internet based tests form a more complete set of test cases because:

- they incorporate both SOAP and POX requests.
- a more complete set of HTTP verbs are tested i.e. GET, PUT, POST and DELETE.

6.6.2.3 Internet performance – StoRHm v2 adapter

The POX, SOAP and RESTful HTTP servers were deployed on a server on the Internet and the adapter resides on the local machine.

Internet based POX results

The four new requests: *ServiceView*, *ServiceAdd*, *ServiceDelete* and *ServiceUpdate* were executed one hundred times each. The new requests *ServiceDelete* and *ServiceUpdate*, map to the HTTP verbs DELETE and PUT respectively. As regards the extra HTTP verbs to be tested, PUT follows the POST model i.e. the adapter just copies the entity body of the Web Service request into the entity body of the RESTful request; DELETE follows the GET model i.e. parsing of the request is required in order to insert data values into the URI.

Table 6-3 outlines these results:

POX Statistics (all figures in msec)		average	standard deviation	standard error	zLow 95%	zHigh 95%	verb	adapter +/-	Significant Value 95% (+/- 1.645)
ServiceView	no adapter	93.79	11.18	1.11	92	96	POST	n/a	-9.02
	adapter	76.83	15.11	1.51	74	80	GET	-18%	
ServiceAdd	no adapter	98.87	18.58	1.85	95	102	POST	n/a	0.7
	adapter	100.3	7.74	0.7	99	102	POST	+1%	
ServiceDelete	no adapter	96.26	12.24	1.22	94	98	POST	n/a	-9.94
	adapter	74.82	17.74	1.77	71	78	DELETE	-22%	
ServiceUpdate	no adapter	95.44	15.41	1.54	92	98	POST	n/a	1.25
	adapter	97.98	13.09	1.3	95	100	PUT	+3%	

Table 6-3 POX Internet Performance Tests

As the second last column in Table 6-3 demonstrates, the adapter adds between 1-3% to the RTT if POST or PUT is the target verb. In both situations, the adapter performs no parsing and the entity body contains the XML message request. The adapter saves 18% on the average RTT if the request is *ServiceView* (which is mapped to GET) and 22% if the request is *ServiceDelete* (which is mapped to DELETE). This is explained by the fact that the original distributed POST request is now a local POST request to the adapter, followed by a more lightweight GET or DELETE distributed request to the RESTful HTTP server. Neither GET nor DELETE has an entity body in their requests and the lack of an entity body results in a smaller message and a faster RTT. For example, Appendix Listing 10 is the RESTful GET of Appendix Listing 1.

Analysis of Internet based POX results

Table 6-3 also outlines statistics gathered. To gather these statistics, each POX Web Service was directly accessed (no adapter) 100 times and the average, standard deviation, standard error, low and high values (with 95% confidence) recorded. The tests were then repeated 100 times with the client accessing the RESTful Web Services via the adapter and the results recorded again. To determine if the adapter was statistically significant in the results (with 95% confidence), we require 2 sets of hypotheses: one for situations where the adapter maps the POST to POST/PUT (*ServiceAdd/ServiceUpdate*) and another for situations where the adapter maps to GET/DELETE (*ServiceView/ServiceDelete*).

For the situation where the adapter maps to POST/PUT, the following null and alternative hypotheses are formulated (note that μ_{WA} = “with Adapter” and μ_{NA} = “No Adapter”) :

$$H_0 : \mu_{WA} = \mu_{NA}$$

$$H_1 : \mu_{WA} > \mu_{NA}$$

In this instance, a one-tailed test is appropriate and at the 95% confidence level, the t-table value is 1.645. The rows applicable in Table 6-3 are highlighted in blue (*ServiceAdd/ServiceUpdate*). *ServiceAdd* and *ServiceUpdate* have test statistic values of 0.7 and 1.25 respectively. As both values are less than 1.645, they fall within the acceptable region; thus, the null hypothesis is accepted i.e. there is no significant difference between the average times taken to issue these POX requests directly to the POX Server and the average times taken to route these requests via the adapter to their RESTful Web Service equivalents.

For the situation where the adapter maps to GET/DELETE, the following null and alternative hypotheses are formulated:

$$H_0 : \mu_{WA} = \mu_{NA}$$

$$H_1 : \mu_{WA} < \mu_{NA}$$

Again a one-tailed test is appropriate and at the 95% significance level, the t-table value is -1.645. The rows applicable in Table 6-3 are highlighted in brown (*ServiceView/ServiceDelete*). *ServiceView* and *ServiceDelete* have test statistic values of -9.02 and -9.94 respectively. In this instance, as both values are less than -1.645, they fall outside the acceptable region and the null hypothesis is rejected. The alternative hypothesis is therefore accepted i.e. the average time taken to route these POX requests via the adapter to their RESTful Web Service equivalents is significantly shorter than the average time taken to issue these requests directly to the POX Server.

Internet based SOAP results

A complete new suite of tests to evaluate SOAP performance was created. These files are listed in the Appendix (Listings 5-8). They were closely related to the POX tests in that the only difference between the requests is that the SOAP requests contain surrounding SOAP metadata. Each request was performed 100 times with/without the adapter and an average RTT calculated. Table 6-4 outlines these results:

SOAP Statistics (all figures in msec)		average	standard deviation	standard error	zLow 95%	zHigh 95%	verb	adapter +/-	Significant Value 95% (+/- 1.645)
ServiceView	no adapter	67	12.08	1.2	65	69	POST	n/a	3.45
	adapter	72.11	8.54	0.85	70	74	GET	+8%	
ServiceAdd	no adapter	65.29	7.95	0.79	64	67	POST	n/a	2.23
	adapter	68.98	14.48	1.44	66	72	POST	6%	
ServiceDelete	no adapter	66.87	13.64	1.36	64	70	POST	n/a	3.39
	adapter	72.22	7.85	0.78	71	74	DELETE	+8%	
ServiceUpdate	no adapter	65.46	7.65	0.76	64	67	POST	n/a	2.67
	adapter	67.84	4.52	0.45	67	69	PUT	+4%	

Table 6-4 SOAP Internet Performance Tests

As Table 6-4 shows, the adapter adds 4-8% to the RTT regardless of the target HTTP verb. The pattern of these figures is different from the POX figures. This is explained by the fact that while the architecture of a request routed via the adapter for both SOAP and POX requests is similar i.e. client (RESTlet) → adapter (RESTlet) → RESTful WS (RESTlet), the architecture of the

tunnelled request *is* different.

In the POX architecture, the POX client request (a RESTlet) is directed to a POX WS (also a RESTlet). In the SOAP architecture, the SOAP client request (a RESTlet) is directed to a SOAP WS (*implemented with Netbeans infrastructure*). Thus the backend SOAP and POX servers are implemented differently. Note that one of the criticisms of the WS-* stack is its complexity and to address this, there is considerable tool support. Therefore to develop SOAP WS using the tool (Netbeans in this instance) is the correct process to follow. There is no such support for POX.

As Table 6-4 demonstrates, even though the adapter maps *ServiceView* and *ServiceDelete* (with XML entity bodies) to a more lightweight GET and DELETE requests (with the data in the URI) respectively, the response times are still longer. The conclusion is therefore, that the SOAP infrastructure implemented in Netbeans is more efficient than the RESTlet implementation.

Analysis of SOAP Internet based results

As with the POX statistics, each SOAP Web Service was directly accessed (no adapter) 100 times and the average, standard deviation, standard error, low and high values (with 95% confidence) recorded. The tests were then repeated 100 times with the client accessing the RESTful Web Services via the adapter and the results recorded again. In this scenario, we require only one hypothesis i.e. either the adapter is or is not present (note that μ_{WA} = “*with Adapter*” and μ_{NA} = “*No Adapter*”):

$$H_0 : \mu_{WA} = \mu_{NA}$$

$$H_1 : \mu_{WA} > \mu_{NA}$$

A one-tailed test is appropriate and at the 95% significance level, the t-table value is 1.645. All rows are applicable in Table 6-4. *ServiceView*, *ServiceAdd*, *ServiceDelete* and *ServiceUpdate* have test statistic values of 3.45, 2.23, 3.39 and 2.67 respectively. As all values are greater than 1.645, they fall outside the acceptable region and therefore the null hypothesis is rejected. The alternative hypothesis is therefore accepted i.e. the average time taken to route these SOAP requests via the adapter to their RESTful Web Service equivalents is significantly longer than the average time taken to issue these requests directly to the SOAP Server.

Edge case example

The example SOAP and POX files used thus far (Listings 1-8 in the Appendix) were relatively small (1643 bytes) and required the adapter to perform very little parsing when mapping a POST to a GET

or DELETE i.e. to elicit the branch code and the account number required the adapter to evaluate 2 XPath expressions. An internet-based test was conducted on a large request file (9882 bytes, see Appendix: Listing 9), which requires the adapter to perform a significant amount of parsing (75 XPath expressions to be evaluated). The algorithm for parsing is the same regardless of whether the request is POX or SOAP (the adapter relies upon the XPath expressions in the CSV file to parse the incoming request) and consequently only one set of tests was necessary. Table 6-5 outlines the results:

Adapter Detailed Parsing (all figures in msec)		average	standard deviation	standard error	zLow 95%	zHigh 95%	verb	adapter +/-	Significant Value 95% (+/- 1.645)
ServiceView DetailedParse	no adapter	134.08	33.35	3.33	128	141	POST	n/a	10.42
	adapter	199.75	53.46	5.34	189	210	GET	49%	

Table 6-5 Adapter statistics when significant Parsing required

As Table 6-5 outlines, the adapter adds 49% to the RTT. This is due to the significant increase in the amount of parsing required.

Analysis of Detailed Parsing results

In this example, to determine if the adapter was statistically significant in the results (with 95% confidence), the following null and alternative hypotheses are formulated (note that μ_{WA} = “with Adapter” and μ_{NA} = “No Adapter”):

$$H_0 : \mu_{WA} = \mu_{NA}$$

$$H_1 : \mu_{WA} > \mu_{NA}$$

In this instance, a one-tailed test is appropriate and at the 95% confidence level, the t-table value is 1.645. The test statistic value from Table 6-5 is 10.42 and as it is greater than 1.645, it falls outside the acceptable region and therefore the null hypothesis is rejected. The alternative hypothesis is therefore accepted i.e. the average time taken to route these requests via the adapter is significantly longer than the average time taken to issue these requests directly to the server.

6.6.2.4 Message Visibility

This section pertains to both the POX and StoRHm adapters. The new messages are now visible to Web intermediaries. This is important as XML Web Service implementations currently have to find alternative ways to cache their messages [9][10][129]. As Vinoski states: *“the uniform-interface constraint helps enable visibility into client-server interactions, making it easier for developers to apply critical distributed systems concepts such as proxying, caching, intermediation and monitoring”* [12].

6.6.3 Ease of Migration

Both adapters would suit enterprises that are considering migrating from SOAP/POX to RESTful HTTP Web Services or where a takeover/amalgamation of an enterprise has taken place and RESTful HTTP is deemed the way forward. In both scenarios, while the server would migrate from SOAP/POX to RESTful HTTP Web Services immediately, a wholesale replacement of the clients can be avoided by adopting StoRHm i.e. no re-compilation of the clients is required as, from the clients perspective, the interface is still the same. These clients can migrate when convenient. Therefore, any enterprise wishing to migrate their Web Services from SOAP/POX to RESTful HTTP can do so gradually, without impacting existing clients, by adopting StoRHm. More importantly, their messages are now visible to Web intermediaries.

6.7 Summary

This chapter focused on the implementation, testing and evaluation of StoRHm. The implementation instances were presented. The configuration wizard and protocol adapters of StoRHm v1 (both SOAP and POX) and StoRHm v2 were detailed. Test cases were outlined and on-the-wire messages were presented showing the adapter transforming the incoming opaque SOAP/POX messages to visible RESTful HTTP format were shown. The configuration wizards were evaluated on the XML operations supported and the URI length restrictions. The protocol adapters were evaluated on message visibility. In addition, the StoRHm v2 protocol adapter was evaluated on its performance impact in both a LAN and Internet setting. Statistics were presented based on the Internet performance metrics. Lastly, StoRHm was evaluated based on how well it eases enterprise migration.

Chapter 7

Conclusion

This chapter summarises the thesis. Section 7.1 provides an overview of the research theme of this thesis. Section 7.2 provides a summary of the principal contributions of the research and Section 7.3 provides some possible ways this work can be extended in the future.

7.1 Thesis Summary

The research work detailed in this thesis focused on an architecture that maps XML-based Web Services to RESTful HTTP format. This architecture leverages Semantic Web technologies to provide intelligent automation.

The thesis began by presenting a background to all the underpinning technologies relevant to this thesis: XML-based Web Services, REST and the Semantic Web. The literature was critically examined in order to establish areas where valid research questions remained. These questions centred on mapping an XML Web Service invocation to RESTful HTTP format. The contributions of this research, motivated by these research questions, are discussed in Section 7.2

The dissertation clearly defined the contribution made by the research work. The focus of research in this thesis was in mapping from one Web Service paradigm to another. An architecture, namely StoRHm, consisting of both a configuration wizard and a runtime adapter was presented. The configuration wizard creates a mapping file that informs the adapter when transforming a message. StoRHm v2 is a semantically-informed version of StoRHm v1. Testing was performed to demonstrate the ability of StoRHm to deliver on its requirements, especially message visibility.

7.2 Research Conclusions

A number of research contributions were described in the Introduction of this thesis. As part of the conclusion of the research, this section will detail how each of these contributions was achieved using the architecture. The contribution areas are: enabling the Web infrastructure, seamless Web Service migration, novel Web Services caching, novel Web Services interoperability, novel use of Semantic Web Services stacks and SWEET editing tool contribution.

Web Infrastructure enabler

One of the key drivers for this research was to map SOAP and POX Web Services to RESTful HTTP format. Both SOAP and POX Web Services issue POST requests to a gateway URI, with the request to be executed encapsulated inside the XML entity body of the message. As the request is opaque, this disables Web intermediaries.

This thesis has proposed, built and demonstrated a client-side architecture that transforms opaque XML Web Service invocations to visible RESTful HTTP format. All of the REST constraints are supported: unique id's (URIs), Hypermedia (links within the response documents), Uniform Interface (GET, PUT, POST and DELETE), Multiple Representations (XML only because the client is SOAP) and stateless communication (no server state is relied upon). SOAP and POX requests are transformed into RESTful HTTP requests by the adapter on the client. Thus, the requests that leave the client are immediately visible, thereby enabling the inherent efficiencies of Web intermediaries.

The constraints imposed by the adapter element are that the RESTful Web Services which respond to PUT/POST should expect the SOAP/POX payload untouched. Also, complex SOAP/POX requests that map to more than one logical URI will be passed on as a POST with the payload untouched.

Migration enabler

Enterprises wishing to migrate from SOAP/POX Web Services to RESTful Web Services are confronted with a situation than when the RESTful server replaces the backend SOAP/POX server, all the clients must be replaced at the same time. As Vinoski states: *“enterprises are quite concerned with avoiding technology changes that require wholesale replacement -- they prefer approaches that allow them to gradually shift from one technology to another.”* [7].

The architecture presented in this thesis offers enterprises an alternative approach to wholesale replacement and is ideally suited to enable enterprises to gradually migrate from SOAP/POX Web Services to RESTful HTTP Web Services without impacting existing clients. This is confirmed by Vinoski: *“Your approach of using a service-specific gateway could potentially provide a way for servers to migrate from SOAP/WS-* to REST without breaking existing clients, allowing those clients to migrate when convenient”* [7]. The architecture enables SOAP/POX clients to migrate when convenient because their interface remains the same, even when the RESTful server replaces the backend SOAP/POX server. The SOAP/POX clients interact with the adapter, which transforms the request into RESTful format before forwarding it on to the RESTful server. The response, in the case of SOAP clients, is wrapped in SOAP before returning to the client.

Novel approach to XML Web Service caching

Traditional approaches to SOAP/POX Web Services caching involve either local [9] or server-side [10] datastores. The architecture in this thesis enables “XML Web Service” caching by leveraging the proven Web infrastructure. This is possible because the messages are visible on leaving the client. The architecture implements a transparent solution due to the fact that the client interface remains untouched.

Novel Web Services interoperability

This thesis presents an architecture that, via use of Semantic Web technologies, implements an intelligent, automated client-side SOAP/POX to RESTful HTTP mapping. The architecture is client-side thereby ensuring on-the-wire message visibility. As the existing SOAP/POX application-specific interfaces are taken into account in the architecture, client impact is minimal.

Semantic Web Services

During the research examination of literature in Semantic Web Services, it became evident that the

Semantic layer is used for integration, mediation and composability within and across the SOAP and RESTful stacks. In the literature however, when a RESTful WS follows a SOAP WS, it is always the output of the SOAP WS that is lifted to the Semantic layer; from where it is lowered to the RESTful WS input format.

The model presented in this thesis, uses the Semantic layer to transform the SOAP input message as opposed to the SOAP output message. Consequently, the SOAP WS is never executed.

In addition, the architecture presented here integrate POX WS. Although SAWSDL caters for both WSDL and XML Schema, the research conducted as part of this thesis indicates that all the research efforts have been with regard to SOAP and RESTful WS.

SWEET contribution

SWEET is a lightweight, online, Semantic Web Service annotation tool for adding semantics to RESTful WS descriptions. As mentioned in the Introduction (Section 1.3), a core issue was encountered and reported. This issue was subsequently fixed.

7.3 Future Work

The research presented in this thesis can be progressed in a number of ways. The following is a list of recommendations for future work:

- One of the constraints imposed by the architecture is that, in situations where the target HTTP verb to be used is PUT or POST, the entity body of the request is sent on untouched. Typically, the outer element of the SOAP Body element contains the operation to be executed e.g. the WSDL wrapped document-literal pattern enforces this. However, this “operation” element is not needed by RESTful HTTP implementations as the “operation” is identified by the URI coupled with the verb. In order to address this, research is required on XSLT transformations to cater for scenarios as described above, where the XML content to be passed on differ from the XML content received.
- The other constraint imposed by the architecture is that, in use cases where complex SOAP/POX requests map to multiple URIs, the POST is left in place and the entity body is passed on untouched. This results in one coarse-grained call replacing several fine-grained calls. As outlined in section 6.6.1.1, this pattern is known as the “Remote Facade” pattern [100]. An extension to the current architecture could be implemented to generate these fine-grained calls and then measure the performance of the multiple fine-grained calls against the current coarse-grained call.

- This thesis is not centred on performance and consequently the performance tests carried out are indicative rather than extensive. Exploring the performance of the architecture could be extended. This would include: round-trip time (RTT) of different request file sizes; the performance impact of message reliability and/or security; and the effect of multiple similar requests with efficiencies such as caching and Conditional GET in place.
- The architecture enables SOAP/POX clients access to pre-existing RESTful HTTP Web Services. The adapter is a client-side migration enabler. Research could be conducted to extend the framework to focus on the server i.e. provide a server-side migration enabler from SOAP/POX to RESTful HTTP. Should an enterprise wish to migrate from SOAP/POX WS to RESTful HTTP WS, this new extension would be executed first to migrate the server. With the server migrated, the current framework would then be used to enable the enterprise to gradually migrate the clients.

References

- [1] S. Vinoski, “An Overview of Middleware”, proceedings of *European International Conference on Reliable Software Technologies*, 2004.
- [2] W. Vogels, “Web Services Are Not Distributed Objects”, *IEEE Internet Computing*, Vol. 7, Issue. 6, pp. 59-66, 2003.
- [3] R. Fielding, “Architectural Styles and the Design of Network-based Software Architectures”, PhD thesis, University of Irvine, California, 2000, available at <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- [4] S. Tilkov, Software Engineering Radio interview recorded at OOP '08, available at <http://www.se-radio.net/2008/05/episode-98-stefan-tilkov-on-rest/>
- [5] R. L. Costello, “REST (Representational State Transfer)”, available at <http://www.xfront.com/files/tutorials.html>
- [6] P. Alexander (VP Cisco Worldwide Field Marketing), “The Internet in Business: How the Internet and Internet Applications are changing business processes and enabling new business models”, keynote address at *International Conference on Internet Technology and Applications*, Wrexham, 2009.
- [7] S. Vinoski, Verivue and Senior Member of IEEE, private communication.
- [8] J. Briggs, “Playing Together Nicely: Getting REST and SOAP to Share Each Other’s Toys”, available at <http://onjava.com/pub/a/onjava/2006/02/15/jython-soap-interface-to-rest.html>

- [9] Microsoft Research , “Caching XML Web Services for Mobility”, Queue magazine 2003.
- [10] D. Andersen, K.Devaram and VP. Ranganath, “LYE: a high-performance caching SOAP implementation”, in *Proceedings of the International Conference on Parallel Processing, 2004*.
- [11] S. Vinoski, “Serendipitous Reuse”, *IEEE Internet Computing, Vol. 12, Issue 1, pp 84-87, 2008*.
- [12] S. Vinoski, “Demystifying RESTful Data Coupling”, *IEEE Internet Computing, Vol. 12, Issue 2, pp 87-90, 2008*.
- [13] M. Maleshkova, C. Pedrinaci and J. Domingue, “Semantic annotation of Web APIs with SWEET”, in *6th Workshop on Scripting and Development for the Semantic Web at Extended Semantic Web Conference, 2010*.
- [14] World Wide Web Consortium (W3C),
<http://www.w3.org/TR/ws-gloss/>
- [15] E. Newcomer and G. Lomow, “Understanding SOA with Web Services”, *Addison-Wesley, 2005*.
- [16] S. Graham, D. Davis, S. Simeonov, G. Daniels, P. Brittenham, Y. Nakamurs, P. Fremantle, D. Konig and C. Zentner, “Building Web Services with Java”, *Sams Publishing, 2005*.
- [17] World Wide Web Consortium (W3C),
<http://www.w3.org/2003/Talks/0521-hh-wsa/soa.png>
- [18] M. P. Papazoglou. “Web Services: Principles and Technology”, *Prentice Hall, 2008*.
- [19] Web Services Interoperability Organisation, <http://www.ws-i.org/>
- [20] Web Services Interoperability Organisation, Basic Profile v1.1
http://www.ws-i.org/Profiles/BasicProfile-1.1-2006-04-10.html#SOAP_encodingStyle_Attribute
- [21] R. Butek (IBM Developer Works), “Which style of WSDL should I use?”, available at
<http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/>
- [22] A. T. Manes (Burton Group consultant), “The "wrapped" document/literal convention” available at

<http://atmanes.blogspot.com/2005/03/wrapped-documentliteral-convention.html>

- [23] G. Flurry (IBM Developer Works), “Create Wrapped Document-Literal WSDL in WebSphere Studio Application Developer”, available at
http://www.ibm.com/developerworks/websphere/library/techarticles/0505_flurry/0505_flurry.html
- [24] SOAP 1.2 Specification Part 2 Adjuncts (Second Edition),
<http://www.w3.org/TR/2007/REC-soap12-part2-20070427/>
- [25] S. Tilkov (principal consultant at innoQ), communication via the REST discussion group at <http://tech.groups.yahoo.com/group/rest-discuss/>
- [26] S. Tilkov (principal consultant at innoQ), presentation at Devovx '08, available at
<http://wiki.parleys.com/display/PARLEYS/Home#;talk=31817742>
- [27] T. Takase, S. Makino, S. Kawanaka , K. Ueno, C. Ferris and A. Ryman, “Definition Languages for RESTful Web Services: WADL vs WSDL 2.0.”, IBM DeveloperWorks White Paper, available at
<http://www.ibm.com/developerworks/library/specification/ws-wadlwsdl/index.html#download>
- [28] A. Khalifa and S. Minocha, “Accessing SAP Systems Using WebSphere Studio Application Developer – Part 1”, *IBM WebSphere Developer Technical Journal*, available at
http://www.ibm.com/developerworks/websphere/techjournal/0304_minocha/minocha.html
- [29] SOAP 1.1 specification, available at
<http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>
- [30] M. Baker and M. Nottingham, RFC 3902: “The ‘application/soap+xml’ media type”, available at
<http://www.ietf.org/rfc/rfc3902.txt>
- [31] A. Vedamuthu and D. Roth, “Understanding Web Services Policy”, *Microsoft Developer Network*, available at
<http://msdn.microsoft.com/en-us/library/ms996497.aspx>
- [32] Web Services Description Language (WSDL) Version 2.0 specification,
<http://www.w3.org/TR/wsdl20/>

- [33] Web Services Description Language (WSDL) Version 1.1 specification,
<http://www.w3.org/TR/wsdl>
- [34] Netbeans IDE v6.8, *<http://netbeans.org/community/releases/68/>*
- [35] WSDL 1.1 versus WSDL 2.0 image,
http://en.wikipedia.org/wiki/Web_Services_Description_Language
- [36] Web Services Description Language (WSDL) Version 2.0 Adjuncts,
<http://dev.w3.org/2002/ws/desc/wsdl20/wsdl20-adjuncts.html>
- [37] WSDL 2.0 Primer,
<http://www.w3.org/TR/2007/PR-wsdl20-primer-20070523>
- [38] Olaf Zimmermann (IBM Research), Software Engineering Radio interview at OOPSLA '07, available at *<http://www.se-radio.net/2008/02/episode-85-web-services-with-olaf-zimmermann/>*
- [39] S. Tilkov, "UDDI R.I.P", available at
http://www.innoq.com/blog/st/2010/03/uddi_rip.html
- [40] A. Ngu, M. Carlson, Q. Sheng and H. Paik "Semantic-Based Mashup of Composite Applications", *IEEE Transactions on Services Computing, Vol. 2, No. 1 pp2-15, 2010.*
- [41] C. O'hEigeartaigh, "Deploying WS-Security", keynote address at *IEEE European Conference on Web Services (ECOWS)*, Dublin, 2008.
- [42] "REST and POX", *Microsoft Developer Network* article, available at
<http://msdn.microsoft.com/en-us/library/aa395208.aspx>
- [43] S. Vinoski, "Putting the 'Web' into Web Services",
IEEE Internet Computing Vol. 6, Issue 4, pp 90-92, 2002.
- [44] S. Allamaraju. "RESTful Web Services Cookbook", *O'Reilly 2010.*
- [45] S. Ruby and L. Richardson, "RESTful Web Services", *O'Reilly 2007.*
- [46] Internet Assigned Numbers Authority (IANA) Multipurpose Internet Mail Extensions (MIME) Media Types
<http://www.iana.org/assignments/media-types/>
- [47] Flickr, *www.flickr.com*
- [48] S. Vinoski (Senior member of IEEE), "REST, Reuse and Serendipity" presentation at QCon 2008, available at
<http://www.slideshare.net/QConLondon2008/rest-reuse-and-serendipity>.
- [49] S. Vinoski, "RESTful Web Services Development Checklist",

- [50] P. Lacey (Burton Group consultant), “REST Easy” presentation, available at http://i.bnet.com/whitepapers/burton_RESTEasy.ppt
- [51] Hypertext Transfer Protocol specification 1.1, RFC 2616, available at <http://www.w3.org/Protocols/rfc2616/rfc2616.html>.
- [52] S. Tilkov (principal consultant at innoQ), “A Brief Introduction to REST”, available at <http://www.infoq.com/articles/rest-introduction>
- [53] N. Gray, “Web Server Programming”, *Wiley Publishing, 2003.*
- [54] The Coyote HTTP/1.1 connector
<http://tomcat.apache.org/tomcat-4.1-doc/config/coyote.html>
- [55] S. Tilkov (principal consultant at innoQ), “Addressing Doubts about REST”, available at <http://www.infoq.com/articles/tilkov-rest-doubts>
- [56] M. Nottingham, “Post Once Exactly” Internet draft, available at <http://tools.ietf.org/html/draft-nottingham-http-poe-00>.
- [57] B. de hOra, “HTTPPLR“ Internet Draft, available at <http://dehora.net/doc/httpplr/draft-httpplr-01.html>
- [58] J. Gregorio, “RESTify DayTrader”, available at <http://bitworking.org/news/201/RESTify-DayTrader>
- [59] Roy Fielding (author of REST), communication via the REST discussion group on Yahoo, <http://tech.groups.yahoo.com/group/rest-discuss/>
- [60] R. Tomayko , “How I explained REST to my Wife”, available at <http://tomayko.com/writings/rest-to-my-wife>
- [61] T. Berners-Lee, “Universal Resource Identifiers – Axioms of Web Architecture” 1996, available at <http://www.w3.org/DesignIssues/Axioms.html>.
- [62] Architecture of the World Wide Web, W3C Recommendation 2004, available at <http://www.w3.org/TR/webarch/>
- [63] T. Berners-Lee, “Cool URIs don’t change” 1998, available at <http://www.w3.org/Provider/Style/URI>
- [64] T. Berners-Lee, R. Fielding and L. Masinter, “Uniform Resource Identifiers (URI): Generic Syntax” 1998, available at <http://tools.ietf.org/html/rfc2396>
- [65] G. Zheng and A.Bouguettaya “Service Mining on the Web”, *IEEE Transactions on Services Computing, Vol. 2, No. 1 pp65-78, 2009.*

- [66] T. Berners-Lee, J. Hendler and Ora Lassila, “The semantic Web”, *Scientific American*, vol. 284, no. 5, pp.34-43, 2001.
- [67] J. Hebel, M. Fisher, R. Blace, A. Perez-Lopez, “Semantic Web Programming”, *Wiley publishing*, 2009.
- [68] Resource Description Framework Specification,
<http://www.w3.org/RDF/>
- [69] Resource Description Framework Primer,
<http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>
- [70] RDF/XML Specification,
<http://www.w3.org/TR/rdf-syntax-grammar/>
- [71] XML Schema Datatypes.
<http://www.w3.org/TR/xmlschema-2/>
- [72] Turtle – Terse RDF Triple Language,
<http://www.w3.org/TeamSubmission/turtle/>
- [73] T. Segaran, C. Evans and J. Taylor, “Programming the Semantic Web”, *O’Reilly publishing*, 2009.
- [74] RDF Schema, <http://www.w3.org/TR/rdf-schema/>
- [75] W3C Web Ontology Language
<http://www.w3.org/TR/2009/REC-owl2-overview-20091027/>
- [76] W3C OWL Web Ontology Language Overview
<http://www.w3.org/TR/owl-features/>
- [77] J. Hendler, Software Engineering Radio interview 2008, available at
<http://www.se-radio.net/2008/11/episode-116-the-semantic-web-with-jim-hendler/>
- [78] SPARQL Protocol and RDF Query Language (SPARQL),
<http://www.w3.org/TR/rdf-sparql-query/>.
- [79] W3C SPARQL Query Results XML Format,
<http://www.w3.org/TR/rdf-sparql-XMLres/>
- [80] J. Kopecky, T. Vitvar and D. Fensel, MicroWSMO and hRESTS,
available at <http://sweet.kmi.open.ac.uk/pub/microWSMO.pdf>
- [81] SOA4All Dashboard tool suite,
<http://coconut.tie.nl:8080/dashboard/#1304859734132>
- [82] Semantic Annotations for WSDL and XML Schema.
<http://www.w3.org/TR/sawSDL/>
- [83] Jon Lathem, Karthik Gomadam and Amit Sheth.

- “SA-REST and (S)mashups: Adding Semantics to RESTful Services”,
in *Proceedings of International Conference on Semantic Computing*,
2007.
- [84] WADL W3C submission, <http://www.w3.org/Submission/wadl/>, 2009.
- [85] Jacek Kopecky, Karthik Gomadam and Tomas Vitvar.
“hRESTS: and HTML microformat for describing RESTful Web
Services”, in *Proceedings of IEEE/WIC/ACM International
Conference on Web Intelligence and Intelligent Agent Technology*,
2008.
- [86] Microformats. <http://microformats.org/wiki/introduction>
- [87] hCard microformat specification, <http://microformats.org/wiki/hcard>.
- [88] hCalendar microformat specification,
<http://microformats.org/wiki/hcalendar>.
- [89] Poshformat, <http://microformats.org/wiki/poshformats>
- [90] Gleaning Resource Descriptions from Dialects of Languages
specification, <http://www.w3.org/TR/grddl/>.
- [91] Jacek Kopecky, Tomas Vitvar and Dieter Fensel.
“WSMO-Lite: Lightweight Semantic Descriptions for Services on the
Web”, in *Proceedings of International Conference on Web Services*,
2007.
- [92] Web Services Modelling Ontology (WSMO), <http://www.wsmo.org/>
- [93] Semantic Universe, [http://www.semanticuniverse.com/articles-
lightweight-semantic-extensions-soa.html](http://www.semanticuniverse.com/articles-lightweight-semantic-extensions-soa.html)
- [94] Semantic Annotations for REST submission,
<http://www.w3.org/Submission/SA-REST/>.
- [95] RDF-in-attributes specification, <http://www.w3.org/TR/rdfa-syntax/>.
- [96] RDF in Attributes Primer, <http://www.w3.org/TR/xhtml-rdfa-primer/>.
- [97] RDFa in XHTML: Syntax and Processing,
<http://www.w3.org/TR/rdfa-syntax/>
- [98] Protégé, <http://protege.stanford.edu/>
- [99] TCPMon, <http://ws.apache.org/commons/tcpmon/>
- [100] Martin Fowler, “Patterns of Enterprise Application Architecture”,
Addison-Wesley, 2003.
- [101] C. Pautasso, O. Zimmermann, F. Leymann, “RESTful Web Services
vs. ‘Big’ Web Services: Making the Right Architectural Decision” in

- Proceedings of the 17th World Wide Web Conference*, pp 805-814, 2008.
- [102] M. Juric, B. Kezmah, M. Hericko, I. Rozman and I Vezocnik, “Java RMI, RMI Tunneling and Web Services Comparison and Performance Analysis”, *SIGPLAN Not.*, vol. 39, no. 5, pp. 58–65, 2004.
- [103] K. Page, D. De Roure and K. Martinez, “REST and Linked Data: a match made for domain driven development”, *2nd International Workshop on RESTful Design (WS-REST) 2011*.
- [104] C. Bizer, T. Heath and T. Berners-Lee, “Linked Data – The Story So Far”, *International Journal on Semantic Web and Information Systems*, Volume 5, Issue 3, 2009.
- [105] S. Staab, “The Emerging Web of Linked Data”, *IEEE Intelligent Systems journal*, Volume 24, Issue 5, pp 87-92, 2009.
- [106] A. Paliwal, B. Shafiq, J. Vaidya, H. Ziong and N. Adam, “Semantics Based Automated Service Discovery”, accepted for publication in the *IEEE Transactions on Services Computing*.
- [107] M. Kuehnhausen, V. Frost, “Framework for Analyzing SOAP Messages in Web Service Environments”, *International Journal of Web Services Practices*, Vol. 5, No. 1, pp. 1-9, 2010.
- [108] N.A. Sultan, “The Evolving Model for Software Delivery: The Case of Web and Semantic Services”, *International Journal of Web Services Practices*, Vol. 3, No. 1-2, pp. 57-65, 2008.
- [109] J. Kopecky, T. Vitvar, C. Bournez and J. Farrell, “SAWSDL: Semantic Annotations for WSDL and XML Schema”, *IEEE Internet Computing*, Vol. 11, No. 6, pp60-67, 2007.
- [110] V. Dabhi, H. Prajapati, V. Doshi and K. Chokski, “Developing Enterprise Solution with Web Services Integration”, *International Journal of Web Services Practices*, Vol. 4, No. 1, pp. 11-17, 2009.
- [111] H. Artail, K. Fawaz and A. Ghandour, “A proxy-based architecture for dynamic discovery and invocation of Web Services from mobile devices”, accepted for publication in the *IEEE Transactions on Services Computing*.

- [112] R. Fielding and R. Taylor, "Principled Design of the Modern Web Architecture", *ACM Transactions on Internet Technology*, Vol. 2, No. 2, pp. 115-150, 2002.
- [113] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi and S. Weerawarana, "Unraveling the Web Services Web", *IEEE Internet Computing*, Vol. 6, Issue. 2, pp86-93, 2002.
- [114] P. Louridas, "SOAP and Web Services", *IEEE Internet Computing*, Vol. 23, Issue. 6, pp62-67, 2006.
- [115] M. Hepp, "Semantic Web and Semantic Web Services: father and son or indivisible twins?", *IEEE Internet Computing*, Vol. 10, Issue. 2, pp85-88, 2006.
- [116] W. Kongdenfha, H. R. Motahari-Nezhad, B. Benatallah, F. Casati and R. Saint-Paul, "Mismatch Patterns and Adaptation Aspects: A Foundation for Rapid Development of Web Service Adapters", *IEEE Transactions on Services Computing*, Volume 2, No. 2, pp 94-107, 2009.
- [117] G. Alonso, F. Casati, H. Kuno and V. Machiraju, "Web Services – Concepts, Architectures and Application", *Springer-Verlag*, 2004.
- [118] B. Benatallah, F. Casati, D. Grigori, H. R. Motahari-Nezhad, and F. Toumani, "Developing Adapters for Web Services Integration", *Advanced Information Systems Engineering, Lecture Notes in Computer Science*, Springer, Volume 3520/2005, pp 415-429, 2005.
- [119] A. Gokhale, B. Kumar and A. Sahuguet, "Reinventing the wheel? CORBA vs. Web Services", in *Proceedings of International World Wide Web Conference*, 2002.
- [120] R. Fatoohi, V. Gunwani, Q. Wang and C. ZHeng, "Performance evaluation of middleware bridging technologies", *IEEE International Symposium on Performance Analysis of Systems and Software*, pp34-39, 2000.
- [121] U. Lampe, S. Schulte, M. Siebenhaar, D. Schuller and R. Steinmetz, "Adaptive Matchmaking for RESTful Services based on hREST and MicroWSMO", in *Proceedings of the 5th International Workshop on Enhanced Web Services Technologies*, pp10-17, 2010.
- [122] J. Mangler, E. Schikuta and C. Witzany, "Quo Vadis Interface Definition Language? Towards a Interface Definition Language for

- RESTful Services”, in *Proceedings of IEEE International Conference on Service-Oriented Computing and Applications*, pp1-4, 2009.
- [123] R. Khare and Richard N. Taylor, “Extending the Representational State Transfer (REST) Style for Decentralised Systems”, in *Proceedings of 26th International Conference on Software Engineering*, pp428-437, 2004.
- [124] R. T. Fielding, E. Whitehead, K. Anderson, G. Bolcer, P. Oreizy and R. N. Taylor, “Web-based development of complex information products”, *Communications of the ACM* 41,8, pp84-92, Aug. 1998.
- [125] A. Marinos, A. Razavi, S. Moschoviannis and P. Krause, “RETRO: A Consistent and Recoverable RESTful Transaction Model”, in *Proceedings of IEEE International Conference on Web Services*, pp181-188, 2009.
- [126] “JSR-311: JAX-RS: Java API for RESTful Web Services”, *Java Community Process*, Sun Microsystems, 2007.
- [127] S. Vinoski, “WS-Nonexistent Standards”, *IEEE Internet Computing*, Vol. 8, Issue. 6, pp94-94, 2004.
- [128] T. Moser and S. Biffl, “Semantic Integration of Software and Systems Engineering Environments”, accepted for publication in the *IEEE Transactions on Systems, Man and Cybernetics*.
- [129] H. Artail and S. Saab, “A Distributed System for Consuming Web Services and Caching their responses in MANETs”, *IEEE Transactions on Services Computing*, Vol.2 No. 1, pp17-33, 2009.
- [130] S. Kim and M. Rosu, “A Survey of Public Web Services”, *Springer LNCS*, Volume 3182/2004, pp96-105, 2004.
- [131] D. Guinard and S. Karnouskos, “Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection and On-Demand Provisioning of Web Services”, *IEEE Transactions on Services Computing*, Volume 3 no. 3, pp223-235, 2010.
- [132] M. Crasso, A. Zunino and M. Campo, “Easy Web Service Discovery: A Query-by-Example Approach”, *Science of Computer Programming*, Volume 71, No. 2, pp144-164, 2008.
- [133] H. Wang, J. Huang, Y. Xu and J. Xie, “Web Services: Problems and Future Directions”, *Journal of Web Semantics*, Volume 11 No. 3, pp309-320, 2004.

- [134] D. Guinard, V. Trifa, T. Pham and O. Liechti, "Towards Physical Mashups in the Web of Things", in *Proceedings of IEEE Sixth International Conference of Networked Sensing Systems*, pp196-199, 2009.
- [135] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos and K. Kim, "TinyREST – A Protocol for Integrating Sensor Networks into the Internet", in *Proceedings of Workshop REALWSN*, 2005.
- [136] J. M. Doderio and E. Ghiglione, "REST based Web Access to Learning Design Services", *IEEE Transactions on Learning Technologies*, Volume 1, No. 3, pp190-195, 2008.
- [137] M. zur Muehlen, J.V. Nickerson and K.D. Swenson, "Developing Web Services Choreography Standards – The Case of REST versus SOAP", *Decision Support Systems*, Volume 40, No.1, pp9-29, 2005.
- [138] S. Kennedy, O. Molloy, R. Stewart and P. Jacob, "StoRHm: A protocol adapter for mapping SOAP based Web Services to RESTful HTTP format", *Electronic Commerce Research Journal*, Volume 11, Issue 3, pp245-269, 2011.
- [139] S. Kennedy, O. Molloy, R. Stewart, P. Jacob and G. Daglioglu, "StoRHm: A semantically automated protocol adapter for mapping SOAP based Web Services to RESTful HTTP format", in *proceedings of 4th International Conference on Internet Technologies and Applications*, Wrexham, 2011.
- [140] S. Kennedy, O. Molloy and R. Stewart, "Leveraging the Semantic Web to automate the mapping of SOAP Web Services to RESTful HTTP format", in *proceedings of 6th International Conference in Networking and Electronic Commerce Research*, Italy, 2010.
- [141] S. Kennedy, O. Molloy and R. Stewart, "An adapter for mapping Plain Old XML (POX) Web Services to RESTful HTTP", in *proceedings of International Conference on e-Society*, Portugal, 2010.
- [142] S. Kennedy, O. Molloy, R. Stewart and P. Jacob, "StoRHm: A novel framework for RESTifying SOAP based Web Services", in *proceedings of 5th International Conference in Networking and Electronic Commerce Research*, Italy, 2009.

- [143] S. Kennedy and O. Molloy, "A novel approach to mapping Web Services to REST", *in proceedings of 3rd International Conference on Internet Technologies and Applications, Wrexham, 2009.*
- [144] S. Kennedy and O. Molloy, "A framework for transitioning Enterprise Web Services from XML-RPC to REST", *in proceedings of 3rd International Conference on Information Resources Management, UAE, 2009.*
- [145] S. Kennedy, O. Molloy and I. Richardson, "eCommerce Web Services: REST or WS-*", *in proceedings of 4th International Conference in Networking and Electronic Commerce Research, Italy, 2008.*
- [146] N. A. Nordbotten, "XML and Web Services Security Standards", *IEEE Communications Surveys and Tutorials, Vol. 11, No. 3, pp 4-21, 2009.*
- [147] L. DeLooze, "Providing Web Service Security in a Federated Environment", *IEEE Security and Privacy, Vol. 5 Issue 1, pp73-75, 2007.*
- [148] M. Naedele, "Standards for XML and Web Services Security", *Computer, Vol. 36 Issue 4, pp96-98, 2003.*
- [149] J. Viega and J. Epstein "Why Applying Standards To Web Services Is Not Enough", *IEEE Security and Privacy, Vol. 4 Issue 4, pp25-31, 2006.*
- [150] A. Bouguettaya, S. Nepal, W. Sherchan, X. Zhou, J. Wu, S.Chen, D. Liu, L. Li, H. Wang and X. Liu "End-to-End Service Support for Mashups", *IEEE Transactions on Services Computing, Vol. 3 No. 3, pp250-263, 2010.*
- [151] B. Medjahed, A. Bouguettaya and A.K. Elmagarmid "Composing Web Services on the Semantic Web", *VLDB Journal, Vol. 12 No. 4, pp333-351, 2003.*
- [152] A. Segev and Q.Z. Sheng "Bootstrapping Ontologies for Web Services", *IEEE Transactions on Services Computing, Vol. 3, No. 99 pp1-13, 2010.*
- [153] A. Segev and E.Toch "Context-Based Matching and Ranking of Web Services for Composition", *IEEE Transactions on Services Computing, Vol. 2, No. 3 pp210-222, 2009.*

- [154] R. Battle and E. Benson “Bridging the Semantic Web with Representational State Transfer (REST)”, *Journal of Web Semantics*, Vol. 6, No. 1 pp61-69, 2008.
- [155] J. Kopecky and T. Vitvar “WSMO-LITE: Lowering the Semantic Web Services barrier with modular and light-weight annotations”, in proceedings of *IEEE International Conference of Semantic Computing*, pp238-244, 2008.
- [156] T. Khdour and M. Fasli “A Semantic-based Web Service Registry Filtering Mechanism”, in proceedings of *IEEE International Conference on Advanced Information Networking and Applications Workshops*, pp373-378, 2010.
- [157] Sample POX file from industry (financial institution).

Appendix

Listing 1 – POX Internet-based *ServiceView* Request

```
<?xml version="1.0" encoding="UTF-8"?>
<Request>
  <Transaction>ServiceView</Transaction>
  <ServiceView>
    <BRANCH_CODE>123456</BRANCH_CODE>
    <ACCOUNT_NUMBER>12345678</ACCOUNT_NUMBER>
    <CUSTOMER_TITLE>Mr.</CUSTOMER_TITLE>
    <CUSTOMER_FIRSTNAME>Joseph</CUSTOMER_FIRSTNAME>
    <CUSTOMER_MIDDLE_INITIAL>P</CUSTOMER_MIDDLE_INITIAL>
    <CUSTOMER_SURNAME>Bloggs</CUSTOMER_SURNAME>
    <CUSTOMER_ADDRESS_LINE1>10 Main Street</CUSTOMER_ADDRESS_LINE1>
    <CUSTOMER_ADDRESS_LINE2>Athlone</CUSTOMER_ADDRESS_LINE2>
    <CUSTOMER_ADDRESS_LINE3>Co. Westmeath</CUSTOMER_ADDRESS_LINE3>
    <CUSTOMER_ADDRESS_LINE4>Ireland</CUSTOMER_ADDRESS_LINE4>
    <CUSTOMER_TYPE>Personal</CUSTOMER_TYPE>
    <CUSTOMER_RATING>4</CUSTOMER_RATING>
    <COMMENT_LINE1>This is a comment with regard to the above
customer...</COMMENT_LINE1>
    <COMMENT_LINE2>This is a comment with regard to the above
customer...</COMMENT_LINE2>
    <COMMENT_LINE3>This is a comment with regard to the above
customer...</COMMENT_LINE3>
    <COMMENT_LINE4>This is a comment with regard to the above
customer...</COMMENT_LINE4>
    <COMMENT_LINE5>This is a comment with regard to the above
customer...</COMMENT_LINE5>
    <COMMENT_LINE6>This is a comment with regard to the above
customer...</COMMENT_LINE6>
    <COMMENT_LINE7>This is a comment with regard to the above
customer...</COMMENT_LINE7>
    <COMMENT_LINE8>This is a comment with regard to the above
customer...</COMMENT_LINE8>
    <COMMENT_LINE9>This is a comment with regard to the above
customer...</COMMENT_LINE9>
    <COMMENT_LINE10>This is a comment with regard to the above
customer...</COMMENT_LINE10>
  </ServiceView>
</Request>
```

Listing 2 – POX Internet-based *ServiceAdd* Request

```
<?xml version="1.0" encoding="UTF-8"?>
<Request>
  <Transaction>ServiceAdd</Transaction>
  <ServiceAdd>
    <BRANCH_CODE>123456</BRANCH_CODE>
    <ACCOUNT_NUMBER>12345678</ACCOUNT_NUMBER>
    <CUSTOMER_TITLE>Mr.</CUSTOMER_TITLE>
    <CUSTOMER_FIRSTNAME>Joseph</CUSTOMER_FIRSTNAME>
    <CUSTOMER_MIDDLE_INITIAL>P</CUSTOMER_MIDDLE_INITIAL>
    <CUSTOMER_SURNAME>Bloggs</CUSTOMER_SURNAME>
    <CUSTOMER_ADDRESS_LINE1>10 Main Street</CUSTOMER_ADDRESS_LINE1>
    <CUSTOMER_ADDRESS_LINE2>Athlone</CUSTOMER_ADDRESS_LINE2>
    <CUSTOMER_ADDRESS_LINE3>Co. Westmeath</CUSTOMER_ADDRESS_LINE3>
    <CUSTOMER_ADDRESS_LINE4>Ireland</CUSTOMER_ADDRESS_LINE4>
    <CUSTOMER_TYPE>Personal</CUSTOMER_TYPE>
    <CUSTOMER_RATING>4</CUSTOMER_RATING>
    <COMMENT_LINE1>This is a comment with regard to the above
customer...</COMMENT_LINE1>
    <COMMENT_LINE2>This is a comment with regard to the above
customer...</COMMENT_LINE2>
    <COMMENT_LINE3>This is a comment with regard to the above
customer...</COMMENT_LINE3>
    <COMMENT_LINE4>This is a comment with regard to the above
customer...</COMMENT_LINE4>
    <COMMENT_LINE5>This is a comment with regard to the above
customer...</COMMENT_LINE5>
    <COMMENT_LINE6>This is a comment with regard to the above
customer...</COMMENT_LINE6>
    <COMMENT_LINE7>This is a comment with regard to the above
customer...</COMMENT_LINE7>
    <COMMENT_LINE8>This is a comment with regard to the above
customer...</COMMENT_LINE8>
    <COMMENT_LINE9>This is a comment with regard to the above
customer...</COMMENT_LINE9>
    <COMMENT_LINE10>This is a comment with regard to the above
customer...</COMMENT_LINE10>
  </ServiceAdd>
</Request>
```

Listing 3 – POX Internet-based *ServiceDelete* Request

```
<?xml version="1.0" encoding="UTF-8"?>
<Request>
  <Transaction>ServiceDelete</Transaction>
  <ServiceDelete>
    <BRANCH_CODE>123456</BRANCH_CODE>
    <ACCOUNT_NUMBER>12345678</ACCOUNT_NUMBER>
    <CUSTOMER_TITLE>Mr.</CUSTOMER_TITLE>
    <CUSTOMER_FIRSTNAME>Joseph</CUSTOMER_FIRSTNAME>
    <CUSTOMER_MIDDLE_INITIAL>P</CUSTOMER_MIDDLE_INITIAL>
    <CUSTOMER_SURNAME>Bloggs</CUSTOMER_SURNAME>
    <CUSTOMER_ADDRESS_LINE1>10 Main Street</CUSTOMER_ADDRESS_LINE1>
    <CUSTOMER_ADDRESS_LINE2>Athlone</CUSTOMER_ADDRESS_LINE2>
    <CUSTOMER_ADDRESS_LINE3>Co. Westmeath</CUSTOMER_ADDRESS_LINE3>
    <CUSTOMER_ADDRESS_LINE4>Ireland</CUSTOMER_ADDRESS_LINE4>
    <CUSTOMER_TYPE>Personal</CUSTOMER_TYPE>
    <CUSTOMER_RATING>4</CUSTOMER_RATING>
    <COMMENT_LINE1>This is a comment with regard to the above
customer...</COMMENT_LINE1>
    <COMMENT_LINE2>This is a comment with regard to the above
customer...</COMMENT_LINE2>
    <COMMENT_LINE3>This is a comment with regard to the above
customer...</COMMENT_LINE3>
    <COMMENT_LINE4>This is a comment with regard to the above
customer...</COMMENT_LINE4>
    <COMMENT_LINE5>This is a comment with regard to the above
customer...</COMMENT_LINE5>
    <COMMENT_LINE6>This is a comment with regard to the above
customer...</COMMENT_LINE6>
    <COMMENT_LINE7>This is a comment with regard to the above
customer...</COMMENT_LINE7>
    <COMMENT_LINE8>This is a comment with regard to the above
customer...</COMMENT_LINE8>
    <COMMENT_LINE9>This is a comment with regard to the above
customer...</COMMENT_LINE9>
    <COMMENT_LINE10>This is a comment with regard to the above
customer...</COMMENT_LINE10>
  </ServiceDelete>
</Request>
```


Listing 4 – POX Internet-based *ServiceUpdate* Request

```
<?xml version="1.0" encoding="UTF-8"?>
<Request>
  <Transaction>ServiceUpdate</Transaction>
  <ServiceUpdate>
    <BRANCH_CODE>123456</BRANCH_CODE>
    <ACCOUNT_NUMBER>12345678</ACCOUNT_NUMBER>
    <CUSTOMER_TITLE>Mr.</CUSTOMER_TITLE>
    <CUSTOMER_FIRSTNAME>Joseph</CUSTOMER_FIRSTNAME>
    <CUSTOMER_MIDDLE_INITIAL>P</CUSTOMER_MIDDLE_INITIAL>
    <CUSTOMER_SURNAME>Bloggs</CUSTOMER_SURNAME>
    <CUSTOMER_ADDRESS_LINE1>10 Main Street</CUSTOMER_ADDRESS_LINE1>
    <CUSTOMER_ADDRESS_LINE2>Athlone</CUSTOMER_ADDRESS_LINE2>
    <CUSTOMER_ADDRESS_LINE3>Co. Westmeath</CUSTOMER_ADDRESS_LINE3>
    <CUSTOMER_ADDRESS_LINE4>Ireland</CUSTOMER_ADDRESS_LINE4>
    <CUSTOMER_TYPE>Personal</CUSTOMER_TYPE>
    <CUSTOMER_RATING>4</CUSTOMER_RATING>
    <COMMENT_LINE1>This is a comment with regard to the above
customer...</COMMENT_LINE1>
    <COMMENT_LINE2>This is a comment with regard to the above
customer...</COMMENT_LINE2>
    <COMMENT_LINE3>This is a comment with regard to the above
customer...</COMMENT_LINE3>
    <COMMENT_LINE4>This is a comment with regard to the above
customer...</COMMENT_LINE4>
    <COMMENT_LINE5>This is a comment with regard to the above
customer...</COMMENT_LINE5>
    <COMMENT_LINE6>This is a comment with regard to the above
customer...</COMMENT_LINE6>
    <COMMENT_LINE7>This is a comment with regard to the above
customer...</COMMENT_LINE7>
    <COMMENT_LINE8>This is a comment with regard to the above
customer...</COMMENT_LINE8>
    <COMMENT_LINE9>This is a comment with regard to the above
customer...</COMMENT_LINE9>
    <COMMENT_LINE10>This is a comment with regard to the above
customer...</COMMENT_LINE10>
  </ServiceUpdate>
</Request>
```

Listing 5 – SOAP Internet-based *ServiceView* Request

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
  <ns2:ServiceView xmlns:ns2="http://BankServicePkg/">
    <branchCode>123456</branchCode>
    <accountNo>12345678</accountNo>
    <CUSTOMER_TITLE>Mr.</CUSTOMER_TITLE>
    <CUSTOMER_FIRSTNAME>Joseph</CUSTOMER_FIRSTNAME>
    <CUSTOMER_MIDDLE_INITIAL>P</CUSTOMER_MIDDLE_INITIAL>
    <CUSTOMER_SURNAME>Bloggs</CUSTOMER_SURNAME>
    <CUSTOMER_ADDRESS_LINE1>10 Main Street</CUSTOMER_ADDRESS_LINE1>
    <CUSTOMER_ADDRESS_LINE2>Athlone</CUSTOMER_ADDRESS_LINE2>
    <CUSTOMER_ADDRESS_LINE3>Co. Westmeath</CUSTOMER_ADDRESS_LINE3>
    <CUSTOMER_ADDRESS_LINE4>Ireland</CUSTOMER_ADDRESS_LINE4>
    <CUSTOMER_TYPE>Personal</CUSTOMER_TYPE>
    <CUSTOMER_RATING>4</CUSTOMER_RATING>
    <COMMENT_LINE1>This is a comment with regard to the above
customer...</COMMENT_LINE1>
    <COMMENT_LINE2>This is a comment with regard to the above
customer...</COMMENT_LINE2>
    <COMMENT_LINE3>This is a comment with regard to the above
customer...</COMMENT_LINE3>
    <COMMENT_LINE4>This is a comment with regard to the above
customer...</COMMENT_LINE4>
    <COMMENT_LINE5>This is a comment with regard to the above
customer...</COMMENT_LINE5>
    <COMMENT_LINE6>This is a comment with regard to the above
customer...</COMMENT_LINE6>
    <COMMENT_LINE7>This is a comment with regard to the above
customer...</COMMENT_LINE7>
    <COMMENT_LINE8>This is a comment with regard to the above
customer...</COMMENT_LINE8>
    <COMMENT_LINE9>This is a comment with regard to the above
customer...</COMMENT_LINE9>
    <COMMENT_LINE10>This is a comment with regard to the above
customer...</COMMENT_LINE10>
  </ns2:ServiceView>
</S:Body>
</S:Envelope>
```

Listing 6 – SOAP Internet-based *ServiceAdd* Request

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
  <ns2:ServiceAdd xmlns:ns2="http://BankServicePkg/">
    <branchCode>123456</branchCode>
    <accountNo>12345678</accountNo>
    <CUSTOMER_TITLE>Mr.</CUSTOMER_TITLE>
    <CUSTOMER_FIRSTNAME>Joseph</CUSTOMER_FIRSTNAME>
    <CUSTOMER_MIDDLE_INITIAL>P</CUSTOMER_MIDDLE_INITIAL>
    <CUSTOMER_SURNAME>Bloggs</CUSTOMER_SURNAME>
    <CUSTOMER_ADDRESS_LINE1>10 Main Street</CUSTOMER_ADDRESS_LINE1>
    <CUSTOMER_ADDRESS_LINE2>Athlone</CUSTOMER_ADDRESS_LINE2>
    <CUSTOMER_ADDRESS_LINE3>Co. Westmeath</CUSTOMER_ADDRESS_LINE3>
    <CUSTOMER_ADDRESS_LINE4>Ireland</CUSTOMER_ADDRESS_LINE4>
    <CUSTOMER_TYPE>Personal</CUSTOMER_TYPE>
    <CUSTOMER_RATING>4</CUSTOMER_RATING>
    <COMMENT_LINE1>This is a comment with regard to the above
customer...</COMMENT_LINE1>
    <COMMENT_LINE2>This is a comment with regard to the above
customer...</COMMENT_LINE2>
    <COMMENT_LINE3>This is a comment with regard to the above
customer...</COMMENT_LINE3>
    <COMMENT_LINE4>This is a comment with regard to the above
customer...</COMMENT_LINE4>
    <COMMENT_LINE5>This is a comment with regard to the above
customer...</COMMENT_LINE5>
    <COMMENT_LINE6>This is a comment with regard to the above
customer...</COMMENT_LINE6>
    <COMMENT_LINE7>This is a comment with regard to the above
customer...</COMMENT_LINE7>
    <COMMENT_LINE8>This is a comment with regard to the above
customer...</COMMENT_LINE8>
    <COMMENT_LINE9>This is a comment with regard to the above
customer...</COMMENT_LINE9>
    <COMMENT_LINE10>This is a comment with regard to the above
customer...</COMMENT_LINE10>
  </ns2:ServiceAdd>
</S:Body>
</S:Envelope>
```

Listing 7 – SOAP Internet-based *ServiceDelete* Request

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
  <ns2:ServiceDelete xmlns:ns2="http://BankServicePkg/">
    <branchCode>123456</branchCode>
    <accountNo>12345678</accountNo>
    <CUSTOMER_TITLE>Mr.</CUSTOMER_TITLE>
    <CUSTOMER_FIRSTNAME>Joseph</CUSTOMER_FIRSTNAME>
    <CUSTOMER_MIDDLE_INITIAL>P</CUSTOMER_MIDDLE_INITIAL>
    <CUSTOMER_SURNAME>Bloggs</CUSTOMER_SURNAME>
    <CUSTOMER_ADDRESS_LINE1>10 Main Street</CUSTOMER_ADDRESS_LINE1>
    <CUSTOMER_ADDRESS_LINE2>Athlone</CUSTOMER_ADDRESS_LINE2>
    <CUSTOMER_ADDRESS_LINE3>Co. Westmeath</CUSTOMER_ADDRESS_LINE3>
    <CUSTOMER_ADDRESS_LINE4>Ireland</CUSTOMER_ADDRESS_LINE4>
    <CUSTOMER_TYPE>Personal</CUSTOMER_TYPE>
    <CUSTOMER_RATING>4</CUSTOMER_RATING>
    <COMMENT_LINE1>This is a comment with regard to the above
customer...</COMMENT_LINE1>
    <COMMENT_LINE2>This is a comment with regard to the above
customer...</COMMENT_LINE2>
    <COMMENT_LINE3>This is a comment with regard to the above
customer...</COMMENT_LINE3>
    <COMMENT_LINE4>This is a comment with regard to the above
customer...</COMMENT_LINE4>
    <COMMENT_LINE5>This is a comment with regard to the above
customer...</COMMENT_LINE5>
    <COMMENT_LINE6>This is a comment with regard to the above
customer...</COMMENT_LINE6>
    <COMMENT_LINE7>This is a comment with regard to the above
customer...</COMMENT_LINE7>
    <COMMENT_LINE8>This is a comment with regard to the above
customer...</COMMENT_LINE8>
    <COMMENT_LINE9>This is a comment with regard to the above
customer...</COMMENT_LINE9>
    <COMMENT_LINE10>This is a comment with regard to the above
customer...</COMMENT_LINE10>
  </ns2:ServiceDelete>
</S:Body>
</S:Envelope>
```

Listing 8 – SOAP Internet-based *ServiceUpdate* Request

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
  <ns2:ServiceUpdate xmlns:ns2="http://BankServicePkg/">
    <branchCode>123456</branchCode>
    <accountNo>12345678</accountNo>
    <CUSTOMER_TITLE>Mr.</CUSTOMER_TITLE>
    <CUSTOMER_FIRSTNAME>Joseph</CUSTOMER_FIRSTNAME>
    <CUSTOMER_MIDDLE_INITIAL>P</CUSTOMER_MIDDLE_INITIAL>
    <CUSTOMER_SURNAME>Bloggs</CUSTOMER_SURNAME>
    <CUSTOMER_ADDRESS_LINE1>10 Main Street</CUSTOMER_ADDRESS_LINE1>
    <CUSTOMER_ADDRESS_LINE2>Athlone</CUSTOMER_ADDRESS_LINE2>
    <CUSTOMER_ADDRESS_LINE3>Co. Westmeath</CUSTOMER_ADDRESS_LINE3>
    <CUSTOMER_ADDRESS_LINE4>Ireland</CUSTOMER_ADDRESS_LINE4>
    <CUSTOMER_TYPE>Personal</CUSTOMER_TYPE>
    <CUSTOMER_RATING>4</CUSTOMER_RATING>
    <COMMENT_LINE1>This is a comment with regard to the above
customer...</COMMENT_LINE1>
    <COMMENT_LINE2>This is a comment with regard to the above
customer...</COMMENT_LINE2>
    <COMMENT_LINE3>This is a comment with regard to the above
customer...</COMMENT_LINE3>
    <COMMENT_LINE4>This is a comment with regard to the above
customer...</COMMENT_LINE4>
    <COMMENT_LINE5>This is a comment with regard to the above
customer...</COMMENT_LINE5>
    <COMMENT_LINE6>This is a comment with regard to the above
customer...</COMMENT_LINE6>
    <COMMENT_LINE7>This is a comment with regard to the above
customer...</COMMENT_LINE7>
    <COMMENT_LINE8>This is a comment with regard to the above
customer...</COMMENT_LINE8>
    <COMMENT_LINE9>This is a comment with regard to the above
customer...</COMMENT_LINE9>
    <COMMENT_LINE10>This is a comment with regard to the above
customer...</COMMENT_LINE10>
  </ns2:ServiceUpdate>
</S:Body>
</S:Envelope>
```

Listing 9 – POX request that involves significant parsing

[illegible]

[illegible]

[illegible]

[illegible]

Listing 10 – GET equivalent of Listing 1

```
"http://109.76.98.126:3050/RESTServer/BankServices/123456/12345678/Mr./Joseph/P/Bloggs/10 Main Street/Athlone/Co.  
Westmeath/Ireland/Personal/4/This is a comment with regard to the above  
customer.../This is a comment with regard to the above customer.../This  
is a comment with regard to the above customer.../This is a comment with  
regard to the above customer.../This is a comment with regard to the  
above customer.../This is a comment with regard to the above  
customer.../This is a comment with regard to the above customer.../This  
is a comment with regard to the above customer.../This is a comment with  
regard to the above customer.../This is a comment with regard to the  
above customer..."
```