A Coordination Approach for Self-Managed Middleware

Edward Curry (B.Sc.)

Department of Information Technology
Faculty of Science

National University of Ireland, Galway

A thesis submitted for the degree of

Doctor of Philosophy

March 2006

Supervisor: Dr. Desmond Chambers
Head of Department: Professor Gerard J. Lyons

Dedication

To my Parents.

Come, my friends. 'Tis not too late to seek a newer world.
- Alfred Tennyson , Ulysses (1842)

Middleware. Reinventing the wheel, so you don't have too!
- John 'Young' Coleman (2005)

Acknowledgements

I am grateful to my supervisor, Dr. Desmond Chambers, for his guidance and advice. It is difficult to overstate my gratitude to the coffee crowd. In particular, I am grateful to Tom Lyons and Enda Ridge. Even though I enjoyed working on this thesis, I could not have coped without some time off. Both Tom and Enda often persuaded me to turn off the computer and have a coffee and chat, or a night out.

My work has benefited from conversations with colleagues, former teachers, and new acquaintances. I am indebted to my student colleagues and friends for providing a stimulating and fun environment in which to learn and grow. I am especially grateful to Ann, Brian, Claire, Mourad, Alan, Dav, Jason, John H., John C., Páraic, Albert, Ted, Catherine, Owen, Noel, Seamus, Niall, Declan, Anto, Seamus, Enda, Mike, Ronan, Weston, and Yan. In particular, I would like to thank Enda, Tom, and Alan for their assistance in formatting, proofreading, and suggesting improvements to my work. I enjoyed the company and support of many others whom I fail to mention here, and I'm thankful for your help and friendship.

I would also like to thank Prof. Roger Needham for his inspirational approach to computer science research. Roger believed that systems should be designed to do useful things for real people. "If there wasn't an industry concerned with making and using computers the subject wouldn't exist. It's not like physics — physics was made by God, but computer science was made by man. It's there because the industry's there". His pragmatic approach to research has been a guiding light for this work.

Thanks are due to three esteemed colleagues who grilled a young PhD student over dinner while watching game 7 of the 2004 Championship Series (Red Sox defeated Yankees, 4-3) in Toronto. For their generous help, I am grateful to Werner Wogels, Steve Vinoski, and Rick Schantz not only did they help me focus on the true contribution of my work, they also created a Red Sox fan.

Lastly, and most importantly, I would like to thank my family. To my sisters Tara, Alison, and my brother Joe, in spite of their mutual incomprehension about what I do, they always believed in me. Finally, I am forever indebted to my father and mother for their inspiration, endless patience, and unconditional love. To them I dedicate this thesis.

Edward Curry
National University of Ireland, Galway.
March 2006

Abstract

A Coordination Approach for Self-Managed Middleware

Edward Curry

Department of Information Technology
National University of Ireland, Galway

A thesis submitted for the degree of Doctor of Philosophy

March 2006

This dissertation investigates the evolution of coordination techniques between self-managed systems within the problem domain of Message-Oriented Middleware (MOM).

The basic goal of autonomic computing is to simplify and automate the management of computing systems, both hardware and software, allowing them to self-manage, without the need for human intervention. Within the software domain, self-management techniques have been utilised to empower a system to automatically self-alter (adapt) to meet their environmental and user needs.

Current self-managed middleware platforms service their environment in an isolated and introverted manner. As they progress towards autonomic middleware, one of the most interesting research challenges facing self-managed middleware platforms is their lack of cooperation and coordination to achieve mutually beneficial outcomes.

The primary hypothesis of this work is that within dynamic operating environments, coordinated interaction between self-managed systems can improve the ability of the individual and collective systems to fulfil performance and autonomy requirements of the environment.

Coordination between next-generation middleware systems will be a vital mechanism needed to meet the challenges within future computing environments. As a step toward this goal, this thesis investigates the benefits of coordination between self-manages middleware platforms. This work explores coordination within the realm of Message-Oriented Middleware (MOM). MOM is an ideal candidate for the study of cooperation as it is an interaction-oriented middleware. In addition, self-management techniques have yet to be applied within the MOM domain, providing an opportunity to investigate their application within this domain.

The main findings of this research are as follows. The coordination of self-managed systems can improve the ability of the individual and collective systems to fulfil performance and autonomy requirements of the environment. Secondly, the introduction of self-management techniques within MOM systems increases their performance within dynamic operating environments.

List of Publications

The following publications resulted from this thesis.

1. E. Curry, “Introducing Reflective Techniques to Message Hierarchies”, presented at Doctoral Symposium at 17th European Conference on Object-Oriented Programming (ECOOP 2003), Darmstadt, Germany, 2003.

2. E. Curry, D. Chambers, and G. Lyons, “Reflective Channel Hierarchies”, presented at 2nd Workshop on Reflective and Adaptive Middleware, Middleware 2003, Rio de Janeiro, Brazil, 2003.

3. E. Curry, D. Chambers, and G. Lyons, “Could Message Hierarchies Contemplate?”, presented at 17th European Conference on Object-Oriented Programming (ECOOP 2003), Darmstadt, Germany, 2003. (poster)

4. E. Curry, D. Chambers, and G. Lyons, “Extending Message-Oriented Middleware using Interception”, presented at Third International Workshop on Distributed Event-Based Systems (DEBS ’04), ICSE ’04, Edinburgh, Scotland, UK, 2004.

5. E. Curry, D. Chambers, and G. Lyons, “ARMAdA: Creating a Reflective Fellowship (Options for Interoperability)”, presented at 3rd Workshop on Adaptive and Reflective Middleware, Middleware 2004, Toronto, Canada, 2004.

Contents

Dedication **i**

Acknowledgements **iii**

Abstract **iv**

List of Publications **v**

List of Illustrations **xiii**

List of Tables **xvii**

I Background **1**

1 Introduction **2**

1.1 Motivation and Problem Domain 2
1.2 Motivational Scenario 4
1.3 Research Hypothesis 6
1.4 Research Methodology 6
1.5 Principal Contributions 6
1.6 Thesis Organisation 7
1.7 Summary of Conclusions 9

2 Adaptive and Reflective Middleware Essentials **11**

2.1 Introduction 11
2.2 Adaptive Middleware 12
2.3 Reflective Middleware 12
2.4 Are Adaptive and Reflective Techniques the Same? 13
2.5 Triggers of Adaptive and Reflective Behaviour 14
2.6 Adaptive and Reflective Techniques 14
2.6.1 Structural Reflection (Programmatic) 14
2.6.2 Behavioural Reflection 15
2.6.3 Architectural Reflection 15
2.6.4 Synchronous Reflection 15
2.6.5 Asynchronous Reflection 16
2.7 Meta-levels 16

2.7.1 Operation Overview 17
2.7.2 In-Line and Out-of-Line Execution 18
2.7.3 Concern Separation 18
2.7.4 Performance 18
2.7.5 Openness to Coordination 19
2.8 Current Reflective Research 20
2.8.1 mChaRM 20
2.8.2 QuO 21
2.8.3 OpenCOM & Open ORB 23
2.8.4 dynamicTAO (UIC/2K) 25
2.8.5 K-Components 28
2.8.6 Other Reviewed Systems 31
2.9 Comparison of Reviewed Systems 33
2.9.1 Coordination Capability 33
2.9.2 Interaction Protocol 34
2.9.3 Meta-Level Access Capabilities 35
2.9.4 Review Summary 37
2.10 Summary 37

3 Rudimentary Message-Oriented Middleware **38**
3.1 Introduction 38
3.2 Interaction Models 39
3.2.1 Synchronous Communication 39
3.2.2 Asynchronous Communication 39
3.3 Introduction to the Remote Procedure Call (RPC) 40
3.3.1 Coupling 40
3.3.2 Reliability 41
3.3.3 Scalability 41
3.3.4 Availability 41
3.4 Introduction to Message-Oriented Middleware (MOM) 42
3.4.1 Coupling 42
3.4.2 Reliability 43
3.4.3 Scalability 43
3.4.4 Availability 43
3.5 When to use MOM or RPC 43
3.6 Message Queues 44
3.7 Messaging Models 46
3.7.1 Point-to-Point 46
3.7.2 Publish/Subscribe 47
3.7.3 Comparison of Messaging Models 49
3.8 Message Filtering 49
3.8.1 Covering & Merging 50
3.9 Java Message Service 51
3.9.1 Programming using the JMS API 52

3.10 Current MOM Platforms . . . 55
3.10.1 CORBA Event Service & Notification Service . . . 55
3.10.2 TIBCO Rendezvous . . . 56
3.10.3 OpenJMS . . . 57
3.10.4 ActiveMQ . . . 58
3.10.5 SonicMQ . . . 59
3.10.6 SIENA . . . 61
3.10.7 REBECA . . . 62
3.10.8 Hermes . . . 63
3.10.9 WebSphere MQ (formerly MQSeries) . . . 64
3.10.10 Other Reviewed Systems . . . 65
3.11 Comparison of Reviewed Systems . . . 67
3.11.1 Message Capabilities . . . 68
3.11.2 Administration Capabilities . . . 69
3.12 Summary . . . 70

II Contribution 72

4 Meta-level Coordination 73
4.1 Introduction . . . 73
4.2 Motivational Scenario . . . 74
4.3 Opening the Meta-Level . . . 76
4.4 A Vision of Coordination and Cooperation . . . 77
4.5 Protocol Prerequisites . . . 78
4.6 Open Meta-level Interaction Protocol (OMIP) . . . 78
4.6.1 Interaction Commands . . . 79
4.6.2 OMIP Walkthrough . . . 80
4.6.3 Domain Specific Languages . . . 81
4.6.4 OMIP Message Definition Format . . . 83
4.6.5 ARMAdA – A Sample Participant Architecture . . . 85
4.7 Summary . . . 85

5 Definition of GenerIc Self-management for Message-Oriented middleware 87
5.1 Introduction . . . 87
5.2 Identification of Generic MOM Elements . . . 87
5.2.1 MOM Participants . . . 88
5.2.2 MOM Behaviour Identification . . . 88
5.2.3 MOM State Identification . . . 91
5.3 Designing a Portable Meta-Level . . . 93
5.3.1 The Role of a Meta-Level . . . 93
5.3.2 Monolithic Meta-Level Design . . . 94
5.3.3 The Model-View-Controller Design Pattern . . . 95
5.3.4 Meta-State Analysis Realisation (M-SAR) Design Pattern . . . 96
5.3.5 Benefits of a Separated Meta-Model Design . . . 97

5.4 GISMO: GenerIc Self-management for Message-Oriented middleware 98
5.4.1 Client and Provider Roles 98
5.4.2 Destination Meta-Model 99
5.4.3 Subscription Meta-Model 102
5.4.4 Interception Meta-Model 104
5.4.5 Meta-Level Event-Model 107
5.4.6 Reflective Engine 108
5.4.7 Extending the Meta-Level 110
5.5 MOM-DSL: Opening GISMO 111
5.5.1 Message Exchange Infrastructure 111
5.5.2 State Structure 112
5.5.3 Available Actions 113
5.5.4 Capabilities Request 114
5.6 Example Interactions 114
5.6.1 Request Capabilities 115
5.6.2 Request Destination State 115
5.6.3 Update Destination 116
5.6.4 Request Filter Analysis 117
5.7 Summary 118

6 Implementation of a GISMO 119
6.1 Introduction 119
6.2 Challenges in System Extension 119
6.3 Options for Extension 119
6.3.1 Interceptor Design Pattern 120
6.3.2 Aspect-Oriented Programming (AOP) 121
6.3.3 Dynamic AOP for Reflective Middleware 122
6.3.4 Multi-Dimensional Separation of Concerns 122
6.3.5 Programmatic Reflection 123
6.3.6 Generative Programming 124
6.3.7 Evaluation 124
6.4 Chameleon 126
6.4.1 Call Capture Proxy 126
6.4.2 Interception 128
6.4.3 Non-Invasive Extension of Functionality 130
6.5 Realisation of a GISMO 132
6.5.1 Destination Realisation 132
6.5.2 Interception Provision 133
6.5.3 Subscription 134
6.5.4 Reflective Engine 134
6.5.5 Event System 135
6.5.6 OMIP Infrastructure 135
6.6 Summary 136

III Evaluation 137

7 Case Study – Coordination-Based Integration 138
7.1 Introduction 138
7.2 Message Infrastructure Coupling 139
7.3 Motivational Scenario 139
7.4 Integration 101 141
7.5 Integration solutions 142
7.5.1 Centralised Content-Based Routing Integration Pattern 142
7.5.2 Decentralised Content-Based Routing Integration Pattern 144
7.6 Achieving Decentralised-CBR 145
7.6.1 MOM-DSL Messages for Decentralised-CBR 146
7.7 Evaluation 146
7.7.1 One-to-One Evaluation 149
7.7.2 Few-to-Many Evaluation 150
7.7.3 Many-to-Few Evaluation 152
7.7.4 Evaluation Summary and Discussion 153
7.8 Summary 153

8 Case Study – Coordinated Self-Managed MOM 155
8.1 Motivational Scenario 155
8.2 Routing Scenarios 156
8.2.1 Single Destination 158
8.2.2 Static Destination Hierarchy 158
8.2.3 Reflective Destination Hierarchy 159
8.3 Creating Reflective Destination Hierarchies 160
8.3.1 Subscription Monitoring Policy - Overview 161
8.3.2 Policy Triggers 161
8.3.3 Analysis Process 161
8.3.4 Adaptation Algorithm 162
8.3.5 Realisation 163
8.3.6 Coordination 163
8.4 Benchmarking Dynamic Environments 163
8.4.1 Dynamic MOM Test Case Walkthrough 164
8.5 Evaluation 165
8.5.1 One-to-One Evaluation 165
8.5.2 Few-to-Many Evaluation 169
8.5.3 Many-to-Few Evaluation 172
8.5.4 Evaluation Summary and Discussion 174
8.6 Summary 176

9 Conclusions 177
9.1 Thesis Summary 177
9.2 Contributions 178
9.2.1 A Self-Managed MOM 178

9.2.2 Coordination between Self-Managed Systems . . . 179
9.2.3 Additional Contributions . . . 179
9.3 Future Research Directions . . . 180
9.3.1 Technology Transfer . . . 180
9.3.2 Research Opportunities . . . 181

IV Appendices 184

A An Extensible MOM Test Suite 185
A.1 Introduction . . . 185
A.2 Test Case Design . . . 185
A.2.1 Messaging Models . . . 186
A.2.2 Producer/Consumer Ratio . . . 186
A.2.3 Configuration . . . 186
A.2.4 Reporting Metrics . . . 187
A.2.5 Timeline . . . 188
A.3 Testbed Configuration . . . 189
A.3.1 Hardware . . . 190
A.3.2 Network . . . 190
A.4 Default Test Cases Setup . . . 190
A.4.1 Test Conditions . . . 190
A.4.2 Desired Metrics . . . 191
A.5 Software – Extensible Mom Test Suite (EMiTS) . . . 191
A.5.1 Architecture . . . 192
A.5.2 Test Drivers . . . 193
A.5.3 Test Case Sequence . . . 194
A.6 Dynamic Environment Simulation . . . 194
A.6.1 Dynamic MOM Test Case Walkthrough . . . 196
A.6.2 Dynamic Report Metric . . . 197
A.7 Summary . . . 197

B XML Schemas 199
B.1 Multimedia-DSL . . . 200
B.1.1 Capability Request Schema . . . 200
B.1.2 Capability Request Example . . . 200
B.1.3 Capability Reply Schema . . . 200
B.1.4 Capability Reply Example . . . 201
B.1.5 Service Request Schema . . . 201
B.1.6 Service Request Example . . . 202
B.1.7 Service Reply Schema . . . 202
B.1.8 Service Reply Example . . . 203
B.1.9 Common Type Schema . . . 203
B.2 MOM-DSL . . . 204
B.2.1 Destination Type Schema . . . 204

B.2.2 Subscription Type Schema 207
B.2.3 Interception Type Schema 208
B.2.4 Reflective Type Schema 209
B.2.5 Event Type Schema 210
B.2.6 Capability Type Schema 211
B.2.7 Common Type Schema 212
B.2.8 Generic MOM-DSL Request/Reply Schema 213

C Additional Results from Chapter 8 **218**
C.1 One-to-One Results 219
C.1.1 3 Producers / 3 Consumers 219
C.1.2 30 Producers / 30 Consumers 220
C.1.3 90 Producers / 90 Consumers 221
C.1.4 150 Producers / 150 Consumers 222
C.1.5 300 Producers / 300 Consumers 223
C.2 Few-to-Many Results 224
C.2.1 3 Producers / 15 Consumers 224
C.2.2 30 Producers / 150 Consumers 225
C.2.3 60 Producers / 300 Consumers 226
C.2.4 90 Producers / 450 Consumers 227
C.3 Many-to-Few Results 228
C.3.1 15 Producers / 3 Consumers 228
C.3.2 150 Producers / 30 Consumers 229
C.3.3 300 Producers / 60 Consumers 230
C.3.4 450 Producers / 90 Consumers 231

References **232**

Illustrations

1.1 A centralised message routing solution 4
1.2 A decentralised message routing solution 5
1.3 The relationships between research contributions 8

2.1 The inter-relationships between reflective techniques 15
2.2 The mChaRM Loci model (from [32]) 21
2.3 A QuO method invocation (from [31]) 22
2.4 The Open ORB meta-architecture (from [21]) 23
2.5 Component framework in Open ORB (from [47]) 24
2.6 dynamicTAO component architecture (from [49]) 26
2.7 Reifying the dynamicTAO structure (adapted from [8]) 27
2.8 A K-Component runtime (from [27]) 29
2.9 Abstract model of interaction between component binding and AMM transfer between K-Component runtimes (from [27]) 30
2.10 An example RAFDA re-distribution transformation (adapted from [52]) 32

3.1 The synchronous interaction model 39
3.2 The asynchronous interaction model 40
3.3 An example remote procedure call deployment 41
3.4 An example Message-Oriented Middleware deployment 42
3.5 The role of a message queue 45
3.6 The point-to-point messaging model 46
3.7 The publish/subscribe messaging model 47
3.8 An automotive destination hierarchy structure 48
3.9 The JMS API programming model (adapted from [79]) 52
3.10 The TIBCO Rendezvous operating environment (from [88]) 57
3.11 The SIENA hierarchical client/server architecture (from [62]) 61
3.12 Acyclic peer-to-peer server architecture in SIENA (from [62]) 62
3.13 Layered networks in Hermes (from [83]) 64

4.1 A non-reflective media broadcast service 74
4.2 A proprietary reflective broadcast service 75
4.3 A coordinated reflective broadcast service 76
4.4 OMIP interaction command hierarchy 79
4.5 OMIP participant interaction sequence 81

4.6 Sample ARMAdA compliant architecture 85

5.1 Participants within a MOM interaction 88
5.2 A monolithic meta-level design 94
5.3 The Model-View-Controller (MVC) design pattern 95
5.4 The Meta-State Analysis Realisation (M-SAR) design pattern 97
5.5 An M-SAR based meta-level design 98
5.6 The GISMO abstract meta-level design 99
5.7 A destination hierarchy example 100
5.8 Reflective policy interface 109
5.9 Policy call sequence within the reflective engine 110
5.10 SonicMQ proprietary state model 111
5.11 MOM-DSL message exchange infrastructure 112
5.12 MOM-DSL command structure 113
5.13 MOM-DSL capability reply structure 114
5.14 Sample destinations used in state request 116

6.1 The POSA interceptor design pattern 120
6.2 MOM call sequence 126
6.3 Notification call sequence 126
6.4 Interception call sequence 127
6.5 JMS API client/provider interaction sequence 127
6.6 JMS API client/provider captured interaction sequence 128
6.7 Chameleon interception architecture 129
6.8 Centralised message transformation 131
6.9 Decentralised message transformation 131
6.10 The GISMO implementation model 132
6.11 Multiple base-level realisations 133
6.12 Subscription meta-model architecture 134
6.13 GISMO reflective engine architecture 134
6.14 Event system architecture 135

7.1 A sample deployment scenario 140
7.2 Hard-coded integration 141
7.3 Centralised content-based routing integration pattern 143
7.4 Decentralised content-based routing integration pattern 144
7.5 D-CBR implemented within a closed self-managed MOM 145
7.6 D-CBR implemented within an open self-managed MOM 146
7.7 Benchmark results integration patterns within the one-to-one test cases 149
(a) 1 Sender / 1 Queue / 1 Receiver 149
(b) 50 Senders / 50 Queues / 50 Receivers 149
(c) 250 Senders / 250 Queues / 250 Receivers 149
(d) 400 Senders / 400 Queues / 400 Receivers 149
7.8 Benchmark results for integration patterns within the few-to-many test cases . . . 151
(a) 1 Sender / 1 Queue / 5 Receivers 151

(b) 50 Senders / 50 Queues / 250 Receivers . . . 151
(c) 100 Senders / 100 Queues / 500 Receivers . . . 151
7.9 Benchmark results for integration patterns within the many-to-few test cases . . . 152
(a) 5 Sender / 1 Queue / 1 Receivers . . . 152
(b) 250 Senders / 50 Queues / 50 Receivers . . . 152
(c) 500 Senders / 100 Queues / 100 Receivers . . . 152

8.1 Venn diagram of relationships between interest groups . . . 157
8.2 Single destination routing scenario . . . 158
8.3 Static destination hierarchy routing scenario . . . 159
8.4 Reflective destination hierarchy routing scenario . . . 160
8.5 Reflective destination hierarchy adaptation algorithm . . . 162
8.6 Dynamic test case timeline . . . 165
8.7 Result of test case OtO-1 (3 publishers and 3 subscribers) . . . 165
8.8 Result of test case OtO-2 (30 publishers and 30 subscribers) . . . 167
8.9 Result of test case OtO-3 (90 publishers and 90 subscribers) . . . 167
8.10 Result of test case OtO-4 (150 publishers and 150 subscribers) . . . 168
8.11 Result of test case OtO-5 (300 publishers and 300 subscribers) . . . 168
8.12 Result of test case FtM-1 (3 publishers and 15 subscribers) . . . 169
8.13 Result of test case FtM-2 (30 publishers and 150 subscribers) . . . 170
8.14 Result of test case FtM-3 (60 publishers and 300 subscribers) . . . 170
8.15 Result of test case FtM-4 (90 publishers and 450 subscribers) . . . 171
8.16 Result of test case MtF-1 (15 publishers and 3 subscribers) . . . 172
8.17 Result of test case MtF-2 (150 publishers and 30 subscribers) . . . 172
8.18 Result of test case MtF-3 (300 publishers and 60 subscribers) . . . 173
8.19 Result of test case MtF-4 (450 publishers and 90 subscribers) . . . 173

A.1 Test case timeline . . . 188
A.2 The testbed deployment . . . 189
A.3 EMiTS architecture . . . 192
A.4 EMiTS test driver interfaces . . . 194
(a) Producer interface . . . 194
(b) Consumer interface . . . 194
A.5 EMiTS test case execution sequence . . . 195
A.6 Dynamic test case timeline . . . 197
A.7 Sample trend result for dynamic test case . . . 197

C.1 One-to-one 3 / 3 . . . 219
(a) Static hierarchy . . . 219
(b) Reflective hierarchy . . . 219
(c) Static hierarchy vs. reflective hierarchy . . . 219
C.2 One-to-one 30 / 30 . . . 220
(a) Static hierarchy . . . 220
(b) Reflective hierarchy . . . 220
(c) Static hierarchy vs. reflective hierarchy . . . 220

C.3 One-to-one 90 / 90 . . . 221
(a) Static hierarchy . . . 221
(b) Reflective hierarchy . . . 221
(c) Static hierarchy vs. reflective hierarchy . . . 221
C.4 One-to-one 150 / 150 . . . 222
(a) Static hierarchy . . . 222
(b) Reflective hierarchy . . . 222
(c) Static hierarchy vs. reflective hierarchy . . . 222
C.5 One-to-one 300 / 300 . . . 223
(a) Static hierarchy . . . 223
(b) Reflective hierarchy . . . 223
(c) Static hierarchy vs. reflective hierarchy . . . 223
C.6 Few-to-many 3 / 15 . . . 224
(a) Static hierarchy . . . 224
(b) Reflective hierarchy . . . 224
(c) Static hierarchy vs. reflective hierarchy . . . 224
C.7 Few-to-many 30 / 150 . . . 225
(a) Static hierarchy . . . 225
(b) Reflective hierarchy . . . 225
(c) Static hierarchy vs. reflective hierarchy . . . 225
C.8 Few-to-many 60 / 300 . . . 226
(a) Static hierarchy . . . 226
(b) Reflective hierarchy . . . 226
(c) Static hierarchy vs. reflective hierarchy . . . 226
C.9 Few-to-many 90 / 450 . . . 227
(a) Static hierarchy . . . 227
(b) Reflective hierarchy . . . 227
(c) Static hierarchy vs. reflective hierarchy . . . 227
C.10 Many-to-few 15 / 3 . . . 228
(a) Static hierarchy . . . 228
(b) Reflective hierarchy . . . 228
(c) Static hierarchy vs. reflective hierarchy . . . 228
C.11 Many-to-few 150 / 30 . . . 229
(a) Static hierarchy . . . 229
(b) Reflective hierarchy . . . 229
(c) Static hierarchy vs. reflective hierarchy . . . 229
C.12 Many-to-few 300 / 60 . . . 230
(a) Static hierarchy . . . 230
(b) Reflective hierarchy . . . 230
(c) Static hierarchy vs. reflective hierarchy . . . 230
C.13 Many-to-few 450 / 90 . . . 231
(a) Static hierarchy . . . 231
(b) Reflective hierarchy . . . 231
(c) Static hierarchy vs. reflective hierarchy . . . 231

Tables

2.1 Comparison of coordinated behaviour within reviewed systems . . . 34
2.2 Comparison of interaction protocols within reviewed systems . . . 35
2.3 Comparison of meta-level access capabilities within reviewed systems . . . 36

3.1 Message queue formats . . . 45
3.2 Message filtering types . . . 50
3.3 JMS acknowledgement modes . . . 54
3.4 JMS message types . . . 55
3.5 Comparison of MOM messaging capabilities within reviewed systems . . . 68
3.6 Comparison of MOM administration capabilities within reviewed systems . . . 70

4.1 OMIP interaction commands . . . 80
4.2 Sample multimedia domain specific language . . . 82
4.3 Sample multimedia service definition in Multimedia-DSL . . . 82
4.4 Sample security domain specific language . . . 83
4.5 Sample security service definition in Security-DSL . . . 83
4.6 Multimedia DSL sample interaction command definitions . . . 84

5.1 General MOM API interface . . . 89
5.2 Survey of MOM messaging capabilities . . . 89
5.3 Survey of MOM administration capabilities . . . 91
5.4 Survey of MOM messaging state . . . 92
5.5 Survey of MOM administration state . . . 93
5.6 Interception points within a MOM base-level . . . 105
5.7 Event types within the GISMO event model . . . 107
5.8 Reflective locations within GISMO . . . 109
5.9 GISMO interceptor state expressed in MOM-DSL . . . 113
5.10 Example MOM-DSL capability request . . . 115
5.11 Example MOM-DSL capability reply . . . 115
5.12 Example MOM-DSL destination state request . . . 115
5.13 Example MOM-DSL destination state reply . . . 116
5.14 Example MOM-DSL destination update request . . . 117
5.15 Example MOM-DSL destination update reply . . . 117
5.16 Example MOM-DSL filter analysis request . . . 117
5.17 Example MOM-DSL filter analysis reply . . . 118

6.1 Summary of extension techniques 124

7.1 Sample destination state model for deployment scenario 147
7.2 Coordination-based integration case study benchmark results 148

8.1 A sample movie service message structure 155
8.2 Summary of message delivery relationships between interest groups 157
8.3 Coordinated self-management case study benchmark test cases 166
8.4 Message receive throughput comparisons for the one-to-one set of test cases 169
8.5 Message receive throughput comparisons for the few-to-many set of test cases . . . 171
8.6 Message receive throughput comparisons for the many-to-few set of test cases . . . 174
8.7 Summary of case study throughput analysis 175

A.1 The effects of message producer/consumer ratios within benchmarks 187
A.2 Possible benchmark reporting metrics 187
A.3 Testbed hardware and software specifications 190
A.4 Client connection configuration settings 193
A.5 General test case 193
A.6 Mandatory test driver settings 194
A.7 Default test driver settings 195

Part I

Background

Chapter 1

Introduction

This thesis is concerned with the evolution of coordination between self-managed systems.

1.1 Motivation and Problem Domain

The vision of future computing initiatives such as ubiquitous and pervasive computing, large-scale distribution, and on-demand computing will foster unpredictable and complex environments with challenging demands [1, 2, 3]. Next-generation systems will require flexible system infrastructures that can adapt to both dynamic changes in operational requirements and environmental conditions, while providing predictable behaviour in areas such as throughput, scalability, dependability, and security. This increase in complexity of an already complex software development process will only add to the high rates of project failure [4, 5]. Successful projects once deployed will require skilled administration personnel to install, configure, maintain, and provide 24/7 support.

In order to meet these challenges head-on, computing systems will need to be more self-sufficient. IBM's vision of autonomic computing [6] is an analogy with the human autonomic nervous system; this biological system relieves the conscious brain of the burden of having to deal with low-level routine bodily functions such as muscle use, cardiac muscle use (respiration) and gland activity. An autonomic computing system would relieve the burden of low-level functions such as installation, configuration, dependency management, performance optimisation management, and routine maintenance from their conscious brain: the system administrators.

The basic goal of autonomic computing is to simplify and automate the management of computing systems, both hardware and software, allowing them to self-manage, without the need for human intervention. Four fundamental characteristics are needed by an autonomic system to be self-managing:

- *Self-Configuring* - The system must adapt automatically to its operating environment. Hardware and software platforms must posses a self-representation of their abilities and self-configure with regard to their environment.

- *Self-Healing* - Systems must be able to diagnose and solve service interruptions. For a system to be self-healing, it must be able to recognise a failure and isolate it, thus shielding the rest of the system from its erroneous activity. It then must be capable of recovering transparently from failure by fixing or replacing the section of the system that is responsible for the error.

- *Self-Optimising* - The system must constantly evaluate potential optimisations. Through self-monitoring and self-configuration, the system should self-optimise to efficiently maximise resources to best meet the needs of its environment and end-users.
- *Self-Protecting* - These systems must anticipate a potential attack, detect when an attack is underway, identify the type of attack (for example, denial-of-service or unauthorised access), and use appropriate countermeasures to defeat or at least nullify the attack.

The common theme shared by all of these characteristics is the ability to handle functionality that has been traditionally the responsibility of a human system administrator.

Middleware facilitates the development of large software systems by relieving the burden on the applications developer of writing a number of complex infrastructure services needed by the system; such services include persistence, distribution, transactions, load balancing, and clustering. Within the software domain, adaptive and reflective techniques have been utilised to empower systems to automatically self-alter (adapt) to meet their environmental and user needs. Adaptive and reflective techniques are currently used to enhance a number of middleware services including multimedia [7], security [8, 9], transactions [10], and fault-tolerance [11] and have been noted as a key emerging paradigm for the development of dynamic next-generation middleware platforms [12].

One of the key mechanisms used in the construction of an reflective self-managed system is a meta-level, sometimes referred to as a meta-space, or meta-model. Systems built using this technique are separated into two levels, a base- and meta-level. The base-level provides system functionality, and the meta-level contains the policies and strategies for the behaviour of the system. The inspection and alteration of the meta-level allows for changes in the system's behaviour. Middleware platforms built using a meta-level with reflective techniques exhibit a number of the fundamental characteristics needed by autonomic systems and provide an ideal foundation for the construction of autonomic middleware platforms.

Current adaptive and reflective middleware platforms service their environment in an isolated and introverted manner. As they progress towards autonomic middleware, one of the most interesting research challenges facing adaptive and reflective middleware platforms is their lack of cooperation and coordination to achieve mutually beneficial outcomes. John Donne, perhaps the greatest of the metaphysical poets, said 'No man is an island' [13]; likewise no adaptive or reflective middleware platform, service or component is an island. Each must be aware of both the individual consequences and group consequences of its actions.

Given the challenges that await middleware platforms in future computing environments, coordination self-management between next-generation middleware systems will be a vital mechanism needed to meet these challenges head-on. As a step toward this goal, this thesis investigates the benefits of coordination between adaptive and reflective middleware platforms. This work explores coordination within the realm of Message-Oriented Middleware (MOM). MOM is an ideal candidate for the study of coordination as it is an interaction-oriented middleware. In addition, self-management techniques have yet to be applied within the MOM domain, providing an opportunity to investigate their application within this domain.

1.2 Motivational Scenario

As a means to direct debate, a motivational scenario is provided to act as a reference point for discussion. The chosen scenario replicates a common messaging requirement within the MOM domain. In the scenario, illustrated in Figure 1.1, the application on the left produces messages for the consumers on the right; messages are delivered using the MOM. Messages are sent to destinations based on a number of dynamic routing rules, which may be subject to frequent changes. The typical messaging solution used to meet these messaging requirements utilises a centralised message router.

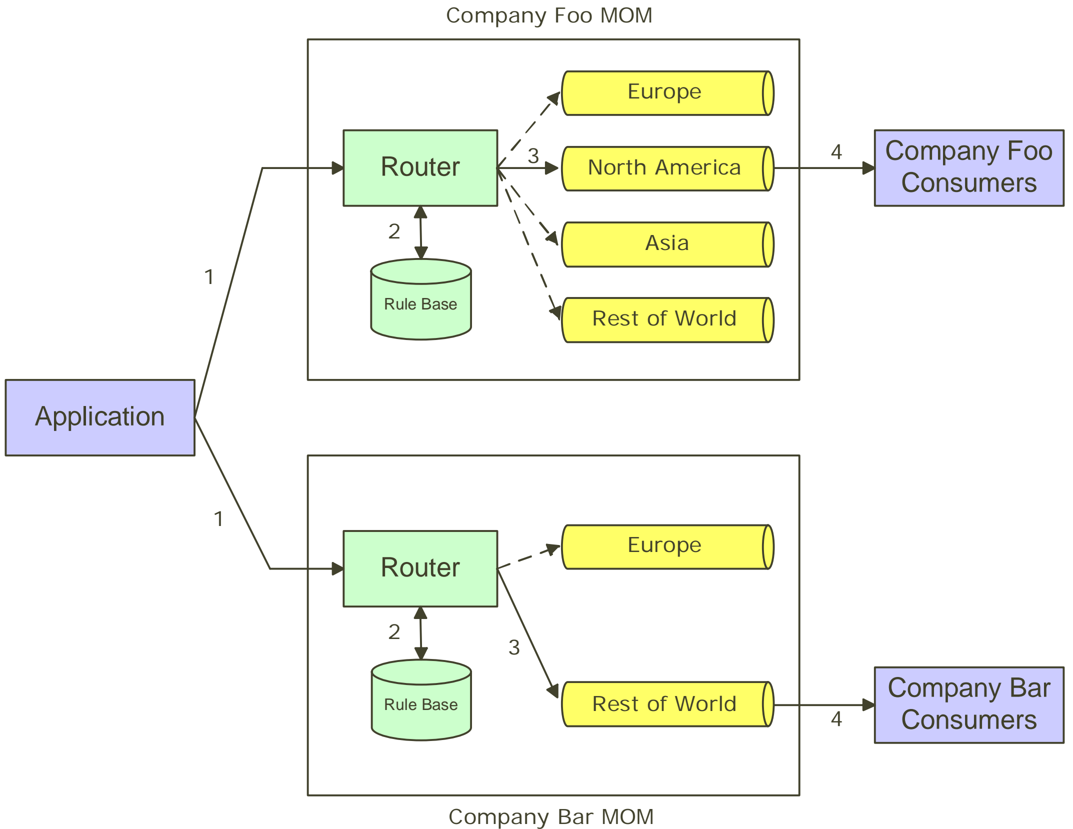


Figure 1.1 *A centralised message routing solution*

This solution, illustrated in Figure 1.1, is comprised of the four steps:

1. Application produces the message and sends it to the router
2. Router evaluates the message against its rule-base to match it to relevant destination(s) based on its content
3. Router forwards the message to the relevant destination(s)
4. Message consumer receives the message

This messaging process is a fundamental [14] design pattern within the messaging domain and is commonly used to compose an overall messaging solution; a number of variations on the

pattern are available such as the Dynamic Router [15] and Content-Based Router [16]. Within these design patterns, the router contains key information needed to direct message distribution. The main benefit of such patterns is the centralisation of routing rules. However, centralisation is also a weakness when it comes to scalability and robustness; a single routing location is both a bottleneck and central point of failure.

A more robust routing solution would maintain the benefits of a centralised rule-base and increase robustness using decentralised message distribution. Such an approach would be possible by sharing the rule-base with message producers, enabling them to distribute their messages directly to relevant destinations. This coordinated approach to messaging, illustrated in Figure 1.2, provides the best of both worlds. It maintains the benefits of a centralised rule base, increases scalability with decentralised message delivery, and removes the single point of failure from the messaging solution.

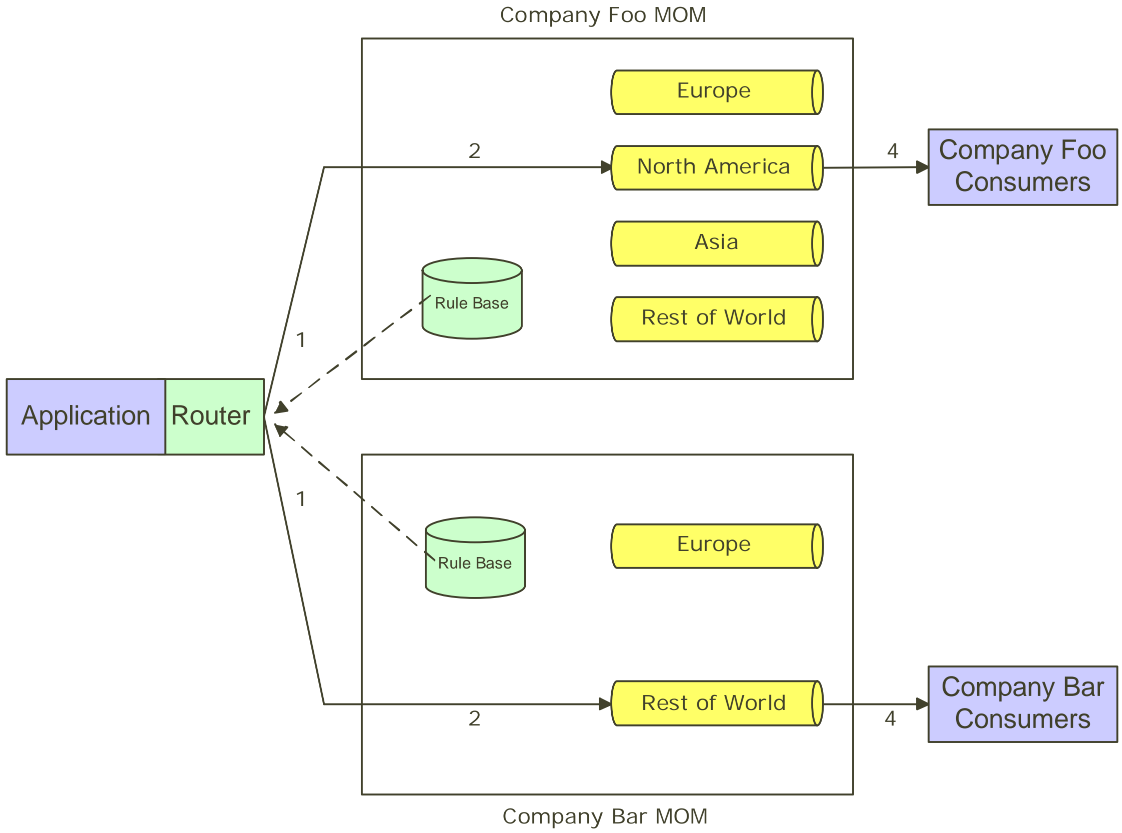


Figure 1.2 *A decentralised message routing solution*

The motivational scenario utilising this decentralised approach operates as follows:

1. Message producer receives routing rules from the centralised rule-base of each company
2. Message producer creates a message and matches it to the relevant destination(s) based on the rule-base
3. Producer sends message directly to destination(s) within each messaging infrastructure

4. Message consumer receives the message

This scenario demonstrates only a single instance where coordination between messaging participants may improve their ability to service their deployment environment. With a generic coordination mechanism in place, the benefits of coordinated interactions may be harnessed within a wide range of messaging scenarios.

1.3 Research Hypothesis

The research hypothesis explored within this work investigates coordination between self-managed systems. The hypothesis is that:

> Within dynamic operating environments, coordinated interaction between self-managed systems can improve the ability of the individual and collective systems to fulfil performance and autonomy requirements of the environment.

1.4 Research Methodology

The research methodology used to test the research hypothesis is broken down into a number of steps. These steps provide an effective roadmap for the research and are summarised as follows:

- Perform a comprehensive literature and technology analysis to evaluate current state of the art practices within the adaptive and reflective middleware and message-oriented middleware domains.
- Define the requirements to facilitate meta-level interaction. Identify a minimal set of communication capabilities to define an appropriate meta-level interaction protocol to achieve coordination.
- Define a generic meta-level relevant to a wide range of message-oriented middleware platforms. This is achieved by the identification of common MOM participants, behaviours, and states.
- Provide a concrete implementation of this generic meta-level to realise a reflective MOM platform. The meta-level must be portable to multiple MOM implementations.
- With a coordinated reflective MOM in place, a substantive evaluation process must be undertaken to assess any benefits. This necessitates the development of a suitable test suite to perform empirical benchmarking with relevant test cases designed and executed to validate the research hypothesis.
- Based on the empirical analysis of the coordinated meta-levels, conclusions are drawn to substantiate the research hypothesis.

1.5 Principal Contributions

The principal contributions of this work are as follows:

1. Investigation into the benefits of coordination between self-managed/reflective systems.
2. Identification of a minimal set of generic prerequisites to facilitate meta-level coordination (See Chapter 4).
3. Definition of an implementation agnostic interaction protocol, the Open Meta-space Interaction Protocol (OMIP), to fulfil the requirements for meta-level coordination (See Chapter 4).
4. Introduction of adaptive and reflective techniques within MOM (see Chapter 4 and 5).
5. Definition of a GenerIc Self-management for Message-Oriented middleware (GISMO), including the identification of common MOM behaviour and state (See Chapter 4).
6. Identification of a design pattern for the development of portable meta-levels. The pattern also promotes a clearer separation of concerns within a meta-level (See Chapter 4).
7. Development of a lightweight framework for non-invasive augmentation of a meta-level on a base-level. Using said design pattern, the framework implements the meta-level and preserves its portability, allowing its attachment to multiple base-levels (MOM implementations) (See Chapter 5).
8. Development of a novel routing solution utilising the capabilities of coordinated meta-level interaction (see Chapter 6).
9. Development of an enhanced destination hierarchy to illustrate the benefits of coordinated meta-levels and reflective capabilities within the MOM domain (see Chapter 7).
10. Identification of requirements to benchmark reflective systems (see Chapter 7).
11. Development of a MOM benchmarking suite, providing a unique simulator to benchmark MOM within dynamic environments (see Chapter 6, Chapter 7, and Appendix A).
12. Design of dynamic test cases to benchmark reflective MOM systems (see Chapter 7).
13. Extensive benchmarking to verify and validate the benefits of these new messaging solutions when compared to their traditional alternative. All benchmarks are performed under conditions comparable to, or better than, standard industrial practices (see Chapter 6, Chapter 7, and Appendix A).
14. Related background knowledge is introduced and future research directions are discussed.
15. International peer-reviewed publications (see Appendix B).

The relationship between research contributions within this work are illustrated in Figure 1.3.

1.6 Thesis Organisation

In brief, the organisation of this thesis is as follows:

Chapter 2 introduces the concepts of reflective middleware and covers the fundamentals of this particular approach to self-management. The distinctions between adaptive middleware and

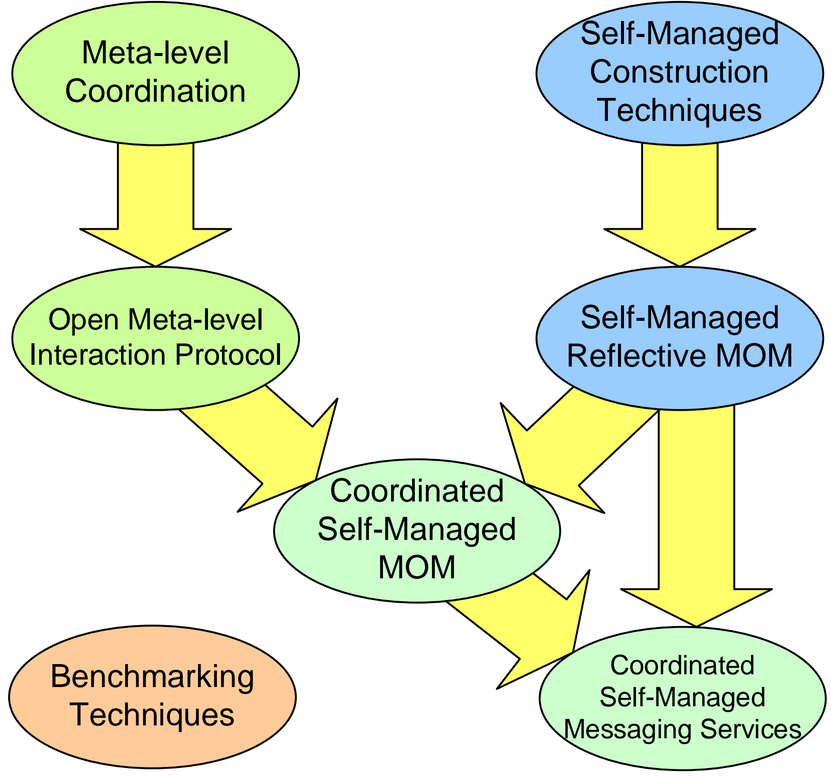


Figure 1.3 *The relationships between research contributions*

reflective middleware are discussed, including different forms of reflective behaviour. Current reflective systems are examined to highlight their capabilities, limitations, and design.

Chapter 3 presents Message-Oriented Middleware (MOM). Rudimentary theories of the MOM domain are summarised to highlight its differences from traditional distribution mechanisms. A number of MOM implementations are examined to highlight both the diversity of MOM application domains and to reveal their reflective ability (self-management capabilities).

Chapter 4 depicts the motivation and pre-requisites for coordinated self-managed systems. A conceptual view of how meta-levels can interact with one another using the Open Meta-level Interaction Protocol (OMIP). The protocol is described and sample uses are discussed.

Chapter 5 brings together the discussion of reflective capabilities, MOM, and coordination, into a definition for a GenerIc Self-management for Message-Oriented middleware (GISMO) for the study of meta-level coordination within self-managed systems. The problems, challenges, and solutions in the design and development of a generic meta-level are highlighted. The identification of common MOM characteristics (participants, behaviour, and state) for defining the meta-level are summarised.

Chapter 6 details an implementation of the GISMO within the Chameleon MOM extension framework. Chameleon illustrates how a meta-level may be augmented to a base-level in a non-invasive manner. Challenges with base-level interaction, the frameworks architecture, and the implementation of the meta-level are all examined. Additional benefits of the framework within the messaging domain are also identified.

Chapter 7 describes the first case study used to evaluate the benefit of coordinated self-managed

systems. The case study examines the benefits of information exchange between interacting participants. The scenario used in this case study expands on the motivational scenario from Section 1.2. With the coordination of participants, a centralised routing solution is altered to a decentralised routing solution. This change maximises maintainability and scalability by combining the advantages of a centralised rule-base with distributed message delivery. Benchmarks are performed on both messaging solutions to assess their benefits.

Chapter 8 contains the second case study used to evaluate this research. The objective of this case study is to examine the benefits of a reflective MOM within dynamic environments. Benchmarks of both reflective and non-reflective MOMs are run within a simulator that recreates dynamic messaging demands.

Chapter 9 concludes with an overview of the experiences and evaluations of the research including lessons learned, possibilities for technology transfer, architectural limitations, future work, and open issues.

Appendix A covers the design of the MOM benchmark suite used within the case studies. Fundamental concepts related to benchmarking MOM solutions are covered, including criteria for metric identification, issues with test case design, and testbed configuration/setup (both hardware and software). The design of dynamic test cases is also discussed.

Appendix B provides XML Schema for the Multimedia and MOM Domain Specific Languages (DSL) described in Chapter 4 and Chapter 5 respectively.

Appendix C contains additional results from the case study carried out in Chapter 8.

1.7 Summary of Conclusions

If the vision of autonomic computing [6] is to be reached, the need for increased coordination between interacting systems is a fundamental prerequisite. The focus of this research was to investigate the utility of coordinated behaviour within the domain of self-managed middleware. As a means to test this hypothesis, Message-Oriented Middleware (MOM) was chosen as the investigational problem domain; MOM is interaction-centric making it an ideal test scenario. Reflection is a key self-management technique that has not previously been utilised within the MOM domain. This creates an additional research theme, the investigation of reflective self-management techniques within MOM. Both of these research tracks were investigated with the definition and development of a coordinated reflective MOM platform. Once the development of the infrastructure was complete, an extensive evaluation was required to assess its benefits.

A comprehensive evaluation process is vital to assess the empirical benefits of research. The benchmarking process for coordinated reflective MOM was extensive. Benchmarks were run on a private network of 12 machines with the execution of 88 benchmark tests taking a combined total of more that 85 hours of benchmarking time. The evaluation process was broken down into two case studies with goals to evaluate coordination between self-managed systems, and evaluate reflective techniques within MOM. All benchmarks were executed under conditions comparable to, or better than, standard industrial practices.

The first case study examines the benefits of coordination between self-managed middleware systems by evaluating the benefits of information interchanges between two meta-levels. With such an ability in place, the benchmarks clearly show the advantages of coordination between reflective self-managed systems and exemplify its potential to foster the development of innovative message

solutions within the MOM domain.

The second case study further assesses meta-level coordination and the benefits of reflective self-management techniques within the MOM domain. The execution of these benchmarks revealed that self-management techniques have a considerable effect on the performance of a MOM provider. Reflective techniques enable the MOM to alter its runtime configuration to match the current demands of its environment thus increasing the level of service it can provide.

The evaluation process is a clear validation of the research hypothesis presented within this thesis. Coordination between reflective self-managed systems can greatly improve their, and the collectives, ability to fulfil the needs of the current operating environment.

Chapter 2

Adaptive and Reflective Middleware Essentials

Adaptive and reflective techniques are a key paradigm for the development of dynamic next-generation middleware platforms. These techniques empower a system to automatically self-manage to meet current operating requirements. This chapter introduces adaptive and reflective techniques and presents a review of current state-of-the-art practice within middleware platforms.

2.1 Introduction

Middleware platforms and services form a vital cog in the construction of robust distributed systems. Middleware facilitates the development of large software systems by relieving the burden on the applications developer of writing a number of complex infrastructure services needed by the system; such services include persistence, distribution, transactions, load balancing, and clustering.

Middleware platforms have traditionally been designed as monolithic static systems. The vigorous dynamic demands of future environments such as large-scale distribution or ubiquitous and pervasive computing [2] will require extreme scaling into large, small, and mobile environments. In order to meet the challenges present in these environments, next-generation middleware researchers are developing techniques to enable middleware platforms to obtain information concerning environmental conditions and adapt their behaviour to better serve their current deployment. Such capability will be a prerequisite for any next-generation middleware; research to date has exposed a number of promising techniques that give middleware the ability to meet these challenges head-on.

Adaptive and reflective techniques have been noted as a key emerging paradigm for the development of dynamic next-generation middleware platforms [12]. These techniques empower a system to automatically self-alter (adapt) to meet its environmental and user needs. Adaptation can take place autonomously or semi-autonomously, based on the systems deployment environment, or within the defined policies of users or administrators [17]. A reflective system is one that can examine and reason about its capabilities and operating environment allowing it to self-adapt at runtime. Reflective middleware is the next logical step once an adaptive middleware has been achieved.

The objective of this chapter is to explore adaptive and reflective techniques, their motivation

for use, and introduce their fundamental concepts. The tools and techniques that enable a system to alter its behaviour are examined. A review and comparison of the application of these techniques within a selection of state-of-the-art middleware platforms is undertaken to reveal their design and coordination capabilities. The first technique explored is adaptive middleware.

2.2 Adaptive Middleware

Traditionally, the design of a middleware platform targets a particular application domain or deployment scenario. In reality, multiple domains overlap and deployment environments are dynamic not static; current middleware technology does not provide support for coping with such conditions. Current research has focused on investigating the possibility of enabling middleware to serve multiple domains and deployment environments. In recent years, platforms have emerged which support reconfigurability, allowing the customisation of platforms for a specific task. This work has led to the development of adaptive multi-purpose middleware platforms. The Oxford English Dictionary defines adapt as:

> Adapt - a. To alter or modify so as to fit for a new use.[1]

An adaptive system has the ability to change its behaviour and functionality. Adaptive middleware is software whose functional behaviour can be modified dynamically to optimise for a change in environmental conditions or requirements [18]. Triggers of adaptations include changes made to a configuration file by an administrator, by instructions from another program or by requests from its users.

2.3 Reflective Middleware

The Oxford English Dictionary defines reflect as:

> Reflect – v. To turn (back), cast (the eye or thought) on or upon something.[1]

The ground-breaking work on reflective programming was carried out by Brian Smith at MIT [19]. A reflective system is one that can examine and reason about its capabilities and operating environment allowing it to self-adapt at runtime. Reflective middleware is the next logical step following the development of adaptive middleware. Reflective middleware builds on adaptive middleware by providing the means to manipulate the internals of a system to adapt it at runtime. This approach allows for the automated self-examination of system capabilities, and the automated adjustment and optimisation of those capabilities. The process of self-adaptation allows a system to provide an improved service for its environment or user's needs. Reflective platforms support advanced adaptive behaviour, adaptation can take place autonomously based on the status of the system's environment, or in the defined policies of its users or administrators [17].

Reflection is currently a hot research topic within software engineering and development. A common definition of reflection is a system that provides a representation of its own behaviour which is amenable to inspection and adaptation, and is causally-connected to the underlying

[1] Oxford English Dictionary, Second Edition, Oxford University Press, 1989.

behaviour it describes [20]. The casual-connection requires that alterations made to the self-representation are mirrored in the system's actual state and behaviour.

Reflective research is also gaining speed within the middleware research community. The use of reflection within middleware for advanced adaptive behaviour gives middleware developers the tools to meet the challenges of next-generation middleware. Its use in this capacity has been advocated by a number of leading middleware researchers [1, 12]. Reflective middleware is self-aware middleware [21].

> The reflective middleware model is a principled and efficient way of dealing with highly dynamic environments yet supports the development of flexible and adaptive systems and applications [21]. This reflective flexibility diminishes the importance of many initial design decisions by offering late-binding and runtime-binding options to accommodate actual operating environments at the time of deployment, instead of only anticipated operating environments at design time [12].

Few aspects of a middleware platform would fail to benefit from the use of reflective techniques. Research is ongoing into the application of these techniques in a number of areas within middleware platforms. While still relatively new, reflective techniques have already been applied to a number of non-functional areas of middleware: system behaviour that is not obvious or visible from interaction with the system. One of the main reasons non-functional system properties are popular candidates for reflection is the ease and flexibility of their configuration and reconfiguration during runtime; changes to a non-functional system property will not directly interfere with a systems user interaction protocols. Non-functional system properties enhanced with adaptive and reflective techniques include distribution, responsiveness, availability, reliability, fault-tolerance, scalability, transactions, and security.

2.4 Are Adaptive and Reflective Techniques the Same?

Adaptive and reflective techniques are intimately related, but have distinct differences and individual characteristics:

- An adaptive system is capable of changing its behaviour
- A reflective system can inspect/examine its internal state and environment

Systems can be developed with adaptive capabilities, reflective capabilities, or with both adaptive and reflective capabilities. On their own, both of these techniques are useful, but when used collectively they provide a very powerful paradigm that allows for system inspection with an appropriate adaptation. When discussing reflective systems the common assumption is that the system has adaptive capabilities. Common terms used in the discussion of adaptive and reflective systems include:

- *Reification* - The process of providing an external representation of the internals of a system. This representation allows for the manipulation of the systems internals at runtime.
- *Absorption / Realisation* - This is the process of enacting the changes made to the external representation of the system back into the internal system. Absorbing these changes into the system realises the casual connection between the model and system.

- *Non-Functional Properties* - The non-functional properties of a system are the behaviours of the system that are not obvious or visible from interaction with the system. Non-functional properties include distribution, responsiveness, availability, reliability, scalability, transactions, and security.
- *Reflective Computation* - Reflective computation is the process of reasoning about and adapting the system.

2.5 Triggers of Adaptive and Reflective Behaviour

The reflective capabilities of a system should trigger the adaptive capabilities of a system. However, what exactly can be inspected to trigger an appropriate adaptive behaviour? Typically, a number of areas within a middleware platform including its functionality and environment are amenable to inspection, measurement, and reasoning as to the optimum or desired performance and functionality. Software components known as interceptors can be inserted into the execution path of a system to monitor its actions. Using interceptors and similar techniques, reflective systems can extract useful information from the current execution environment and perform an analysis on this information.

Usually, a reflective system will have a number of interceptors and system monitors that can examine the state of a system, reporting system information such as its performance, workload, or current resource usage. Based on an analysis of this information, appropriate alterations may be made to the system behaviour. Potential monitoring tools and feedback mechanisms include performance graphs, benchmarking, user usage patterns, and changes to the physical deployment infrastructure of a platform (network bandwidth, hardware systems, etc).

2.6 Adaptive and Reflective Techniques

As a means of implementing adaptive and reflective middleware, a number of techniques have been developed to introduce these capabilities into middleware platforms. These techniques focus on a variety of crosscutting concerns at various levels of the system ranging from low-level structural reflection found within reflective programming languages to high-level architectural reflection. A general outline of the inter-relationship between reflective techniques is illustrated in Figure 2.1. The remainder of this section provides a brief description of these techniques.

2.6.1 Structural Reflection (Programmatic)

Structural reflection [19] provides the ability to alter the statically fixed internal data/functional structures and architecture used in a program. A structurally reflective system would provide a complete reification of its internal methods and state, allowing their inspection and change. Low-level structural reflection is most commonly found in programming languages, such as Smalltalk and java.lang.reflect, offering the ability to change the definition of a class, a function or a data structure on demand.

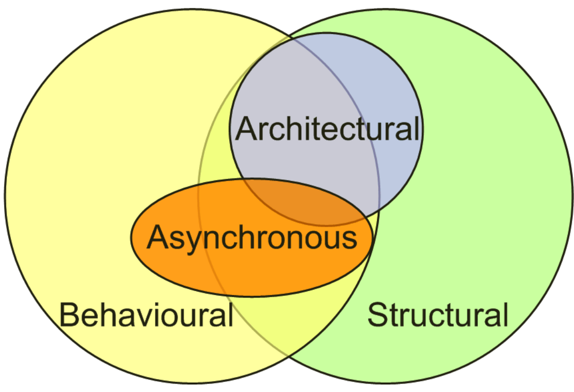


Figure 2.1 *The inter-relationships between reflective techniques*

2.6.2 Behavioural Reflection

Behavioural reflection is the ability to intercept an operation such as a method invocation and alter the behaviour of that operation. This allows a program, or another program, to change the way it functions and behaves. Behavioural reflection alters the actions of a program at runtime with the use of a number of techniques including polymorphism, reflective programming languages, design patterns, and structural reflection.

2.6.3 Architectural Reflection

Architectural reflection is similar in nature to structural reflection but operates at the higher architecture-level of a system. With architectural reflection "there exists a clean self-representation of the system architecture, which can be explicitly observed and manipulated" [22]. Architectural reflection involves the inspection and adaptation of software architecture. This objective is achieved by expressing the composition of the software architecture in an Architectural Meta Model (AMM) [7, 23]. The architecture is typically described within a AMM using a configuration or component graph to express the current structure and composition of the architecture; what components/modules are in use and how they are binded together (connected).

The AMM can then be inspected and manipulated to adapted the architecture by replacing a component, reconfiguring a binding, etc; these changes are then reified in the systems base-level. An unconstrained AMM could lead to invalid adaptations resulting in an unstable system, unpredictable system behaviour, or system failure. To prevent such an outcome the AMM can often be associated with a set of architectural constraints to ensure the safe adaptation of the architecture [7], preserving system integrity.

2.6.4 Synchronous Reflection

The origins of reflection originate from reflective programming languages like OpenC++ [24], OpenJava [25], and CodA [26]. Within these languages, reflective computation is performed in-line or synchronously with program execution.

> Synchronous reflective programming languages generally realise the causal connection between the base-level and meta-level as an implementation link. Reflective code

> can only be inserted/removed at the reification points and is generally executed by an application thread, i.e., it is executed synchronously with program execution. [27]

Language-level reflective capability is useful for creating dynamic flexible applications. However, within a large-scale distributed system a more global approach to reflection is desirable. This form of reflection is referred to as system-level reflection and its goal is to monitor system state and perform configurations based on adaptive policies or logic.

The synchronous approach used within language-level reflection is not always desirable for system-level reflection, as it must take a more global view of the system when it considers possible adaptations. A number of system-wide (global) factors need to be included within the decision making process, these include current system make-up, possible replacement components, new or altering requirements or temporal usage and performance metrics; how the system performed in the past and identification of any usage patterns. Additional factors within the reflective process could include endless domain specific criteria such as QoS within the multimedia domain, or transaction integrity and security within the financial domain.

Given the diverse nature of the reflective criteria and the manner in which they may be collected and analysed it is not always desirable to link system-level reflection with the execution of the system, such reflection should be asynchronous.

2.6.5 Asynchronous Reflection

> Asynchronous reflection is a goal-directed process that is not separable from the system on which it is operating. Its essential characteristics are observation and outputs (new knowledge, system adaptation, plans of action, etc). Ideally, it should be able to acquire new knowledge, store that new knowledge and in turn reflect on that new knowledge, leading to towers of reflection. [27]

Asynchronous reflection decouples system-level reflection from system execution allowing both to execute independently. Such capability is very beneficial within high-demand environments as it allows resources to be directed as needs demand. During peak hours, reflective computation can be minimised to maximise the output of the system. During off-peak hours more intensive reflective computations, such as pattern analysis, can be performed. This maximises system utilisation and minimises any negative effects caused by reflective computation during peak usage periods.

The techniques discussed in this section can be implemented with the use of a number of approaches such as dynamic/smart proxies, dynamic design patterns (such as the Interception and Chain of Responsibility patterns), and dynamic aspect-oriented programming. These techniques are discussed in more detail in Chapter 5. One technique central to the design of a reflective system is the use of a meta-level; a basic understanding of this technique is required to place this research within the current state-of-the-art.

2.7 Meta-levels

In 1991, Gregor Kiczales's work on combining the concept of computational reflection and object-oriented programming techniques lead to the definition of a meta-object protocol [28]. One of the key aspects of this groundbreaking work was the separation of a system into two levels, base and

meta. The base-level provides system functionality, and the meta-level contains the policies and strategies for the behaviour of the system. The inspection and alteration of this meta-level allows for changes in the system's behaviour. The remainder of this section discusses meta-levels from the perspectives of operation, execution synchronicity, concern separation, performance issues, and coordination capabilities.

2.7.1 Operation Overview

Within a system designed using a meta-object protocol, the base-level provides the implementation of the system and exposes a meta-interface accessible at the meta-level. This meta-interface exposes the internals of the base-level components/objects, allowing it to be examined and its behaviour to be altered and reconfigured. The base-level can now be reconfigured to maximise and fine-tune the systems characteristics and behaviour to improve performance in different contexts and operational environments. This interface is often referred to as the Meta-Object Protocol or MOP. The design of a meta-interface/MOP is central to studies of reflection. The interface should be sufficiently general to permit unanticipated changes to the platform but should also be restricted to prevent the integrity of the system from being destroyed [7]. Common terms used in the discussion of meta-level include:

- *Meta* - Prefixed to technical terms to denote software, data, etc., which operate at a higher level of abstraction.[1]
- *Base-Level* - The level of software that provides the functional operations of a system.
- *Meta-Level* - The level of software that abstracts the functional and structural level of a system.
- *Meta-Level Architectures* - Systems designed with a base-level (implementation level) that handles the execution of services and operations, and a meta-level that provides an abstraction of the base-level.
- *Meta-Object* - The participants in an object-oriented meta-level are know as meta-objects.
- *Meta-Object Protocol* - The protocol used to communicate with the meta-object is known as the Meta-Object Protocol (MOP).

In addition to providing access to the system's internal structure such as its dependencies, interfaces, endpoints, architecture and configuration, the meta-level can also be used to store additional information, or meta-information, on the base-level such as usage statistics. Essentially, the meta-level is the "management" level of a system. Reflective techniques enhance this layer with the ability to self-analyse and self-adapt, creating a self-managed system.

When creating a meta-level for a system, a number of factors must be considered in its design. These include its synchronisation, concern separation, and performance. An additional factor that should also be considered is the openness of the meta-level. This is an important design decision and can dictate how coordinated a meta-level can be with the other meta-levels with which it interacts.

[1] Oxford English Dictionary, Second Edition, Oxford University Press, 1989.

2.7.2 In-Line and Out-of-Line Execution

Meta-levels can be executed *in-line* with system execution or independently, *out-of-line* with system execution. In-line execution provides for synchronous reflection and out-of-line execution achieves asynchronous reflection. As covered in Section 2.6.4, the choice of whether a meta-level model is in- or out-of-line with program execution is an important consideration as both synchronous and asynchronous reflection can be used to achieve different goals.

When examined from the perspective of meta-level execution, this decision comes down to a simple choice. Do you wish to make decisions in-line with the execution of the base-level, or do you wish to make decisions out-of-line asynchronously with the execution of the base-level? When designing a meta-level, this is not a mutually exclusive choice as meta-levels can support a mix of both forms of reflective computations to meet the needs of the systems they describe. Such an approach allows the construction of powerful reflective systems that can use both forms of reflective computation to complement one another. For example, synchronous reflection may be used to gather information from the base-level which can then be forwarded to an asynchronous decision making process to perform reflective computations asynchronously at off-peak usage periods.

2.7.3 Concern Separation

Meta-levels may be used to describe a number of aspects of the base-level of a reflective platform. In order to create an effective meta-level, it is important to provide a clear separation of concerns [29] within its design. Meta-levels can describe a number of different concerns within a base-level. These concerns can be split into multiple models [30] within the meta-level to provide clearer concern separation.

A useful illustration of concern separation within a reflective platform is provided by Open ORB [7]. Within the Open ORB meta-level, four distinct meta-models describe the interface, architecture, interception, and resource management concerns of the Open ORB base-level. These divisions simplify the meta-level and reduce the complexity of the meta-object protocol used to interact with the Open ORB base-level. A clear separation of concerns within a meta-level can help to produce a meta-level that is both intuitive to use and easy to understand.

2.7.4 Performance

Traditionally, one of the major arguments against the use of meta-levels is the fear of severe performance loss. Given the fact that interception is a popular method for the implementation of meta-levels [7, 8, 9, 10, 31, 32, 33] such concern is well founded. As with any interception-based mechanism, an interception meta-level will include a layer of indirection to allow the placement of interceptors within the system. Initial reports on the level of performance loss experienced within reflective systems show a substantial decrease in performance; in Iguana, a reified method invocation costs about 12-times a C++ virtual function invocation.[34]. The main case for introducing reflective abilities into a system centres on the expectation that such capability will improve the performance of the system. To benefit from reflection, any performance improvements must negate the effects of performance loss due to the reflective process, returning a net performance contribution to the system.

> ...using reflection to select alternative underlying mechanisms (e.g. alternative load-

> ers, binders, protocols, thread schedulers, etc.) can actually enhance performance by helping to ensure that the underlying system is always configured optimally for the current application mix and environmental circumstances. [35]

However, recent research from Coulson (2004) adds further weight to the case for reflective systems. In [35] he argues that different types of meta-models may be present in a reflective meta-level. These models may deal with aspects of the system, for example OpenCOM [7] has meta-models which deal with architecture, resource management, interfaces, and interception. Based on this assumption Coulson argues that

> ...in many instances, the overhead of reflection need only be incurred ‘occasionally’ in such a way that critical ‘in-band’ performance is not impacted. We use the term ‘in-band’ to refer to segments of essential code that are repeatedly executed in the normal course of events and are therefore particularly performance critical. Conversely, ‘out-of-band’ code is executed only occasionally and its impact on overall performance is negligible. [35]

The concepts of in-band and out-of-band execution are similar to the in-line and out-of-line execution concepts covered in Section 2.7.2. In an examination of the OpenCOM meta-model structure this work found that “two of the three ‘core’ meta-models ... (i.e. the architecture and interface metamodels) hardly incur any ‘in-band’ overhead. This is also true for the resources meta-model”[35], further information on OpenCOM is available in Section 2.8.3. This work confirmed the previous efforts of Dowling [34] on the performance costs suffered from the use of ‘in-band’ interception-based meta-models. These efforts support the case for the use of “out-of-band” reflective techniques, such as asynchronous reflection covered in Section 2.6.4 and out-of-line meta-models discussed in Section 2.7.2, in the development of effective meta-levels that minimise their associated overheads.

2.7.5 Openness to Coordination

Current meta-levels take a closed approach to their implementation, limiting the possibility of coordination with other meta-levels. The objective of this research is to examine the benefits of coordination between meta-levels. Given this basis, the level of openness within a meta-level will be a key factor in the design of a coordinated meta-level. The definition of openness within a coordinated context refers to the level of access available from the meta-level, including the quantity and type of meta-information and adaptations exposed to third parties.

Depending on the meta-level’s application domain, the level of openness will vary. Interaction-oriented environments, such as distributed computing, could benefit from a very open meta-level as apposed to a more isolated environment such as a disconnected embedded system. The meta-level designer will have to choose the level of openness to suit the target domain. Additional factors that could affect this decision include system security, system self-interest, environmental hostility, or system ownership.

2.8 Current Reflective Research

The objective of this review is to provide a state-of-the-art overview of adaptive and reflective research and the use of meta-level techniques within current middleware platforms. In particular, the review examines the state-of-art with respect to coordination capabilities and limitations. The review covers a number of research projects from the groundbreaking GARF [36, 37] and CodA [26] to the more recent K-Components [23, 27, 38]. Well-known reflective projects such as QuO [31, 39, 40] and Open ORB [7] are also included.

2.8.1 mChaRM

The Multi-Channel Reification Model (mChaRM) [32] is a reflective approach that reifies and reflects directly on communications. The mChaRM model does not operate on base-objects but on the communications between base-objects, resulting in a communication-oriented model of reflection. This approach abstracts and encapsulates inter-object communications and enables the meta-programmer to enrich and/or replace the pre-defined communication semantics. mChaRM handles a method call as a message sent through a logical channel between a set of senders and a set of receivers. The model supports the reification of these channels into logical objects called multi-channels. A multi-channel can enrich the messages (method calls) with new functionality such as security, fault tolerance, multi-point delivery, and check pointing. This technique allows for finer reification reflection granularity and a simplified global approach to the development of communication-oriented software. Multiple channel types exchange messages between the same participants, with each channel providing a different communicative capability.

mChaRM operates as an addition to the Java programming language using the placement of keywords within Java code to specific mChaRM behaviour, these keywords are then replaced with generated code using the language extension capabilities of OpenJava [25]. In a simplistic sense, mChaRM operates in a similar fashion to an interception mechanism and provides a global and structured view of the communication process. mChaRM is specifically targeted for designing and developing complex communication mechanisms from the ground up, or for extending the behaviour of current communication mechanisms. mChaRM has been used to extend the standard point-to-point Java RMI mechanism into one that supports multi-point communication with reliable multicast.

Within mChaRM, the communicative act is broken down into entities known as loci. Three distinct loci exist: source loci (sender), abstract loci (message delivery), and target loci (receiver) as illustrated in Figure 2.2.

Each of these loci possess their own meta-level with an associated API to access their base-level. "All messages exchanged among multi-channel components are encapsulated in the API supplied to the meta-programmer" [32]. This architectural approach ensures the encapsulation of each loci. While no direct meta-level interaction is possible between the meta-levels of loci, limited interaction is used to access the base-level of a loci. However, one small exception exists. A restricted coordinated capability is possible between the source loci and the abstract loci with the exchange of a recipient list between them. This illustrates a simplistic meta-information transfer via a RMI method invocation. Increased coordination could be achieved within the mChaRM framework via the implementation of multi-channels, however this would break the encapsulation of each loci.

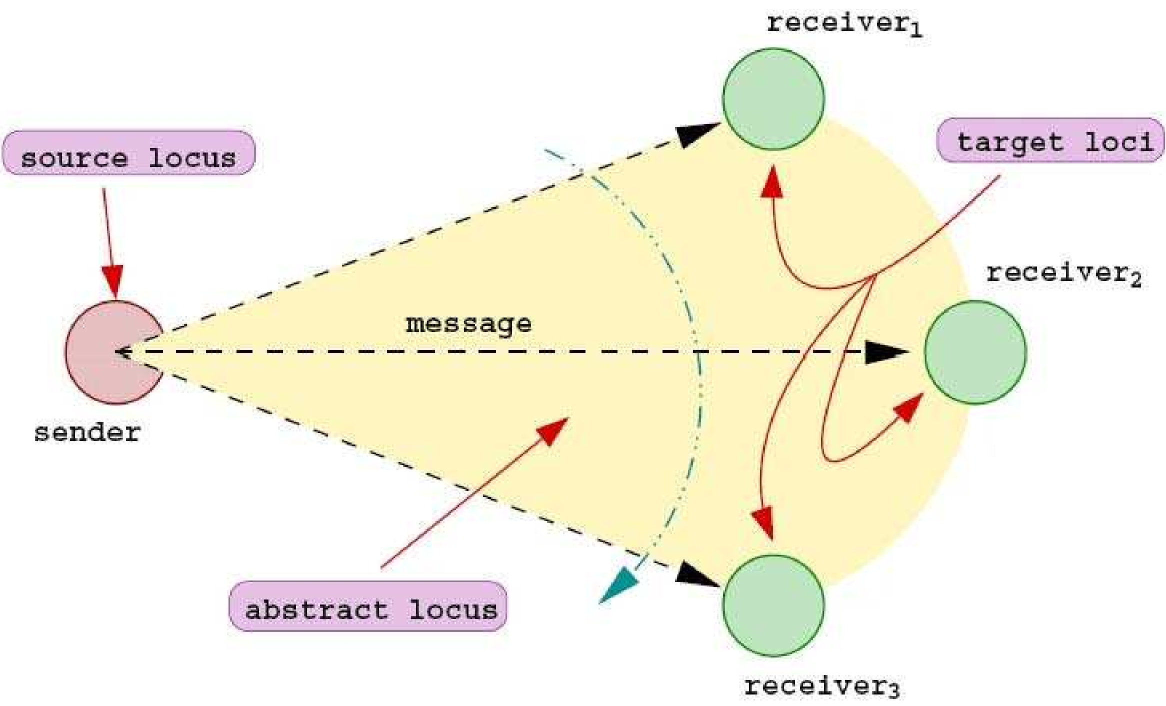


Figure 2.2 *The mChaRM Loci model (from [32])*

2.8.2 QuO

One of the main design goals of traditional middleware platforms was to hide the complexity associated with infrastructure tasks such as distribution, fault-tolerance, and load balancing. Within many application domains, this hiding was welcome. However, for a number of application domains, the middleware hides too much information. Applications dealing with multimedia and real-time demands (i.e. stock markets) have stringent Quality of service (QoS) demands and need access to information concealed within the middleware infrastructure to meet their requirements.

Researchers at Bolt, Beranek, and Newman (BBN) Technologies tackled this problem with the development of the Quality Objects (QuO) framework [31, 39, 41]. The design of QuO aims to assist in the development of QoS stringent distributed systems.

> QuO opens up the implementation of a distributed object application, enabling the specification, measurement, and control of the QoS aspects of an application and the specification and implementation of adaptive behaviour in response to changing system conditions. [40]

QuO supports the construction of client/server-based self-adaptive applications and provides a Quality Description Language (QDL) for describing the QoS aspects of QuO applications. The Contract Description Language (CDL) allows the definition of a QoS contract between a client and servant, the contract specifies the level of service expected and advises on the adaptive actions taken if the QoS moves outside of these limits. The use of the QDL allows the separation of QoS concerns from functionality concerns. The QoS contract expressed in CDL allows the specification of the "level of service desired by a client, the level of service an object expects to provide, operating regions indicating possible measured QoS, and actions to take when the level of QoS changes" [40].

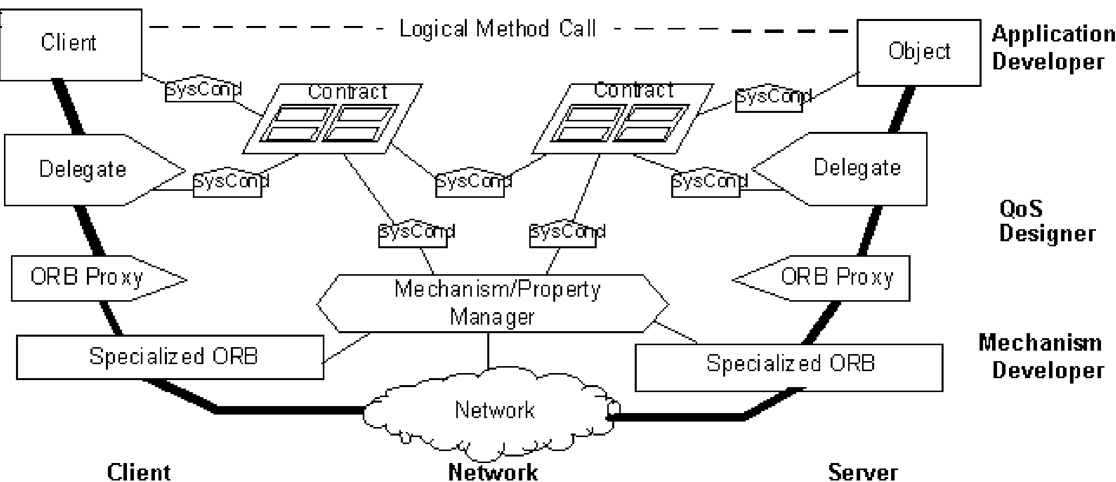


Figure 2.3 *A QuO method invocation (from [31])*

Within the method invocation, as shown in Figure 2.3, System Condition Objects or "SysCond" monitor QoS information and allow its analysis at runtime.

> System condition objects (sysconds) provide interfaces to monitor and control low-level details of the system. The contract uses sysconds as variables in predicates to determine its current state or region. Sysconds can be selectively monitored by the contract so that changes in syscond values will trigger a re-evaluation of the contract, which results in a re-evaluation of the predicates and the execution of any defined transition actions if the resulting state is different from the previous state. [42]

SysCond objects can expose the QoS state in a number of ways from simple value probes to periodical polls and sliding window counters. Once a QuO application is running, its runtime kernel is responsible for the coordination and evaluation of contracts and monitoring of SysCond objects.

QuO delegates are integrated into application code using a similar technique to weaving within Aspect-Orientated Programming (AOP) [43]: "QuO delegates are similar to Composition Filters [44] in that they both weave code into the application by wrapping the target object and intercepting messages to it" [42]. QuO also uses code generators to expand and merge QDL descriptions, QuO kernel code, and application code to produce a single executable program.

When examined from a coordination perspective, QuO does not provide any support for meta-level interaction. While clients and servants possess independent adaptive capabilities and QDL definitions, they cannot interact directly with one another. However, SysCond objects are shared between both the client and servant, providing a shared model of the system state. This allows the possibility of cooperative adaptive behaviour by coordinating both client and servant QDL definitions to perform complementary coordinated actions based on the shared state with the *SysConds* objects [27]. This approach could even be extended to provide a primitive client-servant communication link by hacking a SysCond to act as a link to exchange information between the client/servant and vice versa.

While this approach might sound promising at first, it is not an ideal foundation to build a coordinated self-managed system. The task of coordinating three adaptive artefacts, client QDL, servant QDL and shared sysconds can quickly become complex when modelling an intricate large-scale system; QuO provides no support to normalise these artefacts. Furthermore, the design of QuO only supports the construction of client/server based self-adaptive applications and does not support other topologies such as peer-to-peer decentralised systems.

2.8.3 OpenCOM & Open ORB

The Common Object Request Broker Architecture (CORBA) based Open ORB 2 [7] is an adaptive and dynamically reconfigurable Object Request Broker (ORB) supporting applications with dynamic requirements. Open ORB is designed from the ground up to be consistent with the principles of reflection and is built upon OpenCOM, a lightweight adaptable component object model inspired by Microsoft's COM [45]. Open ORB exposes an interface (framework) that allows components to be plugable; these components control several aspects of the ORBs behaviour including thread, buffer, and protocol management. The ORB implementation consists of a collection of configurable components that are selectable at build-time and reconfigurable at runtime. This process of component selection and configurability enables the ORB to be adaptive.

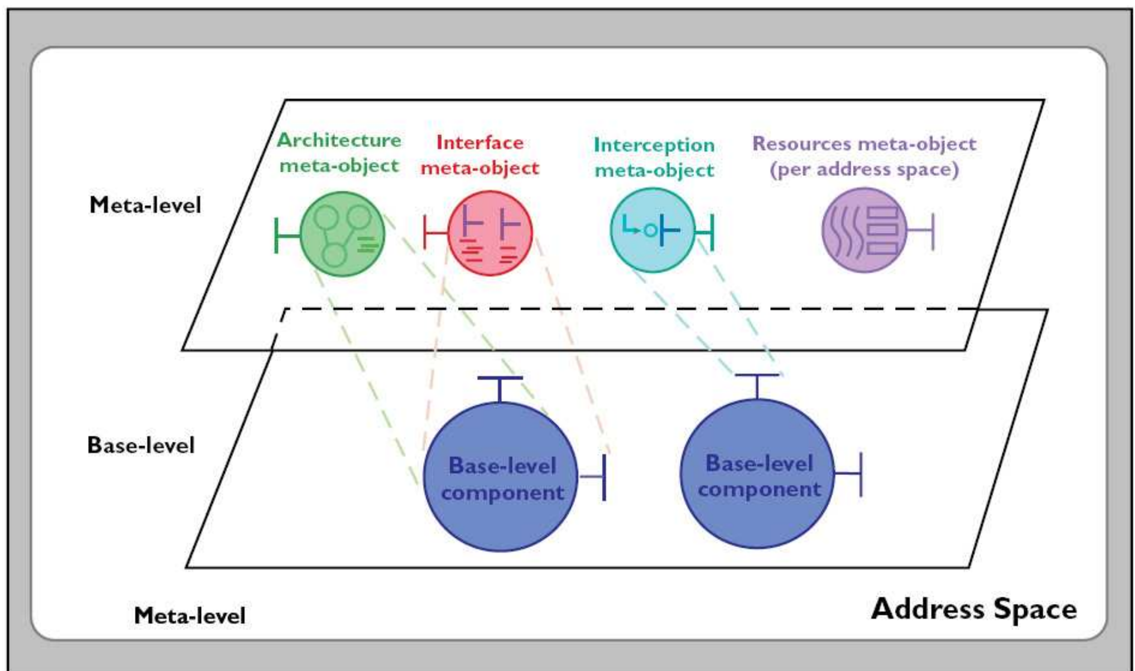


Figure 2.4 *The Open ORB meta-architecture (from [21])*

The design of Open ORB uses a clear separation between base-level and meta-level operations. The ORB's meta-level is a causally connected self-representation of the ORBs base-level (implementation) [7]. Each base-level component may have its own private set of meta-level components that are collectively referred to as the components' meta-level. The meta-level is broken down using the multi-model approach [30] into several distinct models, simplifying the interface to the meta-level by separating concerns between different system aspects. This allows each distinct

meta-level model to give a different, independently reified, view of the platform implementation. As shown in Figure 2.4, meta-models cover the interface (*IMetaInterface*), architecture (*IMetaArchitecture*), interception, (*IMetaInterception*) and resource management concerns of the ORB. These models provide access to the underlying platform and component structure through reflection; every application-level component offers a meta-interface that provides access to an underlying meta-level, which is the support environment for the component.

Open ORB uses two meta-models to deal with structural reflection, one for its external interfaces, and one for its internal architecture. The interface meta-model acts similarly to the Java reflection API allowing for the dynamic discovery of a component's interfaces at runtime. The architecture meta-model details the implementation of a component broken down into two parts: a component graph (a local-binding of components) and an associated set of architectural constraints to prevent system instability [7]. Such an approach makes it possible to place strict controls on access rights for the ORB's adaptation. This allows all users the right to access the interface meta-model while restricting access rights to the architecture meta-model, permitting only trusted third parties to modify the system architecture.

Two further meta-models exist for behavioural reflection, the interception and resource models. The interception model enables the dynamic insertion of interceptors on a specific interface allowing for the addition of pre-behaviour and post-behaviour. This technique can introduce non-functional behaviour into the ORB's execution. Unique to Open ORB is its use of a resource meta-model allowing for access to underlying system resources, including memory and threads, resource factories, and resource managers via resource abstraction [7]. This model provides control and accounting facilities for resources to simplify QoS management [46].

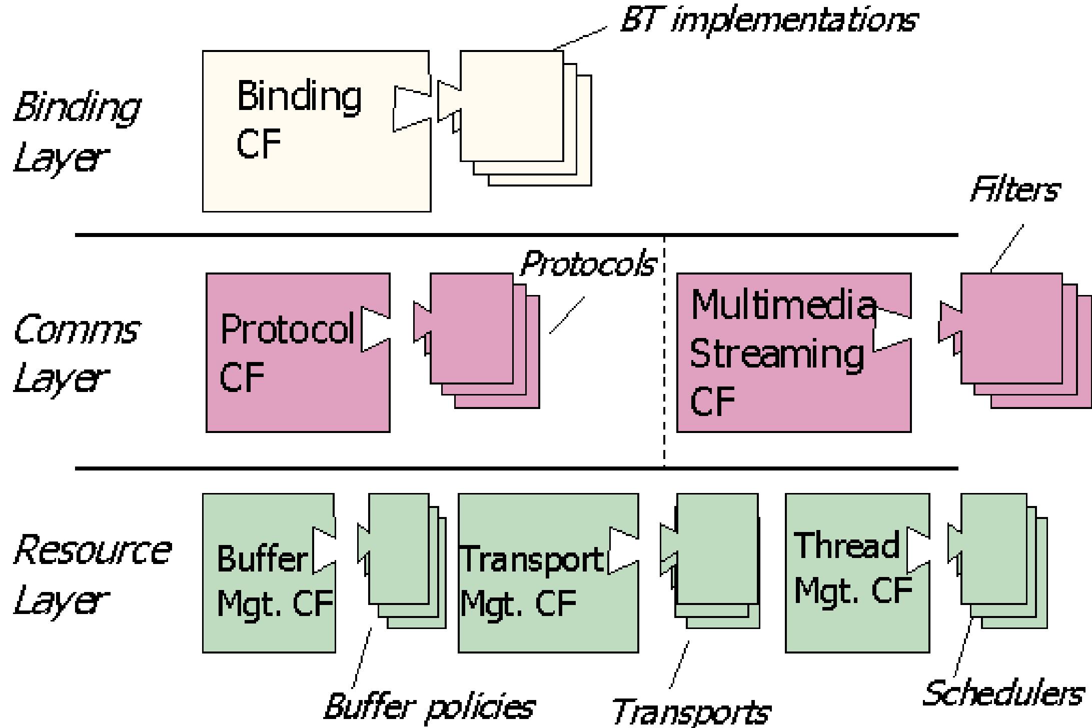


Figure 2.5 *Component framework in Open ORB (from [47])*

An examination from a coordination perspective reveals that Open ORB is designed to run within a single address space, its design is heavily influenced by the client/server distribution model and contains with limited support for other deployment topologies. However, minimal coordinated behaviour is visible within its binding framework. This framework, shown in Figure 2.5, supports dynamic bindings between clients and servers and has the responsibility of ensuring that bindings of interacting clients and servers are consistent across ORB instances. "The [binding] model encompasses both *local* bindings which are primitively realised within a single address space, and *distributed* bindings, which can span address spaces and machines" [47].

> Participants that are remote with respect to a binder's location are represented by reps ('remote participant representatives'). The process of creating a rep falls into two stages as follows. First, a generator is used at the participant's (remote) site to generate both an iref and an associated communication infrastructure. An iref is a value that represents a participant and can be passed around the distributed system. Second, the iref is transferred to the binder's site (by some means or other) at which it is passed to a resolver that is responsible for creating a responding rep. This whole process is referred to as participant remoting. [47]

Remote participants within a binding are initialised and controlled at runtime with the Binder API. The API provides generic interfaces for entities within the binding model. Configuration of the remote participant is possible by passing information such as settings and QoS specifications within the *bindContext* parameter of the *IGenericBinder* interface. When the binding established, runtime control is possible with the use of the *BindingCtl* to alter its operation.

One should note that this is not an example of first-class meta-level interaction, although meta-level information such as the QoS specification might be used within the binding interaction; rather, it is a small and limited interaction within the binding framework. Its examination reveals the use of coordination interactions within the lower layers of current self-managed systems to coordinate their reflective behaviours.

2.8.4 dynamicTAO (UIC/2K)

Another CORBA based reflective middleware project is dynamicTAO [8]. dynamicTAO is designed to introduce dynamic reconfigurability into the TAO ORB [48] by adding reflective and adaptive capabilities.

> TAO is a portable, flexible, extensible, and configurable ORB based on object-oriented design patterns. It is written in C++ and uses the Strategy design pattern to separate different aspects of the ORB internal engine. A configuration file is used to specify the strategies the ORB uses to implement aspects like concurrency, request demultiplexing, scheduling, and connection management. At ORB startup time, the configuration file is parsed and the selected strategies are loaded. [8]

Unlike the segregated meta- and base-level approach taken within Open ORB, dyanmicTAO "concentrates on a simpler reflective model, focusing on high performance" [8]. This results in meta-capabilities weaved within the ORBs implementation with an associated meta-interface. dynamicTAO enables on-the-fly reconfiguration and customisation of the TAO ORBs internal engine, while ensuring it is maintained in a consistent state.

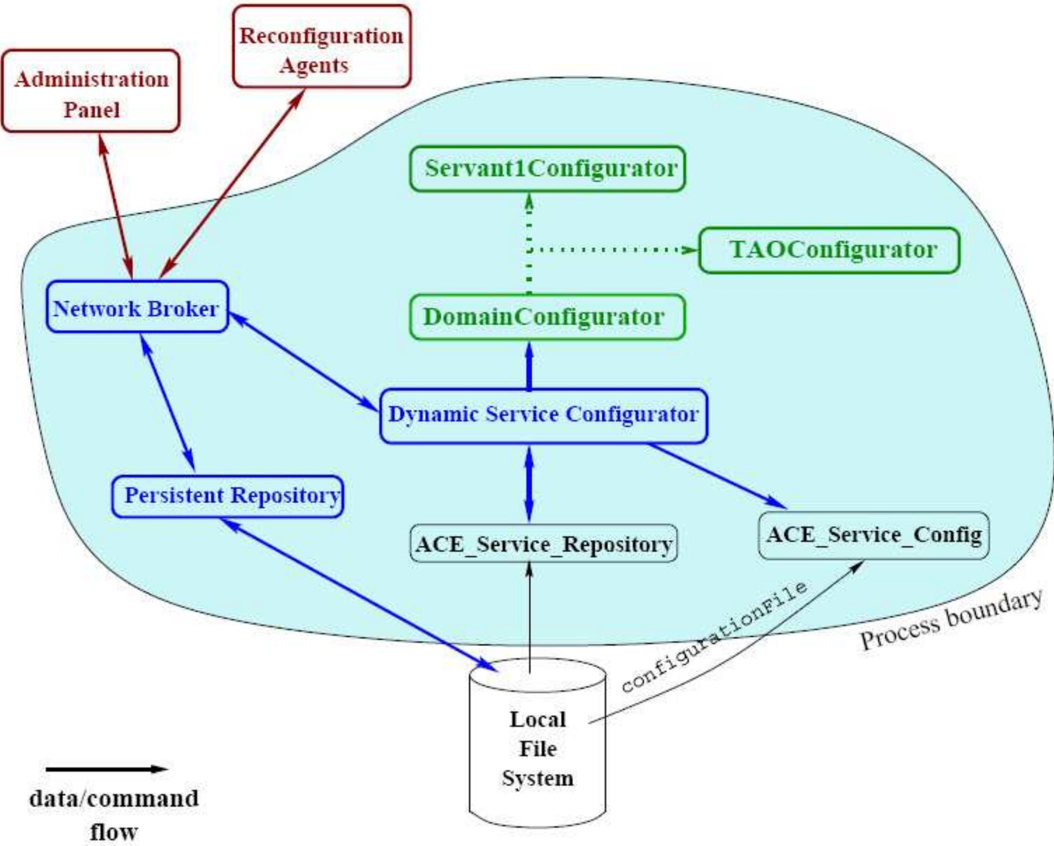


Figure 2.6 *dynamicTAO component architecture (from [49])*

The architecture of dynamicTAO is illustrated in Figure 2.6 and Figure 2.7. Within this architecture, reification is achieved through a collection of component Configurators. dynamicTAO allows components to be dynamically loaded and unloaded from the ORBs process at runtime enabling the ORB to be inspected and its configuration to be adapted. Component implementations are organised into categories representing different aspects of the ORB's internal engine such as concurrency, security, monitoring, and scheduling. Inspection in dynamicTAO is achieved with interceptors that may be used to add support for monitoring. Interceptors may also introduce additional behaviours such as cryptography, compression, and access control.

dynamicTAO "exports a meta-interface for the loading and unloading of components into the ORB at runtime, and the inspection of the ORB's configuration state"[21]. The meta-interface can be accessed remotely through a *Network Broker* and the *Dynamic Service Configurator*. Remote administration interactions are possible using the Distributed Configuration Protocol (DCP). DCP allows the following remote operations:

- Retrieve information of a remote ORB's persistent information repository (what components are available in the ORB's repository)
- Retrieve runtime configuration information (what components are running)
- Load implementations and reconfigure (load or reconfigure components)

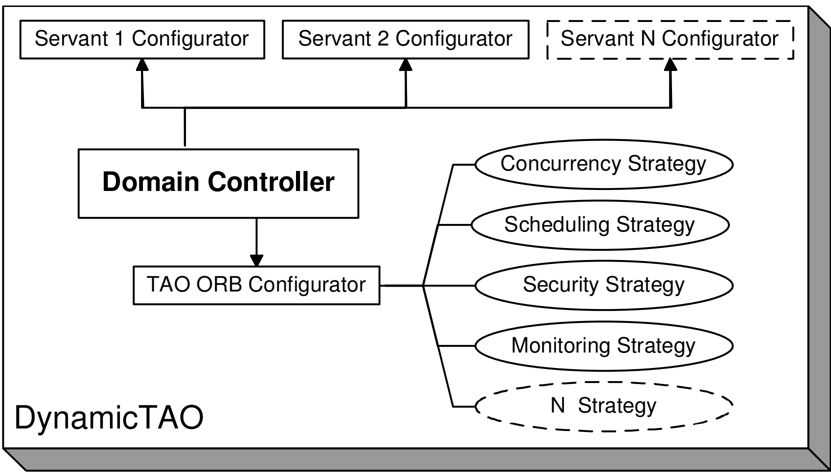


Figure 2.7 *Reifying the dynamicTAO structure (adapted from [8])*

- Access the persistent repository (upload, delete or download a component's code)

The DCP is a straightforward protocol that allows administrative access to a remote ORB instance and illustrates the benefits and potential for meta-level interaction and coordination within reflective systems. One utilisation of the protocol is the development of a Mobile Reconfiguration Agent (MRA) for ORB networks.

As a means of coping with the demands within large-scale deployments, it is common practice to utilise inter-connected networks of ORBs and collections of ORB replicates spread across multiple physical locations. As the size of the ORB network increases the effort required to configure and administrate the network also increases. "In order to configure a particular ORB, a point-to-point connection between the administration node (e.g. running Doctor [a configuration tool]) and the ORB process" [8] is required. Within a large network, this process can be labour and bandwidth intensive. Kon provides a rational and approach for the development of MRAs:

> As a first solution to the problem we considered implementing a management front-end that would allow administrators to type sequences of DCP commands that would be sent to a list of ORBs. Although this approach would simplify the work of the administrator, it would not solve the problem of bandwidth waste, i.e., sending large amounts of duplicated information across long-distance Internet lines.
>
> The solution we adopted was to allow administrators to organize the nodes of their Internet systems in a hierarchical manner for reconfiguration purposes. The administrator specifies the topology of the distributed application as a directed graph and creates a mobile reconfiguration agent which is injected into the network. The reconfiguration agent then visits the nodes of this graph of interconnected ORBs. In each ORB, the agents are received by the Reconfiguration Agent Broker. The broker first replicates and forwards the agent to neighboring nodes, then processes the DCP commands locally, and finally, collects the reconfiguration results, sending them back to the neighboring agent source. [8]

The development of MRAs illustrates the potential power of an open coordinated meta-level. The simple DCP interface facilities the development of MRAs that travel through the ORB network performing reconfigurations based on the administrator's instructions. The DCP was able to achieve this simple form of coordinated interaction between MRAs and the ORBs meta-level.

The primary design objective of the DCP is remote ORB administration; its ability to facilitate non-administrative coordination interactions between ORBs is limited. The protocol offers only limited support for state exchange and provides no mechanism to extend itself to contain such information, or to extend its interaction capability. The protocol assumes a master/slave relationship between client and servant with no ability for a servant ORB to refuse a reconfiguration. In its current form, v1.1[1], it provides simple administrative capabilities but does not support any security or access control mechanisms; although, a security extension is planned. The protocol itself is specific to dynamicTAO/TAO and does not support any other middleware types, however its abstract principles could define a generic protocol for ORBs.

2.8.5 K-Components

K-Components [23] is a component framework designed to support autonomic components with the use of reflective techniques. K-Components is of particular interest to this research as it is one of the first efforts to investigate the coordination of the decentralised reflective systems with the use of Collaborative Reinforcement Learning (CRL) and facilitates communication between decentralised runtime deployments by exploiting the event-communication paradigm.

K-Components provides a description language to describe possible adaptation of components, similar to the QDL used for writing QoS contracts within the QuO project covered in Section 2.8.2.

> The model provides a component interface definition language called K-IDL, an extension to IDL that supports the definition of component states and adaptation actions, as well as required interfaces. Component states and adaptation actions are used by decision making programs to reason about component operation and adapt component operation. A K-IDL compiler translates component interface definitions into an extended version of IDL-2 and interfaces are compiled using modified Orbacus/C++ middleware. [38]

Reflective capabilities are introduced into the framework with the use of "adaptation contracts". These contracts are associated with a component and provide reflective capabilities by "reasoning about adaptation conditions using component/connector states and feedback events regarding remote component states" [38]. These adaptation contracts are specified using the Adaptation Contract Description Language (ACDL); this language allows the "association of component/connector states or feedback events with adaptation actions using if-then rules or the event-condition-action (ECA) paradigm" [38].

Within the context of self-adaptive systems one of the major drawbacks of the top-down "rule-based and the ECA approach to specifying self-adaptive behaviour is that it becomes infeasible as the space of possible feedback events and adaptation actions increases" [38]. To cope with this limitation a capability for components to perform autonomous learning was introduced. Using this approach, self-adaptive behaviour can be "learnt by components using an unsupervised technique

[1] http://choices.cs.uiuc.edu/2k/DCP

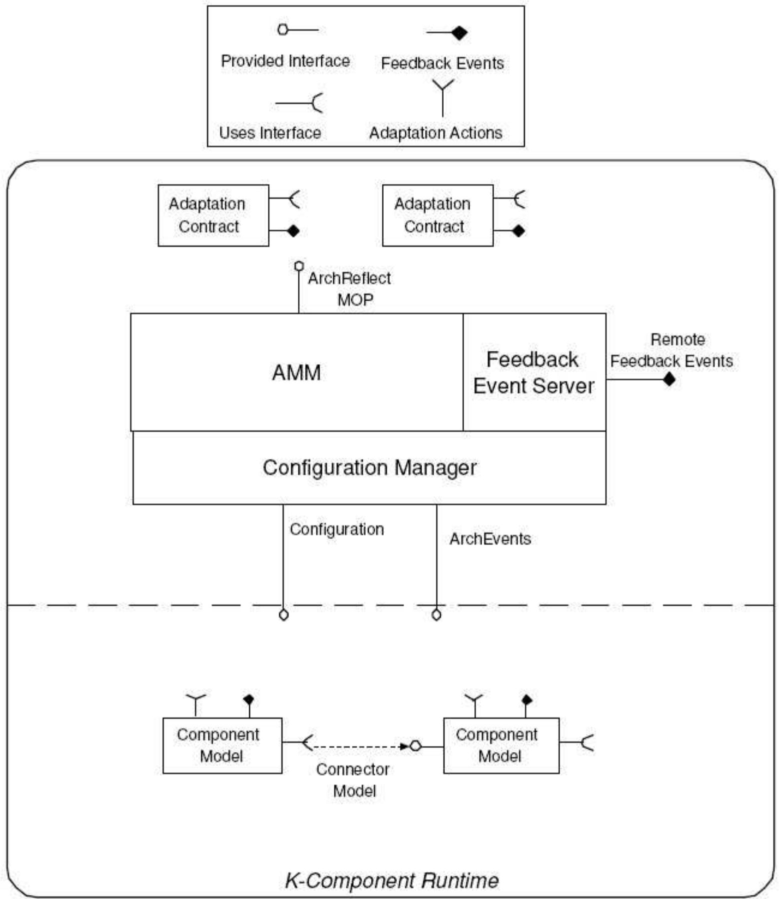


Figure 2.8 *A K-Component runtime (from [27])*

called Collaborative Reinforcement Learning (CRL). CRL enables the decentralised coordination of groups of connected components for the purpose of establishing system-wide properties" [38]. The CRL technique allows similar agents to share their information between one another to help improve their problem solving capabilities (i.e. collaborative feedback). With the sharing of optimal policies at the agent level, a bottom-up approach may be taken to solve complex problems.

The infrastructure for this CRL is provided by the K-Component framework [23]. A K-Component is a runtime within a single address space. K-Components are interconnected using the connector abstraction as defined by Fielding:

> A connector is an abstract mechanism that mediates communication, coordination, or cooperation between components. [50]

"Architectural reflection is used to reify the structure of components and connectors in a K-Component runtime as an [Architectural Meta-Model] AMM" [27]. The AMM is a configuration graph describing components, component interfaces, and component connectors. It "is designed to enable the construction of self-adaptive software in decentralised systems" [27]. Given its decentralised nature, there is no global view of an AMM within the K-Components framework. Each runtime manages an AMM to represent its own interest (internal components, connectors, and ex-

ternal components interacting with components within its runtime). Reconfigurations are limited to “type-safe connector binding and component replacement” [27].

Interaction with the AMM is achieved with the use of the *ArchReflect* MOP. This MOP allows the binding/unbinding and replacement of components and connectors within the AMM and the ability to invoke any methods defined on entities within the AMM. The ArchEvents interface registers entities within the AMM. This architecture is illustrated in Figure 2.8.

The Adaptation Contract Description Language (ACDL) specifies adaptation logic for components and connectors within a K-Component runtime. Within the ACDL, Event Condition Action (ECA) policies are specified with the use of feedback events. “A feedback event is defined as a predicate on the value of a component feedback state and an associated handler” [27]; feedback states are only observable via feedback events. Feedback event communication is handled by the Feedback Event Manager, see Figure 2.8. Within each K-Component runtime, it is possible to subscribe to feedback events from remote runtimes known as remote feedback events. The KBind interface allows remote clients to subscribe to feedback events and the transfer of component descriptions between remote hosts. Figure 2.9 illustrates this process by facilitating the update of a remote component binding.

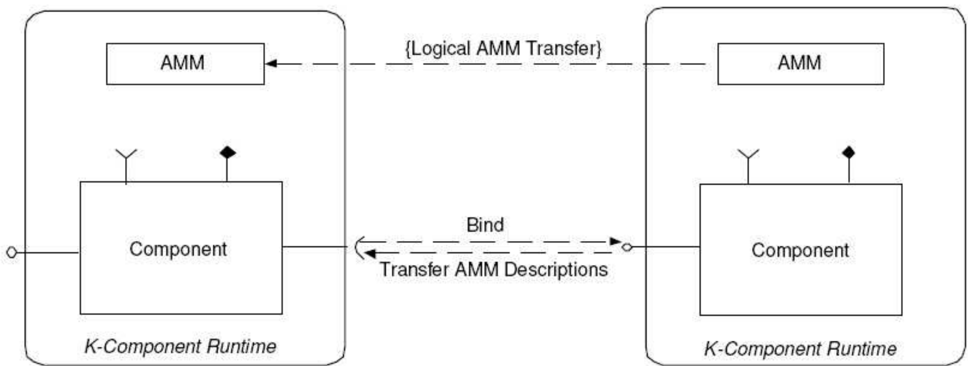


Figure 2.9 *Abstract model of interaction between component binding and AMM transfer between K-Component runtimes (from [27])*

Feedback events and the KBind interface facilitate the construction of coordination protocols between connected K-Components runtimes in a decentralised manner. When examined from a coordination perspective “each K-Component in the decentralised system maintains a local software architecture and a partial view of the system that covers directly connected components on neighbouring nodes” [27]. While this approach is appropriate for large-scale distributed systems, it can limit the level of coordination between participants within the system. With each system only containing a partial system view, it is difficult to create a system-level reflective agent to manage system-wide concerns.

Coordination is further limited by the type of information that can be exchanged between K-Component runtimes; state transfer is limited to feedback events and the transfer of known component definitions. It is not possible to retrieve a list of components within the repository of a remote runtime, as is possible with dynamicTAO’s Distributed Configuration discussed in Section 2.8.4. This limits coordination potential to pre-referenced components, restricting the ability of the framework to perform unanticipated runtime adaptations.

In addition, system-level state exchange is limited: "Feedback states and feedback events can be defined on components, however the only support for a system feedback state is the status state defined on connectors" [27]. This restricts the ability of K-Components to model its underlying infrastructure and constrains its potential to monitor and react to changes within this infrastructure. Such ability was one of the primary reasons the QuO framework was developed, further details on QuO are available in Section 2.8.2.

2.8.6 Other Reviewed Systems

As an addition to in this review, a number of other systems have been examined to place this research within the wider reflective and self-management research community. This review starts with the CodA and GARF projects; both of these works are seen as pioneering landmarks in reflective research.

2.8.6.1 CodA and GARF

The CodA file system is designed as an object meta-level architecture, its primary design goal was to allow for decomposition by logical behaviour. Utilising the principle of decomposition from Object-Oriented (OO) software engineering, CodA eliminates the problems existing in 'monolithic' meta-architectures. CodA achieves this by using multiple meta-objects with each one describing a single small behavioural aspect of an object, instead of using one large meta-object that describes all aspects of an objects behaviour. This approach offers a fine-grained approach to decomposition. Once the distribution concern has been wrapped in meta-objects, aspects of the systems distribution such as message queues, message sending and receiving can be controlled.

GARF [36] (automatic generation of reliable applications) is an object-oriented tool that supports the design and programming of reliable distributed applications. GARF wraps the distribution primitives of a system to create a uniform abstract interface that allows the enhancement of the systems basic behaviour. One such technique improves application reliability by replicating the application's critical components over several machines. Group-communication schemes implement these techniques by multicasting message delivery to the groups of replicas. To facilitate group-communication, multicasting functionality needs to be mixed with application functionally. GARF acts as an intermediate between group-communication functionality and applications, promoting software modularity by clearly separating the implementation of concurrency, distribution, and replication from functional features of the application.

2.8.6.2 Strathclyde Context Infrastructure

The Strathclyde Context Infrastructure (SCI) is a middleware service that provides contextual information (i.e. user preferences, usage history, and environmental constraints) for interactions with users. SCI is focused on the "extraction, placement and management of context in the face of mobility" [51], simplifying the development of context aware applications.

SCI is broken down into two layers. The upper layer manages the interconnection of nodes and utilises an overlay network to provide scalability and robustness, overcoming the limitations of a hierarchical node topology [51]. The lower layer of the infrastructure is concerned with:

> The contents of each node, which consists of entities (People, Software, Places,

Devices and Artefacts) responsible for producing, managing and using contextual information, and is referred to as a range [51].

A *Context Server* within each range provides communication capabilities and access to context utilities. The notion of context services such as SCI is of particular interest to autonomic initiatives and reflective middleware platforms. The benefits of contextual information within user interactions can also be realised within the reflective process via contextual reflection. Enhancing reflective logic with relevant contextual information can improve its ability to adapt its base-level to meet current requirements.

2.8.6.3 RAFDA

The Reflective Architecture Framework for Distributed Applications (RAFDA) [52] is a reflective framework enabling the transformation of a non-distributed application into a flexibly distributed equivalent application. RAFDA allows an application to adapt to its environment by dynamically altering its distribution boundaries. RAFDA can transform a local object into a remote object and vice versa, allowing local and remote objects to be interchangeable.

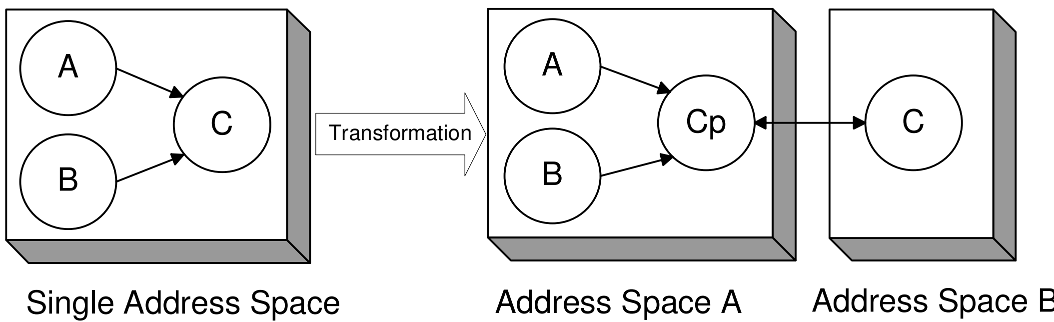


Figure 2.10 *An example RAFDA re-distribution transformation (adapted from [52])*

As illustrated in Figure 2.10, RAFDA achieves flexible distribution boundaries by substituting an object with a proxy to a remote instance. In the example in Figure 2.10, objects A and B both hold references to a shared instance of object C; all objects exist in a single address space (non-distributed). The intention is to move object C to a new address space. RAFDA transforms the application so that the instance of C is remote to its reference holders; the instance of C in address space A is replaced with a proxy, Cp, to the remote implementation of C in address space B.

The process of transformation is performed at the bytecode level. RAFDA identifies points of substitutability and extracts an interface for each substitutable class; every reference to a substitutable class must then be transformed to use the extracted interface. The proxy implementations provide a number of transport options including the Simple Object Access Protocol (SOAP), Remote Method Invocation (RMI), and Internet Inter-ORB Protocol (IIOP). The use of interfaces makes non-remote and remote versions of a class interchangeable thus allowing for flexible distribution boundaries. Policies determine substitutable classes and the transportation mechanisms used for the distribution.

2.8.6.4 Multichannel Adaptive Information Systems

The Multichannel Adaptive Information Systems (MAIS) is designed to create adaptive information systems capable of delivering e-Services using a reflective architecture. The MAIS architecture uses adaptable distribution channels (video, audio, etc) to satisfy QoS demands. The MAIS architecture is broken down into three layers:

> *e-Service composition platform*, which is in charge of receiving the client request, selecting e-Service(s) satisfying it, and invoking the selected e-Service(s).
>
> *Interaction enabling platform*, which is the core of [the] architecture, because it is in charge of collecting constraints from e-Services, clients and context, determining the QoS for each e-Service according to the client profile and selecting the best channel where the e-Service can be delivered.
>
> *Reflective platform*, which is in charge of adapting the selected distribution channel according to the constraints obtained from the Interaction enabling platform and monitoring if the distribution channel along which an e-Service is delivering respects the QoS level chosen by user. [53]

MAIS allows a user to specify the QoS level they desire, the platform then composes a distribution channel to meet these demands; this is a best-effort process. Once in place, the distribution channels QoS is monitored and adapted if necessary to maintain the defined QoS level. Even though the level of interaction is minimal and is user/system centric interaction, MAIS demonstrates the advantages of user direction and the potential of context information exchange (the clients desired QoS) within the reflective process.

2.9 Comparison of Reviewed Systems

Each of the systems examined are designed with different goals in mind. From the perspective of this research, the main objective of this review is to examine each system and consider its capability for runtime coordination with external systems at the meta-level. This analysis can be broken down into the following three steps:

1. Coordination Capability
2. Interaction Protocol
3. Meta-level Access capabilities

Each of these steps builds on the analysis of the preceding step. The remainder of this section presents the results of the survey.

2.9.1 Coordination Capability

The first step within this survey checks for the presence of basic coordination capability. In order to examine a systems coordination capability the following criteria have been identified:

- Coordinated Behaviour – Does the system exhibit any coordinated behaviour?

- Dynamic / Static Participants – Can new participants join the coordination activity at run-time?
- Deployment – What is the topology used within the system?

Table 2.1 presents a summary of coordinated behaviour within each of the projects.

Reviewed System	Coordinated Behaviour	Dynamic/Static Participants	Deployment
mChaRM	Limited to recipient list	Static	Centralised
QuO	Limited to SysCond objects	Static	Centralised
Open ORB	No	No	No
Open ORB (Binding Framework)	Limited to component binding	Dynamic	Centralised
dynamic TAO	Admin protocol (DCP)	Dynamic	Both
K-Components	Feedback Events and KBind interfaces	Dynamic	Decentralised

Table 2.1 *Comparison of coordinated behaviour within reviewed systems*

The overall level of coordinated behaviour within the survey is minimal with only one project, K-Components, exhibiting first-class coordinated behaviour. Limited coordinated behaviour was observed within some aspects of the remaining projects. While Open ORB did not contain any coordinated capabilities within its meta-level, its binding framework does posses coordinated behaviour.

Two of the systems illustrated a dynamic participant capability with respect to their coordinated behaviour. Open ORBs binding framework also displayed limited dynamic capabilities with binding creation. When examined for topological support, only K-Components does not provide centralised support. The next step in the survey focuses on the capabilities of the interaction protocols used to achieve coordinated behaviour.

2.9.2 Interaction Protocol

Coordinated behaviour must be implemented using some form of interaction protocol to facilitate communication between participants. Interaction protocols can vary in nature from simple remote method innovations to elaborate multi-step interactions. This stage of the survey examines the protocols utilised within the reviewed systems, broken down along the following lines:

- Open Accessibility - Can previously unfamiliar systems interact using the protocol?
- Extensible Interaction – Can the protocol be extended to provide additional capabilities?
- Implementation agnostic – Is the protocol capable of working within heterogonous environments and with different technology platforms?
- Independent Control – Does the protocol preserve independent control between participants?

Reviewed System	Open Accessibility	Extensible Interaction	Implementation Agnostic	Independent Control
MChaRM	No	No	No	No
QuO	No	No	No	No
Open ORB	No	No	No	No
Open ORB (Binding Framework)	No	Limited to bindings	No	No
dynamicTAO	Limited to dynamic TAO compatible participants	No	No	No
K-Components	Limited to K-Component runtimes	Connector (Binding) dependant only	No	No

Table 2.2 *Comparison of interaction protocols within reviewed systems*

The results of this examination are presented in Table 2.2.

Only limited interaction support is available within the reviewed systems, providing minimal meta-level interactions. Interactions within mChaRM and QuO were very simplistic offering only basic exchanges between participants, while the binding framework within Open ORB also provided similar capabilities with the capacity for limited interaction extension at the binding-level.

Only two of the reviewed systems, dynamicTAO and K-Components, provided some form of abstract high-level interaction protocol. With DCP, dynamicTAO demonstrates meta-level interactions to simplify administration tasks. The DCP is a straightforward protocol to facilitate remote administration of an ORB and does not possess any facilities to extend its interaction capabilities.

K-Components facilitates interaction as a first-class entity within its runtime to provide meta-information exchange. However, the interaction protocol that it utilises is not designed to facilitate open accessibility and only offers limited extensibility at the connector level, similar to Open ORB.

None of the systems reviewed provided interaction capabilities in a technology neutral manner or preserved participants control independence.

2.9.3 Meta-Level Access Capabilities

The final step in this survey examines the level of access provided by the interaction mechanisms examined in the previous step. Four key capabilities have been identified to measure meta-level accessibility within the reviewed systems:

- State – Can meta-state be transferred using the protocol?
- Realisations – Does the protocol allow an adaptation to be requested and realised by the meta-level?
- Analysis – Will the protocol allow access to the analytical capabilities of the meta-level?
- Event Notification – Is support for meta-level event notifications provided by the protocol?

Reviewed System	State	Realisations	Analysis	Event Notifications
mChaRM	Limited to recipient list	No	No	No
QuO	Shared system state model	Possible coordinated reactions to shared state changes	Limited to syscond analysis	State model events only
Open ORB	No	No	No	No
Open ORB (Binding Framework)	Context information only	Limited to construction of bindings	Binding implementation specific only	Binding implementation specific only
dynamicTAO	Administration state only	Administration only	Object monitor and user supplied	No
K-Components	Feedback events & component descriptions	Component binding connectors only	No	Feedback events only

Table 2.3 *Comparison of meta-level access capabilities within reviewed systems*

As detailed in Table 2.3, meta-level access capabilities vary considerably within the reviewed systems. mChaRM is limited to a very simple state exchange mechanism to facilitate the exchange of recipient lists between loci locations; no support for realisation, analysis, or events are provided.

The QuO project utilises a shared meta-state between clients and servants using SysCond objects. The communal nature of sysconds objects could be exploited to provide a basis for coordinated behaviour between clients and servants. This could be achieved by using SysCond objects to trigger realisation from the client to servant and vice versa. Sysconds also facilitate limited communal state analysis and event notifications between client and servant.

As previously noted, Open ORB does not facilitate interaction at the meta-level. However, limited interaction is present within its binding framework to exchange information on binding implementations. Once this information is exchanged, the receiving binding framework will perform an adaptation to create the binding locally, illustrating a simple realisation. While no support is provided for analysis or event exchange interactions, this functionality could be included within the binding implementation itself.

dynamicTAO offers remote administration facilities with the ability to access the state of an ORB and perform realisations and analysis via DCP. While this scope of the DCP is limited to the configuration of components, it illustrates a more complete interaction protocol when compared to the systems reviewed at this point.

The coordination of distributed components is a key objective of the K-Component framework. The framework allows state exchange in the form of feedback events and component descriptions. A limited form of realisation is available in the form of component binding connectors, similar in concept to the binding framework within Open ORB.

2.9.4 Review Summary

The three stages of this survey have revealed that while some limited cooperation and coordination behaviour exists within current reflective systems, with the exception of K-Components, this behaviour is not seen as a first-class activity within the meta-level. A great deal of the behaviour witnessed can be attributed to the desire for remote system configuration and the infrastructure necessities for distributed component bindings.

Current state-of-the-art systems do not provide full meta-level (state, analysis, realisation, events) access in an implementation agnostic, openly accessible manner. Furthermore, no system offers an extensible interaction protocol in which to create such a capability. Each of the reviewed systems operate within a master/slave environment where participants do not maintain independent control with the ability to refuse a request.

2.10 Summary

Adaptive and reflective techniques have been noted as a key emerging paradigm for the development of dynamic next-generation middleware platforms [12]. These techniques empower a system to automatically self-alter (adapt) to meet its environment and user needs. Adaptation can take place autonomously or semi-autonomously, based on the systems deployment environment, or within the defined policies of users or administrators [17].

The use of a meta-level is a key technique employed in the construction of self-managed systems. Systems built using a meta-level are separated into two levels, a base- and meta-level. The base-level provides system functionality, and the meta-level contains the policies and strategies for the behaviour of the system. The inspection and alteration of this meta-level allows for changes in the system's behaviour. Meta-levels have been developed for a number of middleware platforms such as CORBA ORBs (dynamicTAO and Open ORB), component frameworks (K-Components), and RPC-based distribution mechanisms (mChaRM).

The majority of meta-levels are designed as introverted entities with limited or no coordination capabilities. The interaction mechanisms used do not provide full meta-level (state, analysis, realisation, events) access in an implementation agnostic, openly accessible manner.

Chapter 3

Rudimentary Message-Oriented Middleware

The realm of Message-Oriented Middleware is interaction-centric, providing an ideal environment for the study of coordinated self-managed systems. The goal of this chapter is to introduce the Message-Oriented Middleware domain, its basic concepts, current research activities, and investigate their capability with respect to self-management capabilities.

3.1 Introduction

As software systems continue to be distributed deployments over ever increasing scales, transcending geographical, organisational and traditional commercial boundaries, the demands placed upon their communication infrastructures will increase exponentially. Modern systems operate in complex environments with multiple programming languages, hardware platforms, and operating systems. These systems face requirements including dynamic flexible deployments, 24/7 reliability, high throughput performance, and security while maintaining a high Quality of service (QoS). In these environments the traditional direct Remote Procedure Call (RPC) mechanisms quickly fail to meet the challenges present.

In order to cope with the demands of such environments, an alternative to the RPC distribution mechanism has emerged. This mechanism, called Message-Oriented Middleware or MOM, provides a clean method of communication between disparate software entities. MOM is one of the cornerstone foundations that distributed enterprise systems are built upon. MOM can be defined as any middleware infrastructure that provides messaging capabilities.

A client of a MOM system can send messages to, and receive messages from, other clients of the messaging system. Each client connects to one or more servers that act as an intermediary in the sending and receiving of messages. MOM uses a model with a peer-to-peer relationship between individual clients; in this model, each peer can send and receive messages to and from other client peers. MOM platforms allow the creation of flexible cohesive systems; ones that allow changes in one part of a system to occur without the need for changes in other parts of the system.

This chapter discusses the RPC and MOM distribution mechanisms, providing a through description and comparison of both mechanisms. A number of fundamental MOM concepts such

as queues, filtering, and messaging models are introduced, along with an overview of the Java Message Service API; an API that provides standardised MOM interaction for Java programs. Current MOM research is reviewed to reveal the broad range of multi-purpose implementations available. In particular, the review seeks to identify common messaging and administrative capabilities present within MOM platforms. The chapter commences with an introduction to the main interaction models used within distributed computing.

3.2 Interaction Models

Two interaction models dominate distributed computing environments, synchronous and asynchronous communication. This section introduces both interaction models; a solid knowledge of these models and the differences between them is key to understanding the benefits and differences between MOM and other forms of distribution available.

3.2.1 Synchronous Communication

When using the synchronous interaction model, as illustrated in Figure 3.1, systems do not have processing control independence; they rely on the return of control from the called systems. When a procedure/function/method is called using the synchronous interaction model the caller code must block and wait (suspend processing) until the called code completes execution and returns control to it, the caller code can now continue processing.

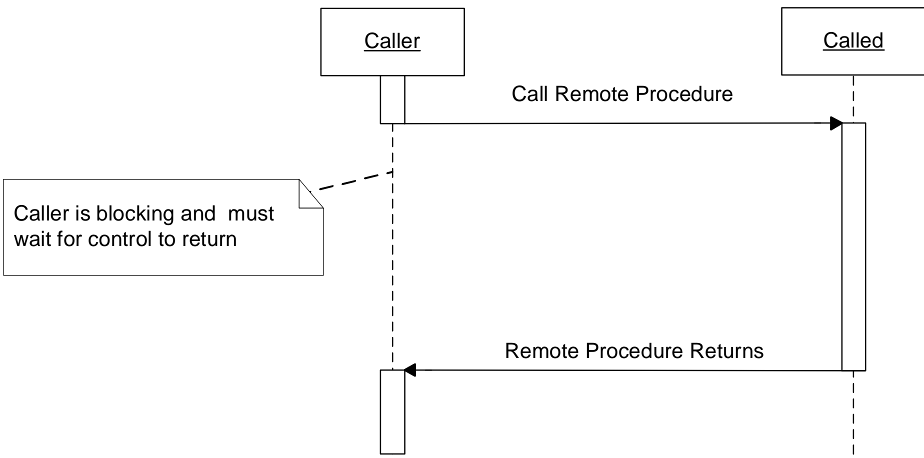


Figure 3.1 *The synchronous interaction model*

3.2.2 Asynchronous Communication

The asynchronous interaction model, illustrated in Figure 3.2, allows the caller to retain processing control. The caller code does not need to block and wait for the called code to return. This model allows the caller to continue processing regardless of the processing state of the called procedure/function/method. With asynchronous interaction, the called code might not execute straight away. This interaction model requires an intermediary to handle the exchange of requests; normally this intermediary is a message queue.

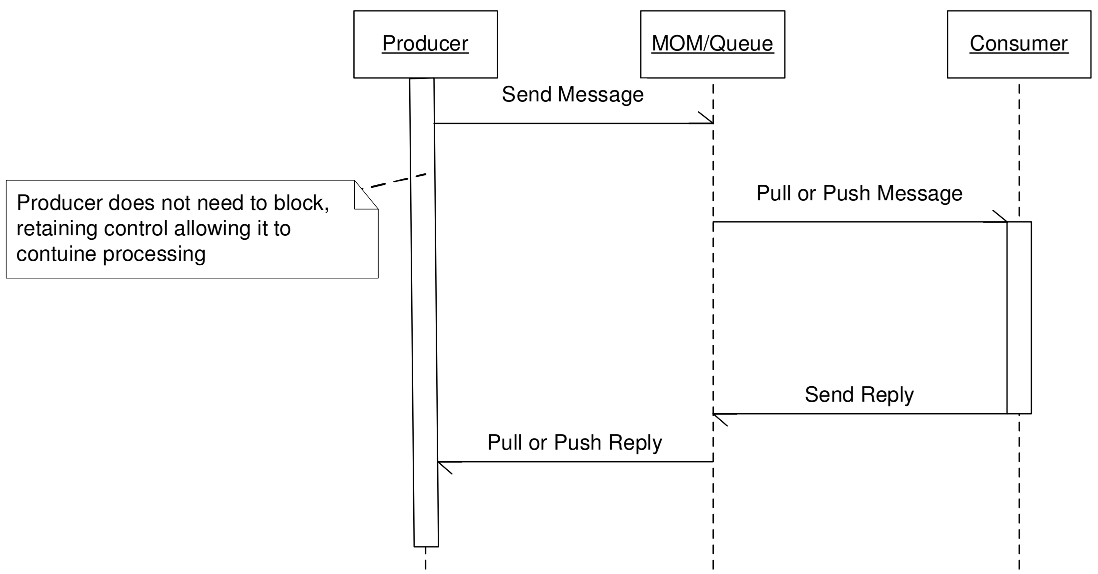


Figure 3.2 *The asynchronous interaction model*

While more complex than the synchronous model, the asynchronous model allows all participants to retain processing independence. Participants can continue processing, regardless of the state of the other participants.

With a clear understanding of both interaction models in place, the next sections introduce the remote procedure call and message-oriented middleware. Both of these distribution mechanisms are examined with respect to coupling, reliability, scalability, and availability.

3.3 Introduction to the Remote Procedure Call (RPC)

The traditional RPC [54, 55] model is a fundamental concept of distributed computing and is utilised within a number of middleware platforms including, CORBA, Java RMI, Microsoft DCOM and XML-RPC. The objective of RPC is to allow two processes to interact. RPC creates the façade of making both processes believe they are in the same process space (i.e. are the one process). Based on the synchronous interaction model, RPC is similar to a local procedure call whereby control is passed to the called procedure in a sequential synchronous manner while the calling procedure blocks waiting for a reply to its call. RPC is analogous to a direct conversation between two parties (similar to a person-to-person telephone conversation). An example of an RPC based distributed system deployment is detailed in Figure 3.3. RPC is now examined with respect to coupling, reliability, scalability, and availability.

3.3.1 Coupling

RPC is designed to work on object or function interfaces, resulting in the model producing tightly coupled systems as any changes to the interfaces will need to be propagated throughout the codebase of both systems. This makes RPC a very invasive mechanism of distribution. As the number of changes to source or target systems increase, the cost will increase too. RPC provides an

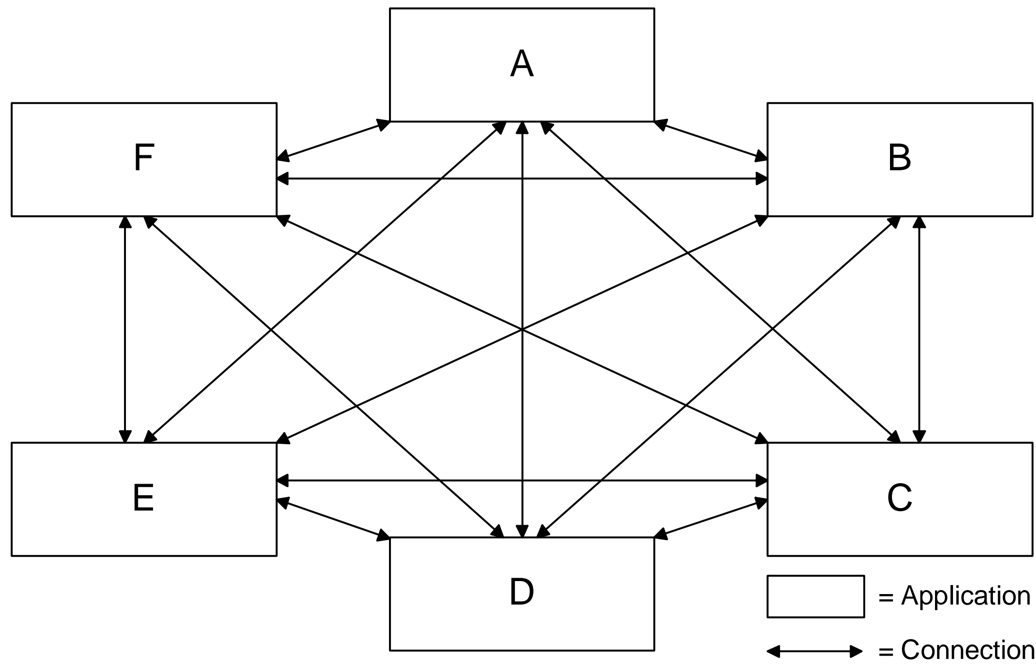


Figure 3.3 *An example remote procedure call deployment*

inflexible method of integrating multiple systems.

3.3.2 Reliability

Reliable communications can be the most important concern for distributed applications. Any failure outside of the application: - code, network, hardware, service, other software or service outages of various kinds (network provider, power, etc), can affect the reliable transport of data between systems. Most RPC implementations provide little or no guaranteed reliable communication capability; they are very vulnerable to service outages.

3.3.3 Scalability

In a distributed system constructed with RPC, the blocking nature of RPC can adversely affect performance in systems where the participating subsystems do not scale equally. This effectively slows the whole system down to the maximum speed of its slowest participant. In such conditions, synchronous based communication techniques such as RPC may have trouble coping when elements of the system are subjected to a high-volume burst in traffic. Synchronous RPC interactions use more bandwidth because several calls must be made across the network to support a synchronous function call. The implication of this supports the use of the asynchronous model as a scaleable method of interaction.

3.3.4 Availability

Systems built using the RPC model are interdependent, requiring the simultaneous availability of all subsystems; a failure in a subsystem could cause the entire system to fail. In an RPC

deployment the unavailability of a subsystem, even temporally, due to service outage or system upgrading can cause errors to ripple throughout the entire system.

3.4 Introduction to Message-Oriented Middleware (MOM)

MOM [56] systems provide distributed communication based on the asynchronous interaction model. This non-blocking model allows MOM to solve many of the limitations found in RPC. Participants in a MOM based system are not required to block and wait on a message send, messages when the sender or receiver is not active or available to respond at the time of execution.

MOM supports message delivery for messages that may take minutes to deliver, as opposed to mechanisms such as RPC (i.e. Java RMI) that deliver in milliseconds or seconds. When using MOM, a sending application has no guarantee that its message will be read by another application nor is it given a guarantee about the time it will take the message to be delivered. The receiving application determines these aspects of the system.

MOM-based distributed system deployments, as shown in Figure 3.4, offer a service-based approach to inter process communication. MOM messaging is similar to the postal service. Messages are delivered to the post office; the postal service then takes responsibility for safe delivery of the message [57]. MOM is now examined with respect to coupling, reliability, scalability, and availability.

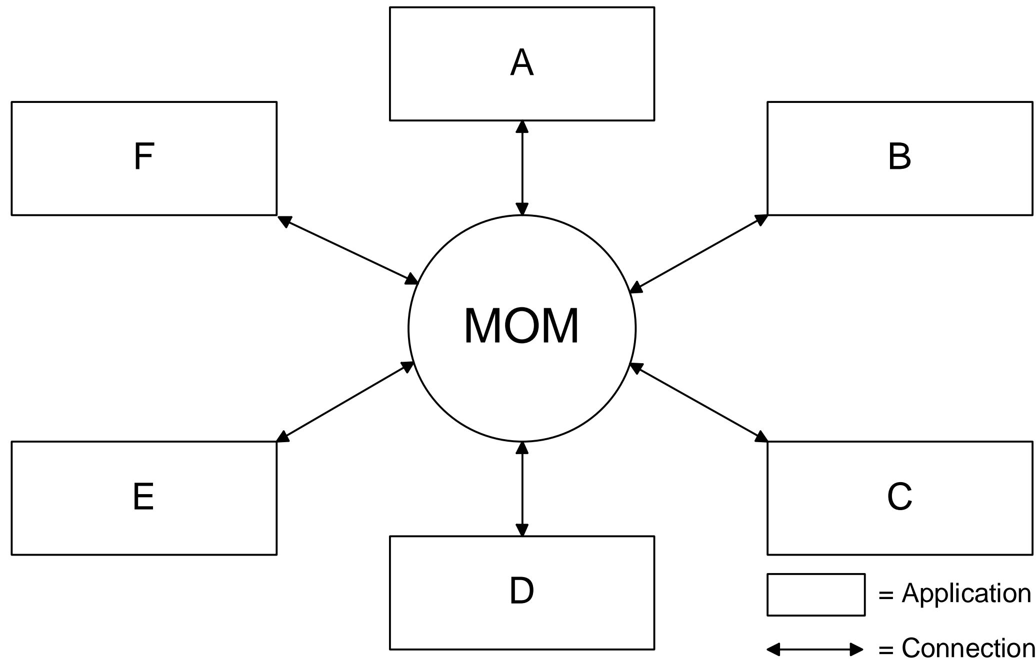


Figure 3.4 *An example Message-Oriented Middleware deployment*

3.4.1 Coupling

MOM injects an independent intermediary between message senders and receivers to facilitate message exchange. An illustration of this concept is provided in Figure 3.4. A primary benefit

of MOM is the loose coupling between participants in a system - the ability to link applications without having to adapt the source and target systems to each other, resulting in a highly cohesive, decoupled system deployment [56].

3.4.2 Reliability

MOM prevents message loss through network or system failure by using a store and forward mechanism for message persistence. This capability of MOM introduces a high-level of reliability into the distribution mechanism. Store and forward prevents loss of messages when parts of the system are unavailable or busy. The specific level-of-reliability is typically configurable, but MOM messaging systems are able to guarantee that a message will be delivered, and that it will be delivered to each intended recipient exactly once.

3.4.3 Scalability

In addition to decoupling the interaction of subsystems, MOM also decouples the performance characteristics of the subsystems from each other. Subsystems can scale independently, with little or no disruption to other subsystems. MOM also allows the system to cope with unpredictable spikes in activity in one subsystem without affecting other areas of the system. MOM messaging models contain a number of natural traits that allow for simple and effective load balancing, by allowing a subsystem to choose to accept a message when it is ready to do so rather than being forced to accept it. This load balancing technique is covered in more detail in the Section 3.7.3.

State-of-the-art enterprise-level MOM platforms have been used as the backbone to create massively scalable systems. Within one study of massive scalability, a real-time customer self-service portal was developed. This system scaled to handle more than 16.2 million customer requests per hour with over 270,000 new order requests per hour [58].

3.4.4 Availability

MOM introduces high availability capabilities into systems allowing for continuous operation and smoother handling of system outages. MOM does not require simultaneous or "same-time" availability of all subsystems. Failure in one of the subsystems will not cause failures to ripple throughout the entire system. MOM can also improve the response time of the system, by reducing temporal coupling to slower subsystems. This can reduce the process completion time and improve overall system responsiveness and availability.

3.5 When to use MOM or RPC

Depending on the deployment scenario, both MOM and RPC have their advantages and disadvantages. RPC provides a more straightforward approach to distribution using the familiar and straightforward synchronous interaction model. However, the RPC mechanism suffers from inflexibility and tight coupling (with potential geometric growth of interfaces) between the communicating systems. It is also problematic to scale parts of the system and deal with service outages. RPC deployments are temporally coupled and assume that all parts of the system will be simultaneously

available. If one part of the system were to fail, or even become temporarily unavailable (network outage, system upgrade), the entire system could stall as a result.

There is a large overhead associated with an RPC interaction; RPC calls require more bandwidth than a similar MOM interaction. Bandwidth is an expensive performance overhead and is the main obstacle to scalability of the RPC mechanism [59]. The RPC model is designed on the notion of a single client talking to a single server. Traditional RPC has no built in support for one-to-many communications. The advantage of an RPC system is the simplicity of the mechanism and straightforward implementation.

The RPC model is ideal if you want a strongly-typed/Object-Oriented (OO) system. A strongly-typed system can produce more stable code and relieve the developer from having to create infrastructure level code needed to move data into strongly-typed language level objects. A strongly typed interface definition can facilitate greater compile time assistance following Bjarne Stroustrup's mantra of preferring compile-time errors to runtime errors [60].

An important advantage RPC has over MOM is the guarantee of sequential processing. With the synchronous RPC model you can control the order in which processing occurs in the system. For example, in an RPC system you can be sure that at any one time all the new orders received by the system have been added to the database and that they have been added in the order in which they were received. However, with an asynchronous MOM approach this cannot be guaranteed, as new orders could exist in queues waiting to be added to the database. This could result in a temporal inaccuracy of the data in the database. There is no concern that these updates will not be applied to the database, however a snapshot of the current database would not accurately reflect the actual state of orders placed. RPC is slow but consistent; work is always carried out in the correct order. These are important considerations for sections in a system that requires data to have 100% temporal integrity. If this type of integrity is more important than performance, you will need to use the RPC model or else design the system to check for potential temporal inaccuracies.

MOM allows a system to evolve with its operational environment without the need for dramatic changes to its application assets. It provides an integration infrastructure that accommodates functionality changes over time without disruption or compromise on performance and scalability. The decoupled approach of MOM allows for flexible integration of clients, support for large numbers of clients, and client anonymity. Commercial MOM implementations provide high scalability with support for tens of thousands of clients, advanced filtering, integration into heterogeneous networks and clustering reliability [58].

The RPC model is ideal if you want a strongly-typed/Object-Oriented (OO) system with tight coupling, compile-time semantic checking, and an overall more straightforward system implementation.

MOM is the ideal solution if the distributed system will be a geographically dispersed deployment with poor network connectivity and stringent demands in reliability, flexibility, and scalability.

3.6 Message Queues

The message queue is a fundamental concept within MOM. Queues provide the ability to store messages on a MOM platform and are central to the implementation of the asynchronous inter-

action model within MOM. A queue, as shown in Figure 3.5, is a destination to which clients may send messages and receive messages from. Messages contained within a queue are sorted in a particular order, the standard queue found in a messaging system is the First-In First-Out (FIFO) queue. As the name suggests the first message sent to the queue is the first message retrieved from the queue.

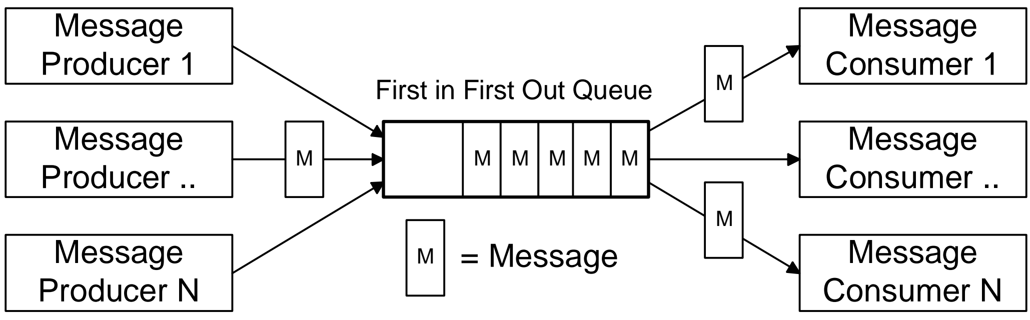


Figure 3.5 *The role of a message queue*

There is no constraint on queue usage within a deployment. Depending on requirements, applications may have their own queue, or they may share a queue, there is no restriction on the set-up. Queuing is of particular benefit to clients without constant network connectivity such as sales personnel on the road accessing a mobile network (GSM, GRPS, etc) to remotely send orders to head office or for remote sites with poor communication infrastructures. These clients can use a queue as a makeshift inbox, periodically checking the queue for new messages when network connectivity exists.

Queue Type	**Purpose**
Public Queue	Public open access queue.
Private Queue	Require clients to provide a valid username and password for authentication and authorisation.
Temporary Queue	Queue created for a finite period, this type of queue will only last for the duration of a particular condition or a set time-period.
Journal Queue	Designed to keep a record of messages or events. These queues maintain a copy of every message placed within them, effectively creating a journal of messages.
Connector/Bridge Queue	Enables proprietary MOM implementation to interoperate by mimicking the role of a proxy to an external MOM provider. A bridge handles the translation of message formats between different MOM providers, allowing a client of one provider to access the queues/messages of another.
Dead-Letter/Dead-Message Queue	Messages that have expired, or are undeliverable (i.e. invalid queue name or undeliverable addresses), are placed in this queue.

Table 3.1 *Message queue formats*

Typically, MOM platforms support multiple queue types, each with a different purpose. Table 3.1 provides a brief description of commonly available queues. Many attributes of a queue may be

configured including the queue's name, queue's size, the save threshold of the queue, and message sorting algorithm.

3.7 Messaging Models

A solid understanding of the available messaging models within MOM is key to appreciate the unique capabilities it provides. Two main message models are commonly available, the *Point-to-Point* and *Publish/Subscribe* models. The exchange of messages through a destination (channel/queue/topic) is the basis for both of these models. A typical messaging solution will utilise a mix of these models to achieve different messaging objectives. The remainder of this section describes both messaging models and concludes with a comparison.

3.7.1 Point-to-Point

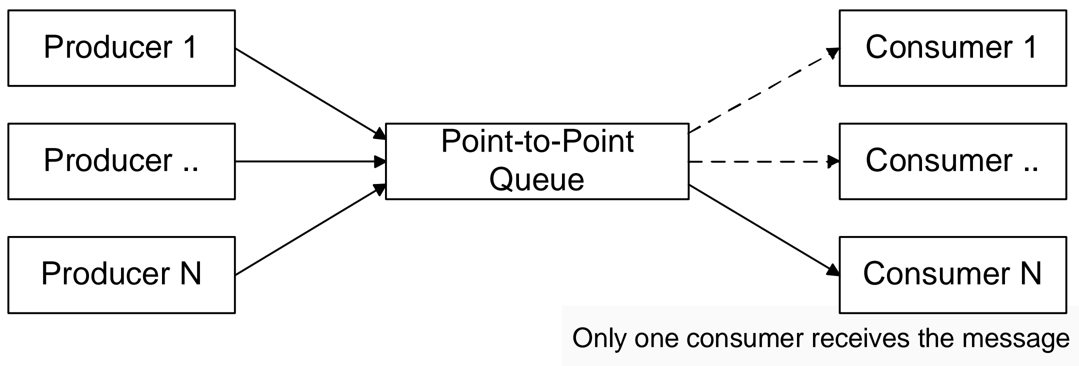


Figure 3.6 *The point-to-point messaging model*

The *Point-to-Point* messaging model provides a straightforward asynchronous exchange of messages between software entities. In this model, shown in Figure 3.6, messages from producing clients are routed to consuming clients via a queue. As discussed in Section 3.6, the most common queue used is a FIFO queue, messages are sorted in the order in which they were received by the message system, and as messages are consumed, they are removed from the head of the queue.

While there is no restriction on the number of clients that can publish to a queue, there is usually only a single consuming client, although this is not a strict requirement. Each message is delivered only once to only one consumer; the model allows multiple consumers to connect to the queue but only one of the consumers will receive the message. This is commonly referred to as *once-and-once-only-messaging*. The technique of using multiple consuming clients to read from a queue can be used to easily introduce smooth, efficient load balancing into a system. In the point-to-point model, messages are always delivered and will be stored in the queue until a consumer is ready to retrieve them.

Request-Reply Messaging Model

This model is designed around the concept of a request with a related response. The request-reply model is used for the World Wide Web (WWW). A client requests a

page from a server, and the server replies with the requested web page. The model requires that any producer who sends a message must be ready to receive a reply from consumers at some stage in the future. The model is easily implemented with the use of the point-to-point or publish/subscribe models and may be used in tandem to complement both models.

3.7.2 Publish/Subscribe

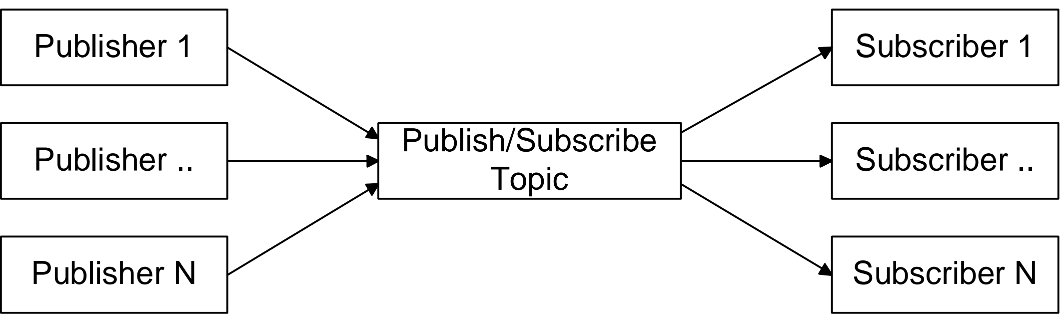


Figure 3.7 *The publish/subscribe messaging model*

The *Publish/Subscribe* messaging model, Figure 3.7, is a very powerful mechanism used to disseminate information between anonymous message consumers and producers. This one-to-many and many-to-many distribution mechanism allows a single producer to send a message to one consumer or potentially hundreds of thousands of consumers.

In the publish/subscribe or "pub/sub" model, the sending and receiving application is free from the need to understand anything about the target application. It only needs to send its information to a destination within the publish/subscribe engine. The engine will then send it to the consumer. Clients producing messages 'publish' to a specific topic or channel, these channels are then 'subscribed' to by clients wishing to consume messages. The service routes the messages to consumers based on the topics to which they have subscribed an interested in. Some publish/subscribe engines don't use the concepts of a destination and route messages directly between publishers and subscribers. Within these systems, message exchange between publishers and subscribers is based purely on subscription constraints.

Within the publish/subscribe model there is no restriction on the role of a client. A client may be both a producer and consumer. A number of methods for publish/subscribe messaging have been developed which support different features, techniques and algorithms for message filtering [61], publication, subscription, subscription management distribution [62], and facilitation within the agent-oriented paradigm [63].

<u>PUSH and PULL</u>

Within both messaging models a consuming client has two methods of receiving messages from the MOM provider.

Pull **-** A consumer can poll the provider to check for any messages, effectively *pulling* them from the provider.

Push **-** Alternatively, a consumer can request the provider to forward relevant messages as soon as the provider receives them; they instruct the provider to *push* messages to them.

3.7.2.1 Destination Hierarchies

Destination or topic hierarchies are a destination grouping mechanism within the publish/subscribe messaging model. This type of structure allows destinations to be defined in a hierarchical fashion, so that they may be nested under other destinations. Each sub-destination offers a more granular selection of the messages contained in its parent. Clients of destination hierarchies subscribe to the most appropriate level of destination to receive the most relevant messages. In large-scale systems, the grouping of messages into related types (i.e. into channels) helps to manage large volumes of different messages [64].

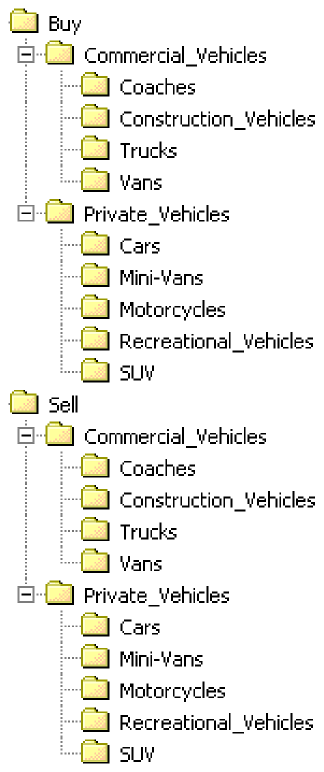


Figure 3.8 *An automotive destination hierarchy structure*

The relationship between a destination and sub-destination(s) allows for super-type subscriptions, where subscriptions that operate on a parent destination/type will also match all subscriptions of descendant destinations/types. A destination hierarchy for an automotive trading service may be structured by categorising messaging into buys or sells, then further sub-categorisation breaking down for commercial and private vehicle types. An example hierarchy illustrating this categorising structure is presented in Figure 3.8. A subscription to the "Sell.Private_Vehicles" channel would receive all messages classified as a private vehicle sale, whereas subscribing to "Sell.Private_Vehicles.Cars" would result in only receiving messages classified as a car sale.

Hierarchical destinations require the destination namespace schema be both well defined and universally understood by the participating parties. Responsibility for choosing a destination in which to publish messages is left to the publishing client. Hierarchical destinations are used in routing situations that are more or less static. Consumers of the hierarchy are able to browse the hierarchy and subscribe to destinations.

Frequently used in conjunction with the publish/subscribe messaging model, hierarchical destinations allow for the dissemination of information to a large number of unknown consumers. Hierarchical destinations can compliment filtering as a mechanism of routing relevant messages to consumers. They provide a more granular approach to consumer subscription that reduces the number of filters needed to exclude unwanted messages, while supporting highly flexible, easy-access, subject-based routing.

3.7.3 Comparison of Messaging Models

The two models have very different capabilities and most messaging objectives can be achieved using either model or a combination of both. The fundamental difference between the models is that within the publish/subscribe model every consumer to a topic/channel will receive a message published to it, whereas in point-to-point model only one consumer will receive it. Publish/subscribe is normally used in a broadcast scenario where a publisher wishes to send a message to one-to-many clients. The publisher has no real control over the number of clients who receive the message, nor have they a guarantee any will receive it. Even in a one-to-one messaging scenario, topics can be useful to categorise different types of messages. The publish/subscribe model is the more powerful messaging model for flexibility; the disadvantage is its complexity.

In the point-to-point model, multiple consumers may listen to a queue, although only one consumer will receive each message. However, point-to-point will guarantee that a consumer will receive the message, storing the messages in a queue until a consumer is ready to receive the message; this is known as *'once-and-once-only'* messaging. While the point-to-point model may not be as flexible as the publish/subscribe model, its power is in its simplicity.

A common application of the point-to-point model is for load balancing. With multiple consumers receiving from a queue, the workload for processing the messages is distributed between the consumers of the queue. In the pull model, the exact order of how messages are assigned to consumers is specific to the MOM implementation, but if you utilise a pull model, a consumer will only receive a message when they are ready to process it. This provides an effective method of load balancing.

3.8 Message Filtering

Message filtering allows a message consumer/receiver to be selective about the messages it receives from a MOM. This section introduces the basic theory of message filters and describes the common forms found within MOM platforms. The techniques of filter covering and filter merging, used to increase efficiency within content-based routing, are also discussed.

Filtering can operate on a number of different levels. Filters use Boolean logic expressions to declare messages of interest to the client, the exact format of the expression depends on the implementation but the WHERE clauses of SQL-92 standard (or a subset of) is commonly used as

the syntax. Filtering models commonly operate on the properties (name/value pairs) of a message. However, a number of projects have extended filtering to message payloads [65].

The overall efficiency of any service and its routing of messages are affected by the power of the language used to construct messages or notifications and to express filters and patterns. As the power of the language increases, so does the complexity of the processing.

Within the MOM domain, the abstract concept of a message is often referred to as an event or notification. While subtle differences exist between these classifications, within the context of this discussion they are interchangeable terms.

Filter Type	**Description**
Channel-based	Channel based filtering categorise messages into pre-defined groups. Consumers subscribe to the group(s) of interest and receive all messages sent to the group(s).
Topic-based / Subject-based	Messages are enhanced with a tag describing its subject or topic. Subscribers can declare their interests in these subjects flexibly by using a string pattern match on the subject, e.g. all messages with a subject/topic of "Car for Sale".
Content-based / Attribute-based	As an attempt to overcome the limitations on subscription declarations, content-based filtering, also referred to as attribute-based filtering, allows subscribers to use flexible querying languages to declare their interests with respect to the contents of the messages. An example query is: Give the price of stock 'SUN' when the volume is over 10000. This query in SQL-92 is: "stock_symbol = 'SUN' AND stock_volume > 10000".
Content/attribute-based with patterns	Content/attribute-based filtering with patterns enhances content-based filtering with additional functionality for expressing user interests across multiple messages. An example query is: Give the price of stock 'SUN' when the price of stock 'Microsoft' is less than $50.
Composite Events	Composite events [66] provide a higher-level abstraction for pattern-based event subscriptions. Composite events encapsulate a pattern of low-level events, and are published when the given pattern occurs. From the client's perspective, composite events simplify complex multi-event pattern-based subscriptions.

Table 3.2 *Message filtering types*

Listed in Table 3.2 are common filtering capabilities found within messaging systems, listed in order of increasing sophistication. It is useful to note that as the filtering techniques get more advanced they are usually able to replicate the techniques which proceed them. For example, subject-based filtering is able to replicate channel-based filtering, just as content-based filtering is able to replicate both subject and channel-based filtering.

3.8.1 Covering & Merging

The techniques of filter covering and filter merging can increase the efficiency of content-based routing solutions. Covering and merging of filters can be used to reduce the size of the routing table and the overhead associated with routing tasks [62, 65, 67, 68]. Before it is possible to describe these techniques, a formal definition of a filter and a notification is required:

> A filter, *F*, is a stateless Boolean function that is applied to a notification n (i.e. message). A notification matches a filter if *F(n)* evaluates to *true*

Each message consumer may have multiple active filters (subscription constraints) that may be interpreted disjunctively. The notification service has a responsibility to only deliver all messages that match the subscription constrains of the consumer.

3.8.1.1 Covering

A covering relationship between two filters exists when one of the filters defines a subset of the constraints in the other *F1* covers *F2* if and only if $N(F1) \supseteq N(F2)$. To illustrate the concept an example is provided:

> Filter A [Vehicle_Type = {'SUV', 'Motorcycle'} AND Colour = 'Blue']
>
> Filter B [Vehicle_Type = 'SUV' AND Colour = 'Blue' AND Mileage =< 60000]

In this example, Filter A covers Filter B because any message that matches Filter A will also match Filter B.

3.8.1.2 Merging

The objective of filter merging is to generate a new filter from a given set of filters. The new filter must cover the set from which it is derived. In order to illustrate this point, Filter A and Filter B from the previous section can be merged to create a new Filter AB:

> Filter AB [Vehicle_Type = {'SUV', 'Motorcycle'} AND Colour = 'Blue'
> AND Mileage =< 60000]

"The aim of filter merging is to generate a filter *F* so that N(F) is a superset of N(F1) .. N(Fn) for a set of filters {F1, ..., Fn}" [65]. Since a merged filter covers the filters from which it was generated, its ancestor filters can be dropped from the routing table, reducing the table's size. Further information on filter covering and merging techniques is available [62, 65, 67, 69].

3.9 Java Message Service

A diverse range of MOM implementations exist that include WebSphere MQ (formerly MQSeries) [70], TIBCO [71], SonicMQ [72], Herald [73], Hermes [64], SIENA [62], Gryphon [74], JEDI [75], REBECCA [76], and OpenJMS [77]. As a means of simplifying the development of systems utilising MOMs, a standard was needed to provide a universal interface to MOM interactions. To date, a number of MOM standardisations have emerged such as the CORBA Event Service, CORBA Notification Service, and most notably the Java Message Service (JMS) [78]. The JMS provides a common way for Java programs to create, send, receive, and read an enterprise messaging system's messages. The JMS provides a solid foundation for the construction of a messaging infrastructure that can be applied to a wide range of applications. This section briefly introduces the JMS and provides an overview of the JMS programming model.

The JMS specification defines a general purpose Application Programming Interface (API) to an enterprise messaging service and a set of semantics that describe the interface and general

behaviour of a messaging service. The goal of the JMS specification is to provide a universal way to interact with multiple heterogeneous messaging systems in a consistent manner. The learning curve associated with many proprietary-messaging systems can be steep, thus the powerful yet simple API defined in the JMS specification can save a substantial amount of time for developers in a pure Java environment.

The specification also defines a Service Provider Interface (SPI). The role of the SPI is to allow MOM developers to hook up their proprietary MOM implementation to the API. This allows you to write code once using the API and plug-in the desired MOM provider, making client-messaging code portable between MOM providers that implement the JMS specification, reducing vendor lock-in and offering a choice for message provision. It should be noted that the JMS is an API specification and does not define the implementation of a messaging service. The semantics of message characteristics such as reliability, performance, and scalability are not fully defined. In addition, the JMS specification does not define an *'on-the-wire'* transportation format for messages. Essentially, two JMS compatible MOM implementations cannot talk to each other directly and will need to use a tool such as a connector/bridge queue to enable interoperability.

3.9.1 Programming using the JMS API

The JMS API provides basic MOM interaction functionality through its programming model, illustrated in Figure 3.9, allowing it to be compatible with most MOM implementations. This section gives a brief overview of the programming model.

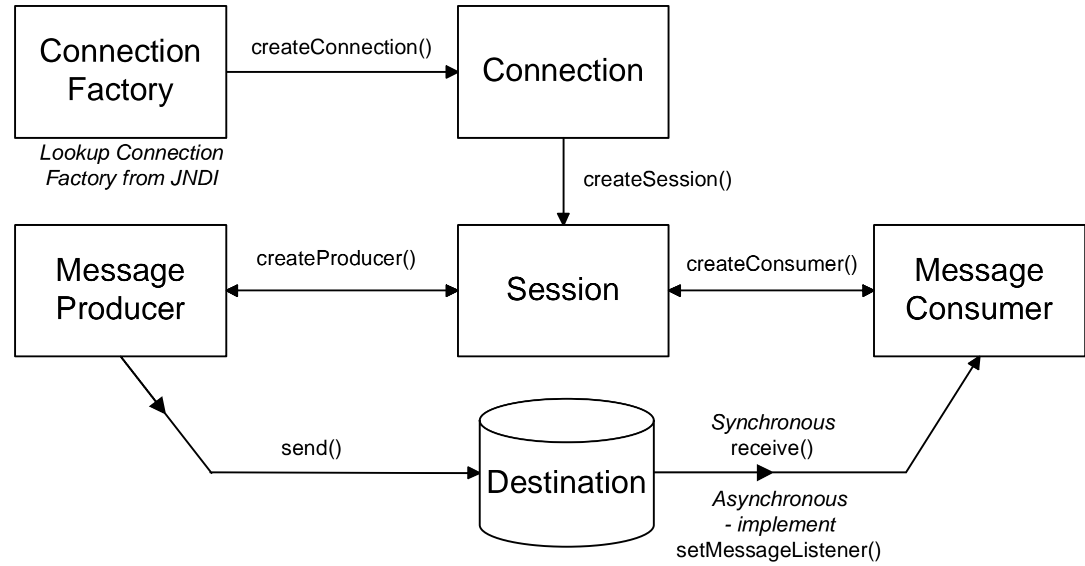


Figure 3.9 *The JMS API programming model (adapted from [79])*

3.9.1.1 Connections and Sessions

When a client wants to interact with a JMS compatible MOM platform, it must first make a connection to the message broker. Using this connection, the client may create one or more sessions. A JMS session is a single-threaded context used to send and receive messages to and from queues

and topics. Each session can be configured with individual transactional and acknowledgements modes.

3.9.1.2 Message Producers and Consumers

In order for a client to send a message to, or receive a message from a JMS provider, it must first create a message producer or message consumer from the JMS session. Within the publish/subscribe model a *javax.jms.TopicPublisher* sends messages to a topic and a *javax.jms.TopicSubscriber* to receive. For the point-to-point model, the j*avax.jms.QueueSender* and *javax.jms.QueueReceiver* send and receive messages respectively.

3.9.1.3 Receive Synchronously and Asynchronously

The JMS API supports both synchronous and asynchronous message delivery. To synchronously receive a message the *receive()* method of the message consumer is used. The default behaviour of this method is to block until a message has been received, however this method may be passed a *time-out* value to limit the blocking period. To receive a message asynchronously, an application must register a *Message Listener* with the message consumer. Message listeners are registered with a message consumer object by using the *setMessageListener(javax.jms.MessageListener msgL)* method. A message listener must implement the *javax.jms.MessageListener* interface. Further detailed discussion and explanations of the JMS API are available in [79, 80].

3.9.1.4 Setting Message Properties

Message properties are optional fields contained in a message. These user-defined fields can be used to contain information relevant to the application or to identify messages. Message properties are used in conjunction with message selectors to filter messages and are sometimes referred to as message attributes.

3.9.1.5 Message Selectors

Message selectors, a form of attributed-based filtering, filter the messages received by a message consumer, they assign the task of filtering messages to the JMS provider rather than to the application. The message consumer will only receive messages whose headers and properties match the selector. A message selector cannot select messages based on the content of the message body/payload. Message selectors consist of a string expression based on a subset of the SQL-92 conditional expression syntax:

"Property_Vehicle_Type = 'SUV' AND Property_Mileage =< 60000"

3.9.1.6 Acknowledgement Modes

The JMS API supports the acknowledgement of the receipt of a message. Acknowledgement modes are controlled at the sessions level, supported modes are listed in Table 3.3.

Acknowledgements Modes	**Purpose**
AUTO_ACKNOWLEDGE	Automatically acknowledges receipt of a message. In asynchronous mode, the handler acknowledges a successful return. In synchronous mode, the client has successfully returned from a call to receive().
CLIENT_ACKNOWLEDGE	Allows a client to acknowledge the successful delivery of a message by calling its acknowledge() method.
DUPS_OK_ACKNOWLEDGE	A lazy acknowledgment mechanism that is likely to result in the delivery of message duplicates. Only consumers that can tolerate duplicate messages should use this mode. This option can reduce overhead by minimising the work to prevent duplicates.

Table 3.3 *JMS acknowledgement modes*

3.9.1.7 Delivery Modes

The JMS API supports two delivery modes for a message. The default *Persistent* delivery mode instructs the service to ensure that a message is not lost due to system failure. A message sent with this delivery mode is placed in a non-volatile memory store. The second option available is the "*Non_Persistent*" delivery mode; this mode does not require the service to store the message or guarantee that it will not be lost due to system failure. This is a more efficient delivery mode because it does not require the message to be saved to non-volatile storage.

3.9.1.8 Priority

The priority setting of a message can be adjusted to indicate to the message service an urgent message that should be delivered first. There are ten levels of priority ranging from 0 (lowest-priority) to 9 (highest-priority).

3.9.1.9 Time-to-Live

JMS messages contain a use-by or expiry time known as the *Time-to-Live* (TTL). By default, a message never expires; however, you may want to set an expiration time. When the message is published, the specified TTL is added to the current time to give the expiration time. Any messages not delivered before their specified expiration times are destroyed.

3.9.1.10 Message Types

The JMS defines five message types, listed in Table 3.4, which allow you to send and to receive data in multiple formats. The JMS API provides methods for creating messages of each type and for filling in their contents.

3.9.1.11 Transactional Messaging

JMS clients can include message operations (sending and receiving) in a transaction. The JMS API session object provides commit and rollback methods that are used to control the transaction from a JMS client. Further detail on transactional messaging is available [81, 82].

Message Type	Message Payload Contains
javax.jms.TextMessage	A java.lang.String object
javax.jms.MapMessage	A set of name/value pairs, with names as strings and values as java primitive types. The entries can be accessed by name.
javax.jms.BytesMessage	A stream of uninterrupted bytes.
javax.jms.StreamMessage	Stream of Java primitive values, filled and read sequentially.
javax.jms.ObjectMessage	A Serializable Java object.

Table 3.4 *JMS message types*

3.10 Current MOM Platforms

The objective of this review is to investigate state-of-the-art MOM platforms to reveal the broad range of multi-purpose implementations available. In particular, the review seeks to identify common messaging and administrative capabilities present within MOM platforms. The review covers a number of MOM platforms from the pioneering commercial developments of TIBCO Rendezvous [71] and IBM MQSeries [70] to innovative academic implementations such as SIENA [62, 67] and Hermes [83]. In addition, the review includes well-known commercial and open-source platforms such as SonicMQ [84] and ActiveMQ [85].

3.10.1 CORBA Event Service & Notification Service

The Object Management Group's (OMG) CORBA *Event Service* specification defines a general purpose communication service that supports the asynchronous exchange of event messages between clients [86]. In this model, event suppliers produce event messages for event consumers to receive. This exchange is achieved with the use of an *Event Channel*; a mediator that propagates events to consumers on behalf of suppliers. Communication between suppliers and consumers may use one of the following two modes:

- Pull Mode – Consumers request data from the supplier using the event channel
- Push Mode – Suppliers push data to a consumer using the event channel

The event service specification defines an EventChannel interface that provides the basic administrative capabilities needed to interact with the channel. The three administrative operations provided by the interface are adding consumers, adding suppliers, and destroying the channel.

The OMG *Notification Service* is designed to enhance the event service by introducing the concepts of filtering and the configurability of QoS requirements [87]. Consumers of event notification channels can specify filters to employ on the channel. The notification service attempts to preserve all the semantics specified for the event service, allowing for interoperability between basic event service clients and notification service clients.

The notification specification includes a brief description of basic administration interfaces for event notification channels, using this interface it is possible to create a new notification channel and obtain a list of all notification channels with associated consumers and suppliers. The ad-

ministration interfaces also support the manipulation of QoS and filtering characteristics of event delivery.

Asynchronous communication within both the event and notification services uses the CORBA synchronous method invocation protocol. This pseudo asynchronicity requires a costly remote method invocation for every event delivered, reducing the scalability of the deployment.

The simplicity of the concept within these specifications has led to widespread acceptance and multitudes of implementations have emerged, including commercial offerings from TIBCO, IBM (MQSeries), and others. The majority of implementations have extended the basic concepts of the specifications and include support for additional proprietary features such as bridges to enable interoperability between disparate implementations, transactional messaging, clustering, and security capabilities. A significant number of the research-based MOM implementations build on the event and notification service specifications.

3.10.2 TIBCO Rendezvous

As the name suggests TIBCO *Rendezvous* [71] is designed to allow programs to locate other clients without the need to determine their network address. Rendezvous, developed by Dale Skeen in the early '90s, is a cornerstone in the foundation of the publish/subscribe messaging model, of which anonymous participants is a fundamental characteristic. Rendezvous provides subject-based addressing which is similar to hierarchical topic namespaces within the JMS API. This addressing schema allows the definition of a namespace that message participants can search with the use of wildcard operators (*, >, <, etc). Rendezvous runs within a Rendezvous daemon on each machine within the messaging solution.

> The Rendezvous daemon runs on each participating computer on your network. All information that travels between program processes passes through the Rendezvous daemon as the information enters and exits host computers. The daemon also passes information between program processes running on the same host. [88]

This federated approach to messaging provision eliminates bottlenecks and single points of failure within a system deployment. Rendezvous support both publish/subscribe and point-to-point messaging; point-to-point messaging is achieved by using a unique "inbox name" address.

Rendezvous supports a large number of programming languages including C, C++, Java, COM, and Perl 5. Each of these languages has a proprietary interface of the generalised Rendezvous API used to connect to the Rendezvous daemon. It should be noted that the Java interface is not JMS compatible. The *Rendezvous Daemon*, illustrated in Figure 3.10, sits between clients and the network to provide the infrastructure for message transportation, including data transmission, packet ordering, receipt acknowledgement, retransmission requests and routing instructions. Run-time configuration of a daemon is available through the *DaemonManager* and *DaemonProxy* APIs. These APIs allow access to the internal configuration of the daemon, facilitating the reconfiguration of its communication transports and routing behaviours. In addition, the APIs also provide access to internal daemon information on aspects such as logging, http tunnelling, fault tolerance, security, and caching.

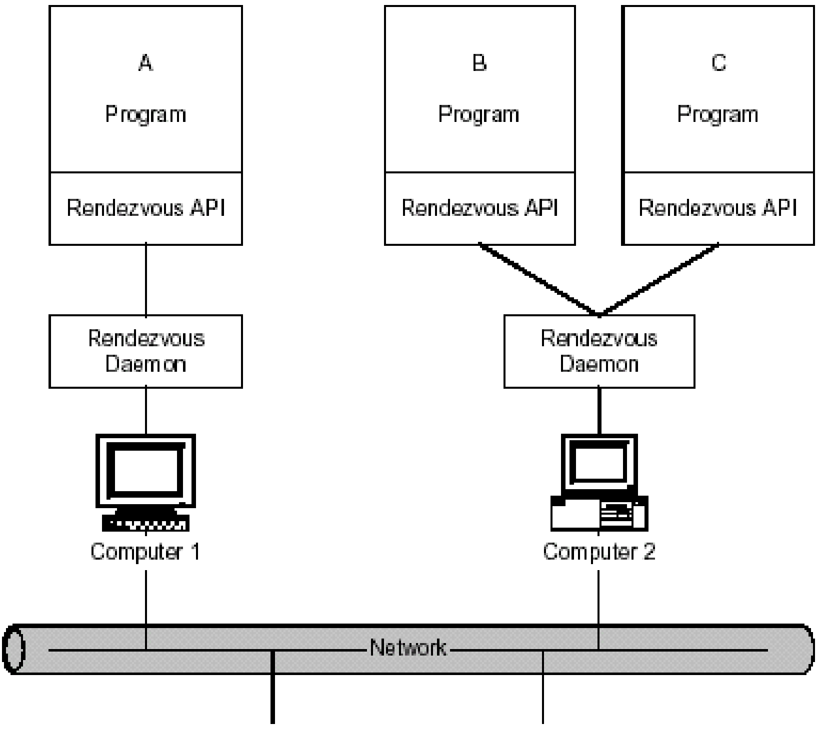


Figure 3.10 *The TIBCO Rendezvous operating environment (from [88])*

3.10.3 OpenJMS

OpenJMS [77] is an open source JMS compatible provider. Initiated in 1999, OpenJMS is part of the ExoLab initiative, an informal organisation working on the development of open source enterprise software projects. ExoLab focuses its efforts on Java and XML technologies and includes a number of leading projects such as OpenJMS, Open ORB[1], Castor, and Tyrex. In addition, it also contributes to external open source projects such as Tomcat, James, Xalan, and Xerces.

As of version 0.7.6.1, OpenJMS is conformant to the JMS 1.0.2 API specification and provides support for both point-to-point and publish/subscribe messaging models. When examined from an administrative perspective, OpenJMS can be configured using an administration GUI, an XML-based configuration file, or programmatically via a non-standardised administrative API. The administrative API allows the following limited manipulation of the provider's internal runtime state:

1. Destination Administration
   (a) Determine if a destination exists
   (b) List destinations
   (c) Create administered destination
   (d) Remove a destination

[1] This is not the same Open ORB project covered in Chapter 2.

(e) Count messages in a queue

(f) Count messages for a durable subscriber

2. User Administration

   (a) List all users

   (b) Add user

   (c) Remove user

   (d) Change user password

The API does not contain any application specific information and offers limited options for interoperability with other platforms. With the ability to retrieve a destination message count aside, the API does not expose any internal provider information such as current usage statistics. This limits any potential to reflect on current operation conditions and perform meaningful adaptations.

3.10.4 ActiveMQ

ActiveMQ [85] is a new open source JMS provider. Released under the Apache 2.0 license, it is a high performance messaging backbone with the ability to create a scalable cluster of message brokers. Version 2.0 fully supports JMS 1.1 and Java 2 Enterprise Edition (J2EE) 1.4, allowing it to integrate seamlessly into J2EE 1.4 compliant containers, light weight containers (such as Pico or Spring), or any Java application using its Java Connector Architecture (JCA) 1.5 resource adaptor.

ActiveMQ supports a wide range of different deployment topologies scaling in both the large and small. Each topology has specific strengths when deployed within different environments. The following topologies are available:

- *Client/Server* - The traditional client/server approach for distributed deployments is an efficient solution for servicing a large numbers of clients requiring a diverse range of messaging demands. With this topology, a number of communication protocols such as the Transmission Control Protocol (TCP), User Datagram Protocol (UDP), or Secure Sockets Layer (SSL) are employed to connect with a message broker. When deployed within a high-volume environment, brokers can be setup in logical clusters to provide high-availability using broker failover and load balancing.

- *Embedded* - Similar to the client-server topology, the embedded topology places an embedded broker locally within each client VM. With such a deployment, communication between the client and broker (server) take place within the same JVM, avoiding the extra hop required to go from producer to broker to consumer. Embedded brokers may also be used to provide store and forward isolation from remote brokers, allowing a remote broker to fail without affecting the publishing client e.g., the network could fail, but the client can continue to publish messages to its embedded broker.

- *Peer-to-Peer* - The peer-to-peer topology allows clients to communication in peer-based clusters, without the use of a central server. The peer-to-peer deployment can be setup in a number of ways. The most popular method is to utilise a multicast transport protocol for

communication; all nodes communicate using the same multicast address. As well as standard multicasting, other options for multicasting include using JGroups [89] or Suns Java Reliable Multicast Service (JRMS) [90]. While multicast deployments have great potential, they are not always a feasible choice for a production deployment. Pointcast based socket communication is a more mature and scalable solution within Java deployments.

Similar to most messaging services, ActiveMQ provides a configuration file to control a number of internal settings including transport connectors (consisting of transport channels and wire formats), network connectors (using network channels), discovery mechanisms, and persistence mechanisms.

While ActiveMQ does not have an administration API for runtime configuration, it does have a number of *on-the-fly* adaptive capabilities that cover some of the more common operations found within such APIs. ActiveMQ has an adaptive capability that creates destinations (queues or topics) on the fly when a message is send to an inexistent destination. While it is currently not possible to direct specific configuration settings of the destination, such functionality is planned for a future version.

3.10.4.1 J2EE Management API

ActiveMQ also provides partial support for the J2EE Management API [91], also know as the Java Specification Request 77 (JSR 77). This management API is a vendor-neutral API used to manage J2EE application servers. The management API enables the development of resource management tools for JSR-77-compliant J2EE servers; such resources include Java Database Connectivity (JDBC) connection pools and deployed applications.

One of the key features of this specification is the ability for an application server to describe its resources in a standard data model. With the use of the model, the API can probe the server to obtain information on its resources. This information can then be used to reflect on the operating conditions of the server and adjust its configuration accordingly. In order to convey performance related information, a substantial portion of the specification is dedicated to providing performance monitoring. Each resource is required to provide performance statistics; this information is grouped into Stats tailored for each managed object. These stats provide a snapshot of the current state of the resource. If a number of snapshots are taken over a time-period, it is possible to build a trend of the resource's usage.

Currently ActiveMQ does not support full J2EE management statistics, however their inclusion is planned in a future release. When in place these statistics will offer various stats concerning the connection, session, consumer, producer, and endpoint constructs within the ActiveMQ provider. This will enable the management stack (e.g. a JMX program) to use JMX based alerts (JMX notifications) to trigger actions when certain conditions exist within the provider. Even though there is currently no management API for ActiveMQ, its planned support for JSR 77 will increase the administrative capability of the service and simplify integration with JSR 77-compliant administration tools.

3.10.5 SonicMQ

SonicMQ [84] is a leading commercial JMS provider from Sonic Software with compliance for JMS 1.1 and J2EE 1.4, which provides integration support for J2EE application servers such as BEA

WebLogic and IBM WebSphere. SonicMQ is a high-performance messaging backbone capable of massive scalability in high-volume messaging environments [58, 92]. With its clustering capability, SonicMQ can share the messaging load among clusters of message brokers, allowing deployment to support large numbers of messages, users, and applications. A broker cluster can handle a greater number of connections, destinations, and persistent messages than an individual broker.

SonicMQ also has support for wide area deployments with the ability to establish groups of clusters, creating highly distributed deployments across loosely coupled locations. With the use of a technique know as Dynamic Routing Architecture (DRA) [93], SonicMQ can automatically senses the availability of communication paths and choose the best route to optimise network resource utilisation, ensuring an efficient traffic flow.

The administrative capabilities of SonicMQ are the most comprehensive of all the messaging services reviewed, providing full configuration control and performance metrics for the message broker at runtime. Broker administration is facilitated via two APIs:

- Configuration API – This API is used to configure all aspects of the broker including connections, containers, brokers, clusters, queues, and routings. It can be used to get and set individual attributes on a configuration, create and delete configurations, and manipulate tables and lists within the configuration.

- Runtime API – Provides monitoring and management facilities for running brokers and their containers. The API can be used to gather runtime information (i.e. performance metrics) on containers and components (i.e. agent, agent manager, activation daemon, and broker) within the broker using a JMX interface or the proprietary Sonic Proxy API.

The metrics exposed by the Runtime API can be selectively and dynamically enabled and disabled during broker execution. A certain amount of overhead is associated with statistical data collection and metric interpretation. Metrics are available for a number of aspects of the broker including:

- Memory usage
- Management thread pool
- Message traffic
- Connection activity
- Queue usage (per queue)
- Size of durable subscription store
- Connection reject rate

The administrative access in SonicMQ is based on a proprietary API that increases the complexity of interaction. However, it does provide a great degree of control over the broker, offering the potential for developing a number of reflective capabilities.

3.10.6 SIENA

The Scaleable Internet Event Notification Architectures (SIENA) [62, 67] is a popular example of a service utilising content-based filtering. SIENA is designed as a ubiquitous event notification service for use in wide-area networks and is suitable to support highly distributed applications. One of the main challenges faced by notification services in wide-area settings is maximising expressiveness in the selection mechanism without sacrificing scalability in the delivery mechanism.

The SIENA notification service extends the traditional publish/subscribe protocol with an additional interface function called advertise. A client uses this function to inform the service of the nature of notification that it might publish. Clients use the access points to advertise the information about events that they generate and to publish notifications containing that information. Clients also use access points to subscribe to notifications of interest. The service will then use the access points to deliver any relevant notifications to the client. The SIENA notification service is implemented as a network of servers that provide points of access to clients; the network of servers may be connected in a hierarchical or peer-to-peer topology.

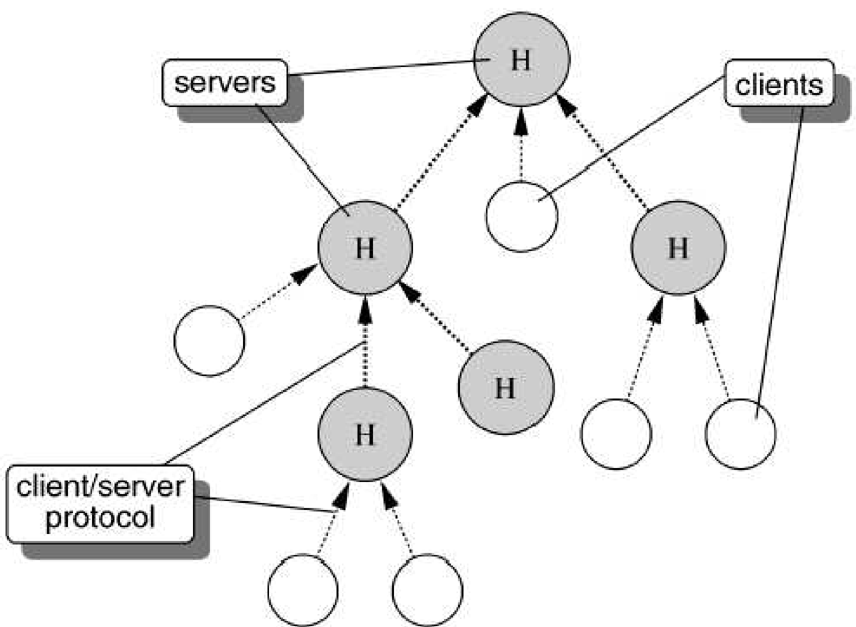


Figure 3.11 *The SIENA hierarchical client/server architecture (from [62])*

The hierarchical topology, illustrated in Figure 3.11, is a straightforward extension of a centralised architecture. Within this topology, the central server must be modified to propagate any information that it receives (i.e., subscriptions) on to its 'master' server within the hierarchy. The inter-server communication protocol used within the hierarchical architecture is the same protocol used between the client and server. Effectively, the master server sees the slave server as just another client. The slave receives subscriptions, advertisements, and sends and receives notifications like any other client. The main shortcoming of the hierarchical architecture is the possibility of overloading servers located at the top of the hierarchy. In addition, each server within the hierarchy acts as a critical point of failure for the whole network; failure in a server disconnects all the slave

servers and their clients from the rest of the network.

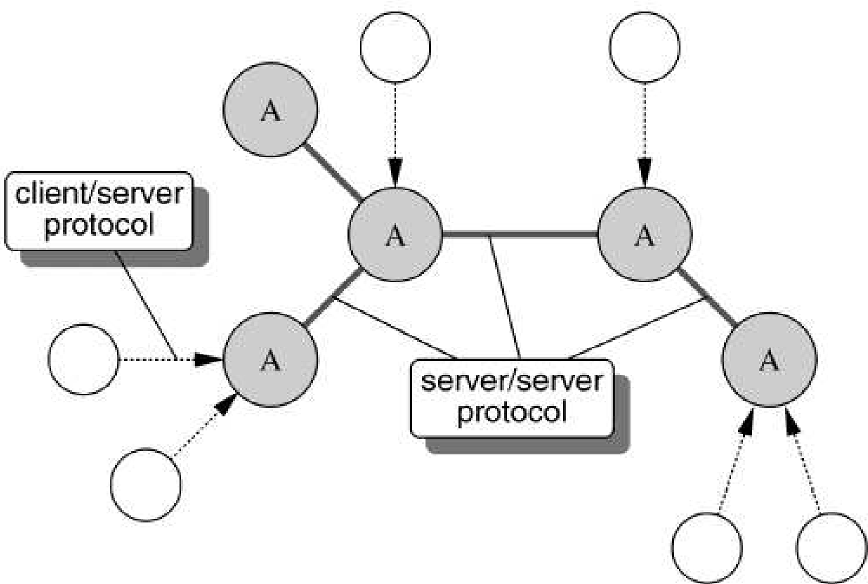


Figure 3.12 *Acyclic peer-to-peer server architecture in SIENA (from [62])*

SIENA also has a peer-to-peer architecture as illustrated in Figure 3.12. Within this topology, servers communicate with each other symmetrically as peers, utilising a protocol that allows a bidirectional flow of subscriptions, advertisements, and notifications between servers. SIENA conceptually supports both an acyclic and general peer-to-peer topology; however, only an acyclic topology has been implemented. To ensure the correct function of the routing algorithms it is important that server inter-connections maintain the property of acyclicity. However, administration of this can be difficult in a wide-area deployment. As with the hierarchical architecture, a lack of redundancy in the topology constitutes a limitation in assuring connectivity, a failure in one server isolates all the servers and clients reachable from the failed server.

SIENA does not provide an administrative interface to examine the internal routing information within a server, nor does it offer any API to alter these structures. This reduces the ability for reflective capabilities within SIENA. However, if such access were available a number of opportunities exist to develop reflective capabilities to improve redundancy within the SIENA architecture. One prospect is the development of an agent to identify servers with a history of common outages or failures. Once vulnerable servers are identified, the agent establishes redundant back-up routes around these servers to provide an alternative route when they fail. This reflective capability increases service availability and reduces the occurrence of subnet isolation within the overall notification service.

3.10.7 REBECA

The Event-Based Electronic Commerce Architecture (REBECA) [94, 95] aims to provide an event-based architecture for electronic business applications. Within such environments, highly customised multi-step flows require the engineering and administration of complex event-based appli-

cations and infrastructure. REBECA has two unique capabilities to service these environments; scopes to improve the interoperability of notification services and services to support location-mobility. In addition, REBECA also introduces the concepts of subscription merging [65, 69], described in Section 3.8.1, as a mechanism to minimise the size of routing tables.

The concept behind scopes [96] is to bundle the functionality that comprises an event-based system into a collection of sub-components. A scope is a collection of these sub-components that provides the rest of the world with common higher-level input and output interfaces to the event service. Scopes also act as an encapsulation mechanism by hiding the details of service implementation, such as the underlying data transmission mechanisms, the interface mappings between internal and external notification representations, security, and transmission policies. Scopes from the notification perspective provide an "encapsulation unit, a scope constrains the visibility of the notifications published by the grouped components" [96], effectively limiting the visibility of notifications within the system to a subset of message participants.

REBECA offers support for location mobility services [97] within event-based systems with the use of location-dependent subscriptions as a means to use the event-based paradigm within mobile scenarios. Mobile consumers using the location mobility service will only receive notification events related to their current location.

With no direct management interface, limited administrative capabilities are provided by Java Management Extensions (JMX) [98] events at key locations within the REBECA mobility service infrastructure [99]. The mobility service relies on data distributed among the brokers within the network; this information is exchanged between brokers using management events. Management events are first-class participants within the event service and may be subscribed to in a similar fashion to any other event within the service. These events allow a client, such as an administrative service, to receive interval management communications and 'update events' [99] from the REBECA mobility service and provide limited management capabilities.

3.10.8 Hermes

The Hermes distributed event-based middleware platform, developed at the university of Cambridge, is "a type- and attribute-based publish/subscribe model that places particular emphasis on programming language integration by supporting type-checking of event data and event type inheritance" [83]. A Hermes deployment consists of a network of peer-to-peer event brokers using an overlay network to create a distributed dissemination tree.

> The three layers of networks in Hermes are illustrated in Figure [3.13]. The bottom layer is the physical network with routers and links that Hermes is deployed in. The middle layer constitutes the peer-to-peer overlay network that offers a distributed hash table abstraction. The top layer consists of multiple event dissemination trees that are constructed by Hermes to realise the event-based middleware service. When a message is routed using the peer-to-peer overlay network, a callback to the upper layer is performed at every hop, that allows the event broker to process the message by altering it or its own state. [83]

Within Hermes, *Rendezvous Nodes* ensure consensus between brokers within dissemination tree for a particular event type. Any broker within a Hermes network can be a rendezvous node for an event type(s).

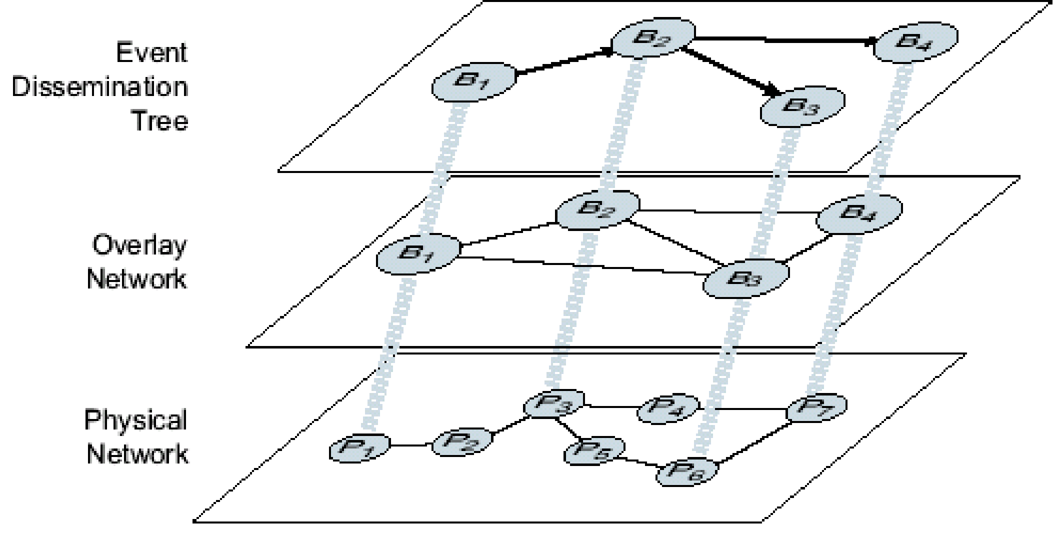


Figure 3.13 *Layered networks in Hermes (from [83])*

Hermes offers two forms of filtering, type-based, and type- and attribute-based filtering. Type-attribute based filtering enhances type-based filtering with content-based routing on the attributes of an event. Type-based filtering is similar in concept to topic-based (group) filtering within JMS. In addition to its filtering services, Hermes also offers a number of higher-level publish/subscribe services such as composite events, access control and a congestion control service.

Hermes use of a peer-to-peer overlay network has a number of benefits for event broker deployments. It also exhibits some of the self-management characteristics desirable within autonomic systems:

> The advantage of such peer-to-peer overlay network are threefold: First, the overlay network can react to failure by changing its topology and thus adding fault-tolerance to Hermes. Second, the peer-to-peer routing substrate that manages the overlay network is responsible for handling membership of event brokers in a Hermes deployment. Third, the discovery of rendezvous nodes, which must be well-known in the network, is simplified by the standard properties of the distributed hash table. [83]

The Hermes *Broker API* allows limited administration access for the interconnection of brokers within a network; the API also provides a list of neighbouring event brokers to which a broker is connected. An additional API is available to manipulate the distributed hash table, called PAN, within a Hermes broker. The *PAN API* enables administration of the overlay network, allowing nodes to join and leave the network, and provides capabilities to route messages over the overlay network.

3.10.9 WebSphere MQ (formerly MQSeries)

WebSphere is an IBM software family designed for business integration. The family contains a number of members including Business Integration Adapters, a Business Integration Message Broker, and Data Interchange services. The members of the family deal with various aspects of

the integration process such as centralising and applying business operations rules, or enabling the capture, visualisation, and automation of business processes. WebSphere MQ (formerly MQSeries [70]) is the family member that provides the communication mechanism between applications on different platforms.

WebSphere MQ enables integration by helping business applications to exchange information across different platforms; sending and receiving data as messages. WebSphere MQ provides a number of Application Messaging Interfaces (AMI) to interact with the message provider; interfaces are available for C, C++, COBOL, and Java using the JMS 1.1 API. WebSphere MQ can also be deployed on a number of operating systems including AIX, HP OpenVMS, HP-UX, iSeries, Linux for Intel and zSeries, Solaris, z/OS, and Microsoft Windows.

WebSphere MQ contains comprehensive administration capabilities, allowing the management of its configuration at run-rime. These include:

- Resource management - Queue creation and deletion
- Performance monitoring - Maximum queue depth or message rate
- Control - Tuning queue parameters such as maximum queue depth, maximum message length, and enabling and disabling queues
- Message routing - Definition of alternative routes through a network

Administration of a WebSphere MQ provider can be performed in a number of ways. The two more popular methods of administration are:

- Programmable Command Formats (PCF) - PCFs define command and reply messages that may be exchanged between a program and a compatible queue manager in a network. PCF commands are used for the administration of WebSphere MQ objects: queue managers, process definitions, queues, and channels. Each queue manager has an administration queue with a standard queue name to send PCF command messages to. PCF commands and reply messages are sent and received using the normal Message Queue Interface (MQI).
- The Message Queuing Administration Interface (MQAI) – MQAI is a programming interface to WebSphere MQ, using the C or Visual Basic languages. It performs administration tasks on a WebSphere MQ queue manager. The MQAI provides a more straightforward method of administration than PFCs.

Two further administration interfaces are available. The OS/400 Control Language (CL) may be used to issue administration commands to WebSphere MQ for iSeries. WebSphere MQ Commands (MQSC) provides a uniform method of issuing commands, expressed in WebSphere MQ Script (MQSC), across WebSphere MQ platforms. MQSC responses are designed to be human readable, whereas PCF command and response formats are intended for program use.

3.10.10 Other Reviewed Systems

As an addition to the projects examined within the main review, a number of other projects are briefly highlighted to place this research within the wider MOM research domain.

3.10.10.1 Gryphon (IBM Research prototype)

Initiated in 1997 at the IBM T. J. Watson Research Centre the Gryphon [74] project was designed to support the next generation of web applications, to go beyond the pull-based generation of applications and develop a robust publish/subscribe message broker.

Gryphon is implemented 100% in Java with optional native libraries for performance-critical sections; non-blocking I/O socket libraries for Windows NT, Linux, and AIX. Clients access the provider system through the JMS API.

Gryphon provides a content-based publish/subscribe service with a matching engine [100, 101] that offers high-speed filtering when compared to a topic-only approach. Gryphon also takes advantage of the benefits of channel hierarchies by organising its topics into hierarchies with support for wildcards to allow subscribers maximum flexibility in their topic subscriptions. Central to Gryphon's scalability and reliability is the possibility for deployment of Gryphon message brokers into a multi-broker network. Similar to concepts of clustering, a Gryphon multi-broker network can accommodate growth in load by simply adding additional broker machines. Gryphon provides a rich set of configuration options, allowing multi-broker deployments to exploit underlying network configurations and congestion control techniques [102] to improve scalability.

Gryphon was used as a core part of the Sydney Olympics Intranet (providing real-time monitoring and statistics data) and for real-time score delivery for the US Tennis Open, the Ryder Cup, and the Australian Open where it was used to push real-time information to over 50,000 concurrent clients. Gryphon now forms part of the IBM's WebSphere suite as the WebSphere MQ Event Broker, extending the suite with multi-broker, content-based publish/subscribe functionality.

3.10.10.2 Cambridge Event Architecture

The Cambridge Event Architecture (CEA) [103] extends the traditional synchronous request/reply interaction pattern to an asynchronous publish-register-notify interaction. CEA was designed to provide publish/subscribe communications for multimedia and sensor rich distributed applications. Message publishers within the CEA are known as an e*vent source*, while message consumers are known as an e*vent sink*. Within the CEA, it is possible for event sources and sinks to communicate directly with one another without the use of an intermediary. However, to reduce source/sink coupling *event mediators* can be introduced into the messaging deployment.

The CEA provides a strongly-typed event system where events are of a particular event class and are statically type-checked at compile time [83], due to this event type-checking, event sources and sinks are tightly coupled to one another. Content-based routing is performed at event sources to minimise communications overhead. This complicates the implementation of the event source as it is required to perform subscriptions evaluation for each of its event sinks; event mediators have no filtering capabilities, limiting the scalability of a deployment to that of its event sources.

3.10.10.3 ECO

The Event, Constrains, and Objects or ECO distributed event model, developed at Trinity Collage Dublin, is designed to support virtual world applications. ECO is "designed to be scalable by including filtering capabilities that were intended to decrease network traffic in a distributed implementation" [104]. ECO demonstrates "that filtering coupled with multicast communications can substantially decrease network traffic and thus enhance scalability" [104] within an event service.

ECO is designed to extend a “host” language with event-based communication concepts using the ECO model. ECO provides a simple three operation API that is used by message participants, or entities, to communicate. These operations are:

- *Subscribe(eventType, eventHanlder, constraint)* – Used by an entity to declare an interest in a particular type of event
- *Raise(event)* – Invoked to produce an event, ECO will deliver this event to all interested subscribers
- *Unsubscribe(event-type, event-handler)* – Used to remove a subscription

ECO does not include any administration API at the abstract model level, however such capability is not restricted within the models implementation.

3.10.10.4 JEDI

The Java Event-Based Distributed Infrastructure (JEDI) [75] is a distributed content-based publish/subscribe middleware. Within JEDI, *active objects* produce or consume messages. Messages are routing between active objects through *event dispatchers* organised as a tree structure; with support for the reconfiguration of the tree to cope with the failure of an event dispatcher. Subscriptions within JEDI are to a specific event or to an *event pattern*. “An event pattern is an ordered set of strings representing a very simple form of regular expression” [75].

JEDI has support for mobility and allows active objects to join and leave at different points within the network with the use of the *moveOut* and *moveIn* operations. “While the [Active Object] is disconnected, the event dispatcher stores the event patterns the [Active Object] is subscribed to, so that, when it reconnects, it does not have to resubscribe” [75].

3.10.10.5 Self-Organising Broker Topology

Researchers at the Technical University of Berlin are currently investigating the use of self-organising techniques within publish/subscribe broker topologies. Within such networks, the topology of the brokers is often assumed static. This work investigates the possibilities for “reconfiguration of the broker overlay network structure to increase pub/sub system efficiency. This includes adaptivity to changing usage patterns, issues in reconfiguration (like delay, message overhead, and message ordering) as well as the development of a metric to decide whether reconfiguration might be beneficial” [105].

While at an early stage of investigation, the approach to reconfiguration is based on the analysis of usage patterns. The self-organisation operation is based on the “actual message flow to decide if and how a reconfiguration of the overlay network can help to improve system performance” [106]. Reconfigurations are performed “while notifications are queued in the critical section to maintain message completeness and ordering properties” [106] ensuring against message loss due to self-organisation activity.

3.11 Comparison of Reviewed Systems

Each of the MOM systems reviewed are designed with different goals in mind. From the perspective of this research, the main objective of this review is to examine each system and consider its

messaging and administration capability. This analysis easily breaks down into two key questions:

- What are the common message capabilities of MOM platforms?
- What common administration access is available?

The remainder of this section presents the results of the survey.

3.11.1 Message Capabilities

Given the diverse objectives of the reviewed systems, it is no surprise that there is significant variety in their approach to messaging provision. However, considering one of the objectives of this research is to define a generic MOM meta-level, it is important to identify common messaging capabilities among the systems. To this end, four criteria have been selected to highlight commonalities:

- Supported Messaging Models - Does the implementation support point-to-point and/or publish/subscribe messaging model(s)?
- Filtering/Routing Capability – What filtering capabilities are provided?
- JMS Compliance – Does the implementation provide JMS support?
- Destination Construct – Is a destination-like construct used in the implementation?

The results of this MOM survey are available in Table 3.5.

MOM Name	**Supported Messaging Models**	**Filtering Type** (Routing Capability)	**JMS Compliant**	**Destination Construct**
CORBA Event Service	Publish/Subscribe	Channel-based	No	Yes
CORBA Notification Service	Publish/Subscribe	Content-based with patterns	No	Yes
OpenJMS	Both	Attribute-based Topic based	Yes	Yes
ActiveMQ	Both	Attribute-based Topic based	Yes	Yes
SonicMQ	Both	Attribute-based Topic based	Yes	Yes
SIENA	Publish/Subscribe	Content-based	No	No
TIBCO Rendezvous	Both	Topic-based	No	Yes
REBECCA	Publish/Subscribe	Content-based	No	No
Hermes	Publish/Subscribe	Composite events	No	No
WebSphere MQ (formerly IBM MQSeries)	Both	Attribute-based Topic based	Yes	Yes

Table 3.5 *Comparison of MOM messaging capabilities within reviewed systems*

Within the survey, half the systems reviewed supported both messaging models, with the remainder supporting only the publish/subscribe model. While it is possible to replicate a basic a point-to-point solution using a publish/subscribe only system, it can be difficult to recreate all of the characteristics of point-to-point; such as *once-and-once-only* messaging using multiple queue consumers.

Filtering capabilities of the systems within the survey range from simple channel-based mechanisms present within the CORBA Event Service, to content-based pattern filtering and composite events within SIENA and Hermes respectively. The majority of systems supported a form of content/attribute-based filtering, however the capabilities of the mechanisms used varied. JMS compliant systems provide support for attribute-based filtering via message selectors and the possibility of topic-based filtering via hierarchical topic structures[1].

Within the survey, of the ten systems examined, only two are system specifications: the CORBA Event and Notifications services. The remaining eight are a mix of three commercial, three academic and two open source systems. Of these eight systems, four are JMS compliant (2 open source and 2 commercial). Compliance to the JMS specification was non-existent among academic systems, this is most likely a result of the target audience of the work; research prototypes are unlikely to be utilised within production environments where portability, vendor neutrality and legacy integration concerns are paramount.

The destination construct was present in seven of the ten systems reviewed. The three systems without such an entity were academic research prototypes supporting content-based publish /subscribe, these system also lacked support for the point-to-point messaging model and JMS standard.

3.11.2 Administration Capabilities

The second part of this survey examines the administration capabilities within the reviewed systems. The objective of this survey is to reveal any administration facilities and to classify their capability. To this end, the following four criteria are used to examine each system:

- Administration API – Does the implementation provide an API to access its administration capabilities?
- Extensibility – Can the Admin API be extended to perform additional operations?
- Runtime Reconfiguration – Can the MOM be reconfigured at runtime?
- Monitoring Capability – Does the MOM provide any monitoring capabilities to reveal its current operating condition?

The results of this administration survey are presented in Table 3.6.

All but one of the systems reviewed, SIENA, has some form of administration API. The only system capable of extending its administration capability is WebSphere MQ using its Programmable Command Formats, three other systems have a limited possibility of extension through the Java Management eXtensions API [98] which contains a built in ability to define generic management actions. Administration capabilities varied with eight systems providing some form of

[1] The JMS specification does not specify the implementation of this functionality. However, the majority of implementations contain a provider-specific topic grouping mechanism.

MOM Name	Administration API	Extensibility	Runtime Reconfiguration	Monitoring Capability
CORBA Event Service	Very Basic	No	Limited to associating suppliers and consumers with a channel	No
CORBA Notification Service	Yes	No	Yes	Yes (Channel, Supplier and Consumer lists)
OpenJMS	Yes	No	Yes	Limited to Queue message count
ActiveMQ	Planned	Limited, via JMX	Yes (partial)	Planned
SonicMQ	Yes	Limited, via JMX	Yes	Yes
SIENA	No	No	No	No
TIBCO Rendezvous	Yes	No	Yes	Yes
REBECCA	Yes	Limited, via JMX	No	Restricted to update events
Hermes	Yes	No	Broker and Overlay network node administration	Neighbouring Broker List
WebSphere MQ (formerly IBM MQSeries)	Yes	Yes (via PCF)	Yes	Yes

Table 3.6 *Comparison of MOM administration capabilities within reviewed systems*

runtime-reconfiguration ability ranging from simple supplier/consumer channel associations with the CORBA Event Service to fully featured APIs within SonicMQ and WebSphere MQ. A similar variation in capacity is also observed for monitoring capacity within the systems.

Overall, the survey uncovered that while most MOM implementations posses some form of proprietary administration API, none possessed a first class generic meta-level capable of runtime monitoring and inspection of a MOM base-level. Even within the systems possessing a comprehensive administration API, no standard representation of an internal MOM structure and state was evident between the systems, limiting the possibility of coordinated administration interactions.

3.12 Summary

Message-Oriented Middleware (MOM) provides an alternative to the Remote Procedure Call distribution mechanism. With its clean method of communication between disparate software entities, MOM is one of the cornerstone foundations that distributed enterprise systems are built upon. MOM can be defined as any middleware infrastructure that provides messaging capabilities and a multitude of implementations exist to target diverse messaging requirements such as mobility and enterprise integration to scaling in both the large and small.

While MOM provides a powerful mechanism to integrate multiple systems, its setup and maintenance is unnecessarily labour intensive. When examined from the perspective of administration, no standardised management interface is available for MOM, limiting the scope of cross-platform administration tools and coordination. Limited reflective self-management techniques are utilised within current MOM implementations, presenting an opportunity to investigate their use within this domain to develop self-management capabilities to ease the maintenance burden, and improve service performance.

Part II

Contribution

Chapter 4

Meta-level Coordination

The investigation of meta-level coordination is the primary research objective of this work. This chapter details the process used to define an interaction protocol to facilitate coordination between meta-levels.

4.1 Introduction

Self-managed middleware systems have been developed to cope with the demands of current and next-generation computing environments. Such systems are capable of adapting or self-adapting to meet changing user or environmental requirements. A number of reflective middleware platforms have been developed to provide such capabilities including Open ORB [7], DynamicTAO [8], RAFDA [52], QuO [39], and mChaRM [32].

Current research has focused on the reflective capabilities of a system to examine its own internal operation, known as internal reflection. In reality, the vast majority of software solutions comprise of a number of interacting proprietary implementations. In such environments, internal reflection is not sufficient; systems need to examine both themselves and their interactions with other systems.

Standards and protocols such as General Inter-ORB Protocol (GIOP), Java Remote Method Invocation over IIOP (RMI-IIOP), and the Web Service stack (HTTP, SOAP, UDDI, and WSDL) have provided a common infrastructure to enable proprietary software implementations to interact. This regulated infrastructure facilitates the exchange of information between implementations. As self-managed platforms are deployed in the field, they will inevitably encounter and interact with other self-managed capable systems. These infrastructure standards enable the base-level of a self-managed platform to interact. However, no standard or protocol is available to allow the meta-levels of the platforms to interact.

Interoperability within the meta-level is one of the key challenges for future self-managed platforms. The emergence of an open standard for meta-level interaction is imperative to support the development of next-generation, self-managed middleware that coordinate and cooperate with systems with which they interact. Such standards need to define adaptive and reflective capabilities in a neutral format, increasing interoperability across proprietary implementations.

This chapter provides motivation for the investigation of meta-level coordination activities with a vision of coordinated self-managed environments. The requirement for an interaction protocol

for such environments is then highlighted. The chapter concludes with the definition of the Open Meta-level Interaction Protocol (OMIP) that meets these requirements.

4.2 Motivational Scenario

As motivation for the research hypothesis, a hypothetical scenario of an online multimedia broadcasting service is presented. This service broadcasts audio streams using an open standard, an MP3 stream, as its distribution format. Utilisation of this open format enables multiple proprietary media players to seamlessly connect to the service. In this environment, as illustrated in Figure 4.1, any media client that supports the MP3 format is capable of receiving the multimedia (MP3 stream) broadcasts. With this configuration, clients connect to a MP3 stream of fixed quality.

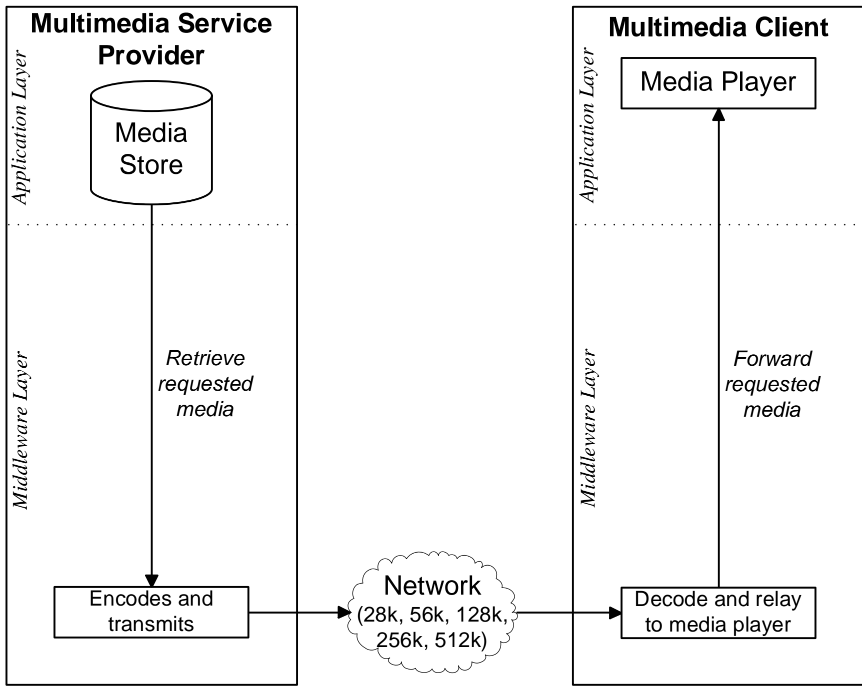


Figure 4.1 *A non-reflective media broadcast service*

With the use of current reflective research, such as Open ORB, it is possible to build such a service with the ability to self-adapt to a client's capability including bandwidth, latency, or connection reliability. Current reflective techniques improve the Quality of service (QoS) provided by the multimedia broadcasting service by altering the process in which it serves media to its clients. For example, attempting to send a high-quality live audio stream to a client on a low-bandwidth connection will result in a poor QoS for that client. Ideally, the service should recognise the current network capability and transmit a more suitable, lower-quality stream to the client. The choice of an appropriate media encoding and service infrastructure can achieve an increase in the QoS received by the client.

As illustrated in Figure 4.2, current reflective research is proficient at implementing a service with self-adaptive capability. In this reflective broadcast service, it is important to note that the

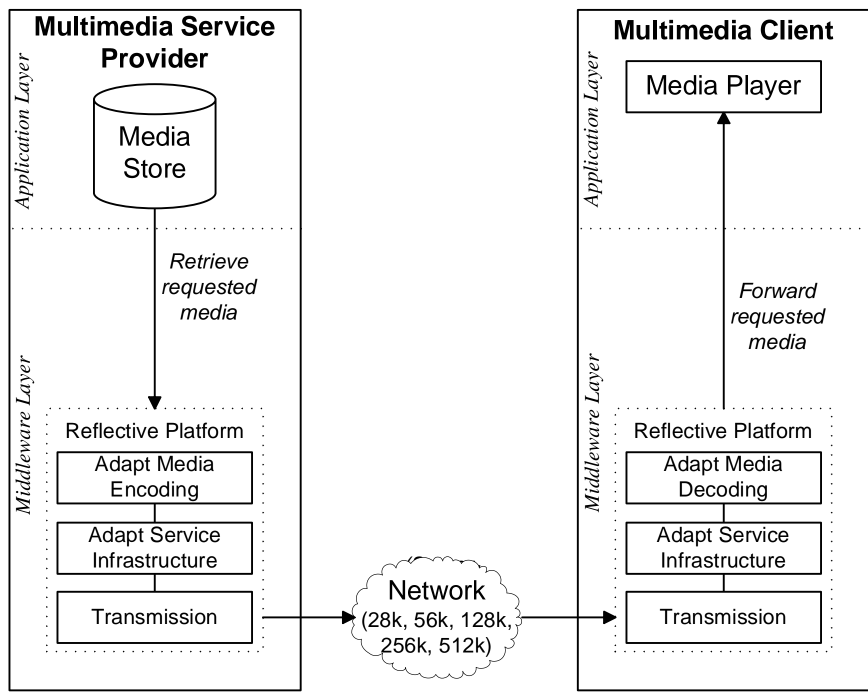


Figure 4.2 *A proprietary reflective broadcast service*

same reflective implementation (i.e. Open ORB) is present in both the server- and client-side middleware stacks. When compared to the stack of the first solution, the reflective capabilities have greatly improved the QoS provided by the service. However, utilisation of these capabilities removes the independence provided by the open MP3 format, resulting in *lock-in* to a particular proprietary reflective implementation[1].

Ideally, the use of adaptive capabilities should not result in lock-in to a proprietary implementation. This could be achieved with the use of an interaction mechanism for the meta-level of adaptive and reflective platforms to coordinate the operation of these capabilities between diverse implementations.

In Figure 4.3, a coordinated reflective broadcast service is presented. The middleware stack introduces a new meta-level interaction protocol. This protocol allows for communications between the server- and client-side meta-levels to coordinate their reflective activities. When a new client joins, it is able to discover what "capabilities" or "formats" the service can distribute its multimedia content in, i.e. what adaptations are available/provided by the service? The client is now able to request the service to deliver the media in a format that best suits the client's current operating conditions. It may also request additional adaptations if its infrastructure was to change (improve or worsen).

When compared to the previous proprietary reflective approach, the service adapts to improve its QoS in the same manner. However, the additional meta-level interaction protocol alleviates the problem of proprietary lock-in. Any client that supports this protocol can benefit from the service's reflective capabilities and request adaptations while maintaining implementation independence.

[1] This problem is not unique to Open ORB and would be experienced if any propriety technology were to be used.

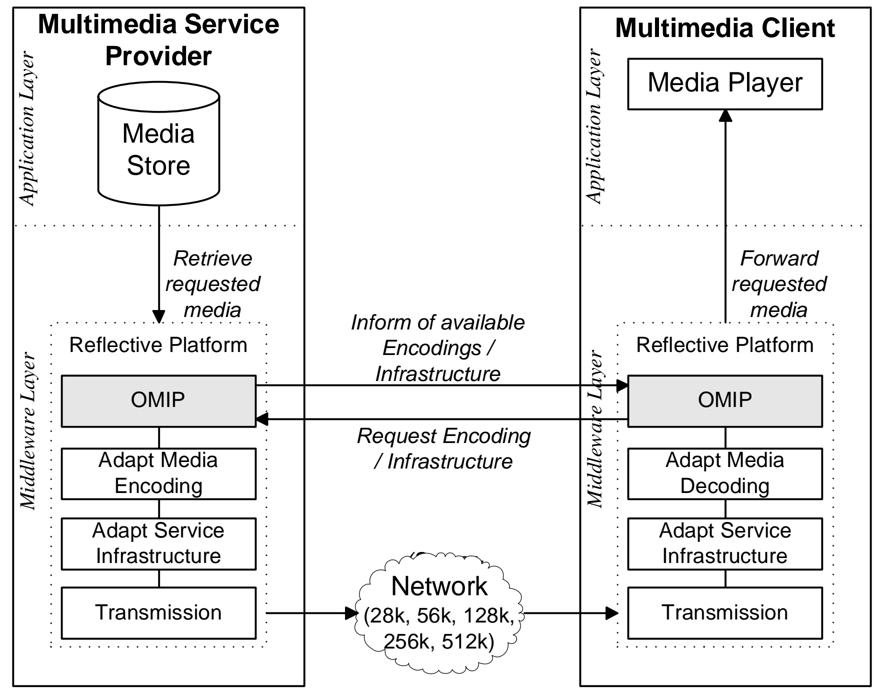


Figure 4.3 *A coordinated reflective broadcast service*

4.3 Opening the Meta-Level

Critical to the development of interoperable heterogeneous systems is the definition of standards that define the interface and interactions among interacting systems. To provide some background to this problem and shed some light on possible solutions, this section briefly examines standardisation efforts within other domains.

The networking community developed standards such as the Management Information Base (MIB) and Simple Network Management Protocol (SNMP) for network management. These standards have made network devices easier to control through a common administrative protocol. Similar management standards have also reached the software management domain with efforts such as Java Management Extensions (JMX). JMX enables the integration of Java application into existing network management solutions, simplifying the management of software applications. The vision of Grid computing has been made possible thanks to the development of the Open Grid Services Architecture (OGSA) which defines the standard interfaces and behaviours of a Grid service by building on a web services base. Each of these efforts illustrate the benefits of standardised interaction protocols at the management layer. If self-managed systems are to take their place within future computing environments, they will require their own management interaction protocol.

Researchers within the Agent community encountered similar issues at an early stage of investigation into open multi-agent based systems. After a number of years the Foundation for Intelligent Physical Agents (FIPA) [107] emerged as a standards body for the agent community. FIPA is an international organisation dedicated to promoting the industry of intelligent agents by openly developing specifications to support interoperability amongst agents and agent-based systems. FIPA

defines a number of standards and specifications that include architectures to support inter-agent communication [108], interaction protocols [109] between agents, as well as communication and content languages [110] to express the meaning of these interactions. With the use of these standards, any FIPA-compliant platforms and their agents are able to seamlessly interact with any other FIPA-complaint platform. FIPA serves as an example for a similar effort with self-managed reflective middleware.

4.4 A Vision of Coordination and Cooperation

The merits of coordination and cooperation have been highlighted but before the enabling protocol is discussed, it is beneficial to consider potential characteristics of open self-managed environments to offer guidance for the definition of the protocol. To this end, a vision of open self-managed systems is examined to investigate potential participant interactions, participant relationships, and the changing role of reflection. All of these factors are considered to define the prerequisites needed by the protocol to enable open self-management interaction.

One of the most interesting aspects of open self-managed systems is their relationship to one another and the rules that govern the relationship. Within an open environment, participants are able to reflect on their current requirements and request actions from others, such as requesting information or an adaptation. The dynamics of inter-participant relationships may be set up in a variety of manners allowing the definition of communities in a number of fashions, including:

- Cooperatives – All participants work together to achieve mutual goals
- Market Places – Resource trading between participants
- Master/Slave – Slaves relinquish control to a master
- Commons – Deregulated control of common resources between participants

Participant relationships may also extend beyond simple adaptation requests to the sharing of information between participants. When a new participant arrives into an environment, it may request an individual or group of participants to inform it of their history within the environment. This sharing enables the participant to examine past requirements or patterns within the environments, enabling it to rapidly gain experience of its new environment with the help of its community. Such arrangements could also exchange real-time environmental information. This can reduce the overhead and burden of reflection by normalising and sharing common monitoring activities between participants. The concept of information sharing can be extended to a common independent information service to assist in the collection, aggregation, maintenance, and dispersion of heterogeneous information sources within self-managed middleware, minimising the cost associated with these tasks. The Collective [111] is one such service which assimilates common information collection tasks within self-managed systems, reducing the effort required to implement self-managed systems by removing the need to duplicate common information infrastructure.

When operating within an open environment the scope of reflective computation must also change from an introverted process to a more extroverted one. Reflective computation will need to be more liberal as the bounds of the reflection process are extended beyond the concern of the systems own base-level to include the affects of cooperation and coordination with other self-managed systems. Coordinated environments provide interesting possibilities for the formation of

autonomic system groups. When examined from a group/community perspective, the reflective process is opened to the investigation of group dynamics and game theory, where participants desire to coordinate but may not have mutual goals.

4.5 Protocol Prerequisites

When designing a protocol to facilitate open interaction it is important to consider that reflective self-managed systems have been developed for a diverse range of deployment environments. Given the diversity of these environments, and the impossibility of evaluating each one, a pragmatic approach to the development of an interaction protocol is needed to facilitate as much openness as possible, while minimising the complexity of the mechanism. With this approach in mind, the basic requirement of the protocol is to:

> Facilitate access to self-management services (state, adaptive capability, and analysis capacity) in a generic manner

Facilitating generic access to self-management services is vital to unleash the maximum potential of coordination and cooperation within an environment. Given the diversity of potential environments, it is important that an interaction protocol does not define inter-participant relationships as this could limit the usability of the protocol within certain environments. With the core requirement of the protocol defined, the next task is to specify desirable qualities of the interaction protocol. Given the heterogeneous nature of current and future computing environments it is vital that the interaction protocol possess the following characteristics:

- Open Accessibility - Previously unfamiliar systems must be able to interact using the protocol.
- Extensible Interaction – The protocol is extendable to provide previously undefined interaction requirements, facilitating interaction within newly defined application domains.
- Implementation Agnostic – The protocol must work with different implementations and technologies.
- Independent Control – The protocol must not dictate a relationship between interacting participants. Participants maintain independent control and can refuse requests and interactions. This does not prevent relationship definition at the participant level.

The open and generic nature of these characteristics ensure the protocol will exploit the potential of open interaction between self-managed systems to maximise the likelihood of meeting unpredictable requirements within the diverse range of deployment scenarios they encounter.

4.6 Open Meta-level Interaction Protocol (OMIP)

Facilitating interaction within diverse operating environments and application domains requires interactions to be based on a model of semantically grounded communication that is pre-defined, semantically rich and well understood by all parties. To this end, the Open Meta-level Interaction Protocol (OMIP) is introduced as the basic building block of communication between self-managed

participants. The remainder of this section details the design of the OMIP, including a walkthrough of an OMIP interaction.

The role of the OMIP is to standardise conversations between participants, creating the rules under which dialogue takes place. The basic requirements of this protocol are:

- An ability to request available capabilities (available actions and information)
- An ability to exchange information
- An ability to request an action (adaptation)[1]

The OMIP captures a common understanding of the basic elements that comprise a conversation between two participants. The two fundamentals steps needed to define the OMIP are; firstly, detail a minimal set of interactions needed to perform the basic requirements of OIMP. Secondly, it must capture a common understanding of the basic elements that comprise a conversation between two participants.

To achieve this, the OMIP defines a number of generic interaction commands based on the *Request-Reply* paradigm. Each interacting command contains an associated message. The content of this message is expressed in a domain specific language. These languages are responsible for describing the application/domain-specific details of the request (i.e. security, hardware, multimedia, telecoms, flight control, education, or UI); these languages may also contain their own specific interaction commands. Interactions between systems are achieved with a combination of the relevant interaction command with an associated message expressed in a domain specific language.

4.6.1 Interaction Commands

The goal of *Interaction Commands* (IC) is to describe entire conversations between participants to achieve some action (i.e. an adaptation) or outcome (i.e. resource allocation). ICs provide the context in which to interpret the associated messages. Figure 4.4 illustrates the core ICs.

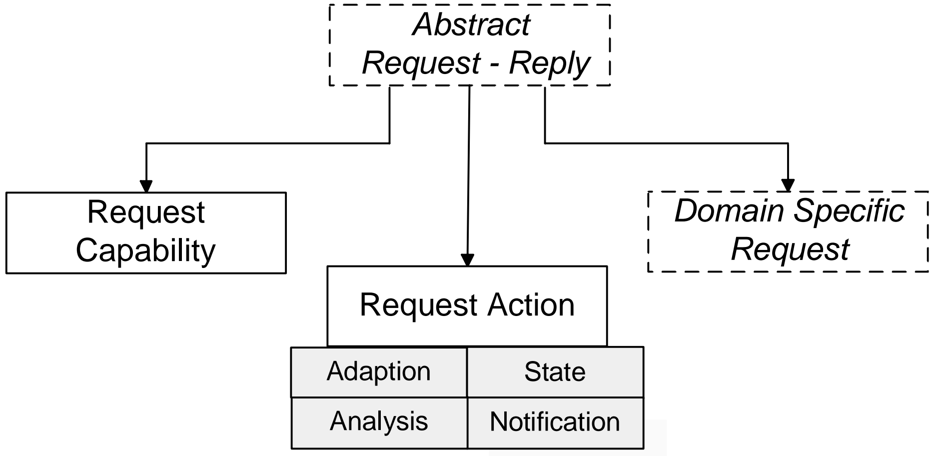


Figure 4.4 *OMIP interaction command hierarchy*

[1] An important point on this requirement is to note that this is merely an ability to request an action, the decision on whether the change takes place or not is up to the target participant.

The OMIP utilises this collection of request-reply interactions to construct conversations between participants. Each of these interactions makes a request of another participant; the target participant returns a related reply to the initiating participant. The core ICs are described in Table 4.1.

Command	Description	Possible Reply
Request Capabilities	Retrieve a participant's self-management capabilities and the domain specific languages it uses to describe them.	Available capabilities / Refuse
Request Action	Request a participant to perform one of its supported actions (Adaptation | Analysis | Notification | Return State).	Accept (optional content) / Refuse

Table 4.1 *OMIP interaction commands*

These core ICs are implemented within the domain specific language for the application domain. For a DSL to be OMIP-compatible it must support the core ICs. This is an important point in the design of the OMIP. Allowing the DSL to define interaction commands empowers it with the flexibility to implement the command in a manner that best suits the application domain it represents. No restriction exists on the definition of the command nor is they're a restriction on the medium used to express the commands. The main role of the IC is to capture the semantic meaning of these commands to provide the basic building blocks of open interaction.

Further non-core ICs may be easily defined for interactions including auctioning, issuing a Call for Proposals or Participation (CFP), negotiation, brokerage services, and subscription-based services.

4.6.2 OMIP Walkthrough

With the use of these request/reply commands, it is possible to perform a full-scale interaction with an external meta-level. As an demonstration of the interaction mechanism proposed, a systematic walkthrough of a conversation between two participants is described. Figure 4.5 highlights all the stages of an OMIP interaction, from acquiring the available capabilities of a self-managed system, to requesting the system to perform an action:

1. *Discovery* – there is no restriction on participant discovery within OMIP. Participants discovery make take place via a directory service, user prompting, system configuration, or from another participant.

2. *Request Capabilities* – post discovery, the first act is to exchange capability information using the *Request Capabilities* command. This command will return a list of available capabilities including possible adaptations, analysis, event, and state information.

3. *Request Action - State* – The *Request Action* command allows a participant to request any of the supported capabilities. The first use of this command in the walkthrough requests the self-managed system to return state information, enabling the imitator participant to understand the systems internal management state. The target participant may agree to the request and reply with the relevant information or refuse the request.

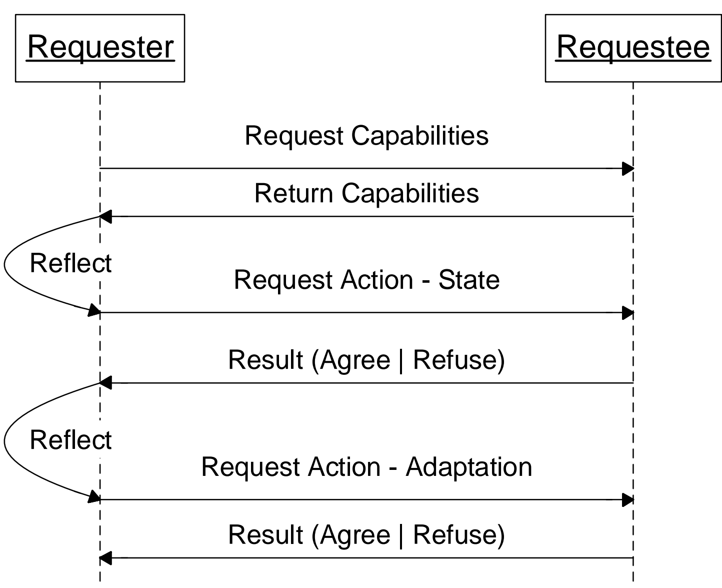


Figure 4.5 *OMIP participant interaction sequence*

4. *Request Action - Adaptation* – The second request action command in the sequence asks the system to perform an adaptation supported within its capabilities advertisement. The target participant may agree to the request and perform the adaptation request or refuse the request.

4.6.3 Domain Specific Languages

With the ability to choreograph interactions between participants in place, the next step is to define the content associated with these interactions. In order for two independently-developed platforms to interact with one another, a shared common understanding of content must exist to enable interoperability. To achieve this communication, OMIP uses Domain Specific Languages (DSL) to describe content related to self-management operations within diverse deployment environments. The division of languages allows the definition of highly specialised languages to describe specific conditions within a domain. This allows OMIP to be very comprehensive in its descriptions of domains without resulting in a bloated single language; such an approach would result in a high cost of entry by requiring conformance to a large specification.

The use of multiple DSLs reduces the cost of entry to only the relevant DSLs that describe the domain and its capabilities. DSLs can be defined for a number of areas such as network connectivity, security, transactions, and notification services. The definition and format of a DSL is a task for the interest groups of the respective domain to decide. The only requirement for a DSL to be OMIP compatible is for it to support the core interaction commands described in Section 4.6.1. DSL designers have the freedom to define these commands in the manner that best suits their application domain, without restriction. To facilitate further discussion two sample DSLs are provided for the multimedia and security domains.

4.6.3.1 Multimedia-DSL

In order to provide interaction within the motivational scenario introduced in Section 4.2, a DSL is required to describe possible multimedia capabilities within the multimedia domain. A DSL for multimedia platforms such as the broadcast service is provided in Table 4.2. Given the range of possible service offerings within this domain, only a subset of capabilities is included within the DSL to maintain clarity.

Name	**Purpose**	**Service Options**
Media Type	Type of media available	Audio Only, Video Only, Audio and Video
Encoding Format	Formats in which media can be encoded	aac, avi, mp3, mpg, ram, ogg, wav, wma
Bit Rate	Quality of the encoding	28kbps, 56kbps, 128kbps, 256kbps, 512kbps
Delivery Mechanism	Mechanism of deliver	Streamed, Download

Table 4.2 *Sample multimedia domain specific language*

With the use of this Multimedia-DSL it is possible for a platform to describe the adaptations it can perform.

Name	**Available Service Options**
Media Type	Audio Only
Encoding Format	aac, mp3, ogg, wav, wma
Bit Rate	28kbps, 56kbps, 128kbps, 256kbps, 512kbps
Delivery Mechanism	Streamed, Download

Table 4.3 *Sample multimedia service definition in Multimedia-DSL*

A sample service description is provided in Table 4.3. This service definition details possible adaptations available from the service. With this information, a participant can request changes to the format/compression and delivery mechanism of a particular audio source provided by the service. With the definition provided in Table 4.3, audio can be requested as a 128kbps MP3 encoded stream.

4.6.3.2 Security-DSL

The next example presented is a DSL to describe the security domain for user authentication and encryption protocols. Again, for the sake of clarity, only a minimal DSL is described and a public key infrastructure is assumed to be in place. Table 4.4 details the Security-DSL.

The Security-DSL allows a platform to describe the security protocols it supports, Table 4.5 provides a sample service description.

Using this service description it is possible to request authentication to be achieved using EAP with a SHA-1 hash, and for information to be encrypted using AES and sent using the SSH

Name	Purpose	Service Options
Authentication Protocol	Authentication options	RADIUS, S/Key, TACACS, CHAP, Kerberos, IPSec (AH), EAP
Hashing Algorithm	Optional hashing algorithm used with authentication protocol	MD5, SHA-1
Encryption Protocol	Protocol for encryption	CIPE, SSL, SHTTP, SSH, SSH2, IPSec
Encryption Cipher	Cipher used with encryption protocol	Triple DES, RC4, RC5, AES, IDEA

Table 4.4 *Sample security domain specific language*

Name	Available Service Options
Authentication Protocol	Kerberos / IPSec (AH) / EAP
Hashing Algorithm	MD5, SHA-1
Encryption Protocol	SSL / SHTTP / SSH
Encryption Cipher	Triple DES / RC4 / RC5 / AES

Table 4.5 *Sample security service definition in Security-DSL*

protocol. DSL's can be used in conjunction with one another to describe different aspects of a service, for example the Security-DSL could be used with the Multimedia-DSL to describe a secure multimedia broadcast service. With the semantic of the DSL agreed, the next step is to define the message format.

4.6.4 OMIP Message Definition Format

The definition of a message format for OIMP must be semantically well-grounded and universally understood by all participants. The responsibility for message definition rests with the designers of the DSLs, providing flexibility for the authors of these languages to choose the format which best suits their domain. To demonstrate the process of creating a message format the remainder of this section defines one for the Multimedia-DSL. When defining a message format for a DSL two key factors must be considered; the medium in which messages are expressed and the definition of the DSL representation within this medium.

4.6.4.1 Message Medium

The choice of medium for message expression is an important one. The medium must be suitable to convey the interaction commands and express related information. The multimedia domain is a diverse one, with many media formats and technologies available across heterogeneous platforms. In order to enable an open exchange of information between such systems, the representation format becomes key to a successful transference.

Within the Enterprise Application Integration (EAI) and Business-to-Business (B2B) domains, XML has proved very successful as a means to bridge heterogonous systems. XML is a popular

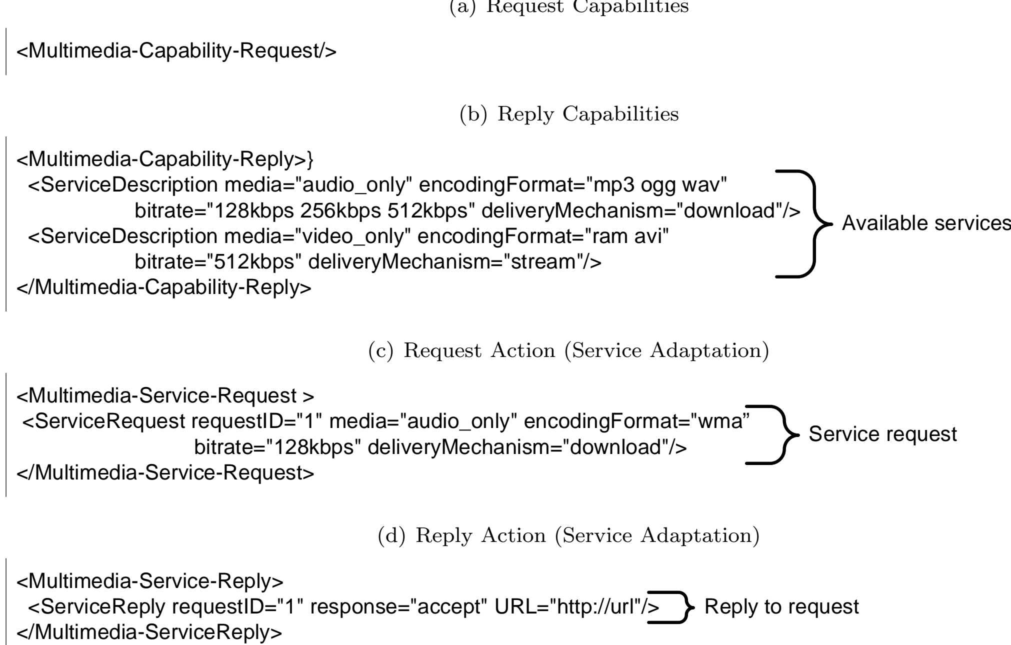
medium for the exchange of information within heterogeneous environments. XML would also provide an ideal format to bridge diversity within the multimedia domain by providing a standardised medium for the expression of infrastructure. With the medium for message chosen the next step is to define how the DSL will be expressed within the XML medium.

4.6.4.2 Multimedia-DSL Definition

The definition of the Multimedia-DSL in XML is a straightforward process of expressing the DSL in an XML document. Two key steps map the DSL in XML:

- Define interaction commands
- Define associated content for commands

Interaction commands are available in two forms, standalone commands and commands with nested content related to the command. Standalone commands are simple to define as they need only convey the requested actions and contain no additional content. An example of a standalone command is the request capabilities command. The second type of command contains related information such as a request for a particular multimedia service. The body of these commands must convey the details of the request. A sample of both command types for the Multimedia-DSL is provided in Table 4.6. A full XML schema definition of the Multimedia-DSL is provided in Appendix B.



(a) Request Capabilities

```
<Multimedia-Capability-Request/>
```

(b) Reply Capabilities

```
<Multimedia-Capability-Reply>}
  <ServiceDescription media="audio_only" encodingFormat="mp3 ogg wav"
              bitrate="128kbps 256kbps 512kbps" deliveryMechanism="download"/>
  <ServiceDescription media="video_only" encodingFormat="ram avi"
              bitrate="512kbps" deliveryMechanism="stream"/>
</Multimedia-Capability-Reply>
```


(c) Request Action (Service Adaptation)

```
<Multimedia-Service-Request >
 <ServiceRequest requestID="1" media="audio_only" encodingFormat="wma"
                    bitrate="128kbps" deliveryMechanism="download"/>
</Multimedia-Service-Request>
```


(d) Reply Action (Service Adaptation)

```
<Multimedia-Service-Reply>
  <ServiceReply requestID="1" response="accept" URL="http://url"/>
</Multimedia-ServiceReply>
```


Table 4.6 *Multimedia DSL sample interaction command definitions*

With the use of this XML-based definition of the Multimedia-DSL, two independently developed multimedia systems are able to interact with one another requesting information and actions from the other.

4.6.5 ARMAdA – A Sample Participant Architecture

With the OMIP mechanism in place, open interactions are possible between self-managed systems where each participating entity (platform, system, service, etc.) is part of a community (group). Utilising this capability will require a rethink in the design of self-managed systems. The OMIP does not attempt to describe how to implement a self-managed participant, nor does it attempt to specify the internal architecture of a participant. To initiate discussion on the design and implementation of OMIP provision within a self-managed system a sample OMIP enabled architecture is presented.

The Adaptive and Reflective Middleware Abstract Architecture (ARMAdA) is a conceptual view of how self-managed systems can interact with one another. ARMAdA is a suggestion of how such a system may be constructed. The sample architecture, illustrated in Figure 4.6, builds on a traditional reflective implementation of a base-level controlled by a meta-level.

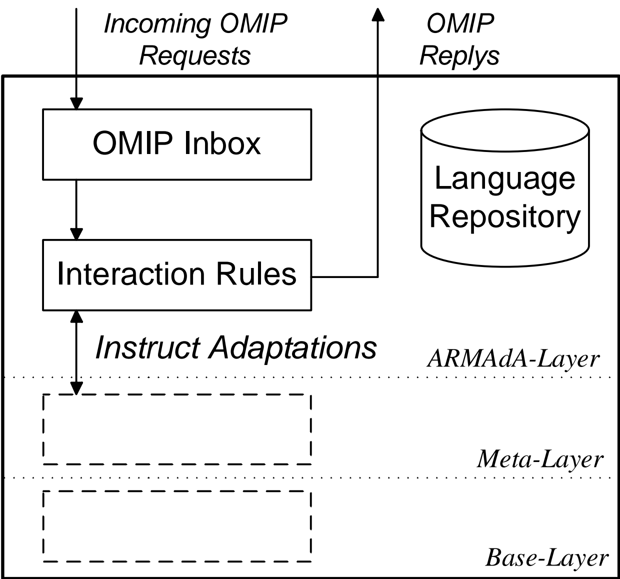


Figure 4.6 *Sample ARMAdA compliant architecture*

The novel element of this sample architecture is an additional layer on top of the meta-level. This new layer handles OMIP interactions for the participant, acting as a gateway to the platforms meta-level. The ARMAdA-layer consists of an Inbox, a Language repository, and a description of the capabilities offered by this platform. With these three elements, the system is able to perform all the basic interaction needed to participate within a self-managed environment. The ARMAdA architecture illustrates an approach to implement a new OMIP platform as well as a mechanism to retrofit OMIP capabilities on a legacy system.

4.7 Summary

Interoperability between adaptive and reflective platforms is an important next step in the development of self-managed systems. This chapter presents the vision of coordinated self-managed systems and introduces the Open Meta-level Interaction Protocol (OMIP) that uses a minimal set

of interaction commands to enable interaction between self-managed systems.

The OMIP defines self-managed capabilities with the use of Interaction Commands (IC) and Domain Specific Languages (DSL). These languages allow the creation of highly specialised descriptions for each domain, allowing the OMIP to be very comprehensive in its descriptions of environments without resulting in a bloated single language.

Chapter 5

Definition of GenerIc Self-management for Message-Oriented middleware

This chapter introduces GISMO, a generic portable Meta-level for Message-Oriented Middleware platforms. The chapter covers the challenges and solutions involved with the design of the meta-level, including the process used to identify common MOM elements for inclusion within the meta-level.

5.1 Introduction

The motivation for a MOM meta-level is to "open-up" the internal dynamics of a MOM provider, enabling the provider to be self-managed at runtime. Towards this objective, a meta- level is first defined to represent and track the state, operations, and events that exist within a MOM. The process of creating the meta-level is broken down into the following steps:

- Identify generic MOM elements for inclusion within the meta-level
- Enable portability of the meta-level between base-levels
- Design the meta-level

With the meta-level in place, the next step is to empower it with OMIP support by defining a DSL for the meta-level. This chapter covers the challenges posed with these tasks and their solutions. The first task discussed is the identification of generic MOM behaviour and state.

5.2 Identification of Generic MOM Elements

To define a generic MOM meta-level the fundamental elements of a MOM need to be identified. This includes the participants, behaviour, and state of a MOM. Once defined, these elements will be used as the building blocks for the meta-level. Given the large number of MOM implementations available, the task of identifying commonalities is a highly complex and time-consuming one.

However, a relevant standardisation effort exists in the form of the Java Message Service (JMS) specification [78].

The JMS specification defines a generic client-oriented interface for MOM interaction; it is feasible to assume that for every action (interface or method) within this specification, there is a corresponding action within the MOM implementation. The fact that the JMS specification does not force any internal provider implementation is one of the main reasons it is a widely supported format for MOM interaction. For this reason, the specification is a core information source used to derive common MOM elements. The examination starts with the identification of participants within a MOM interaction.

5.2.1 MOM Participants

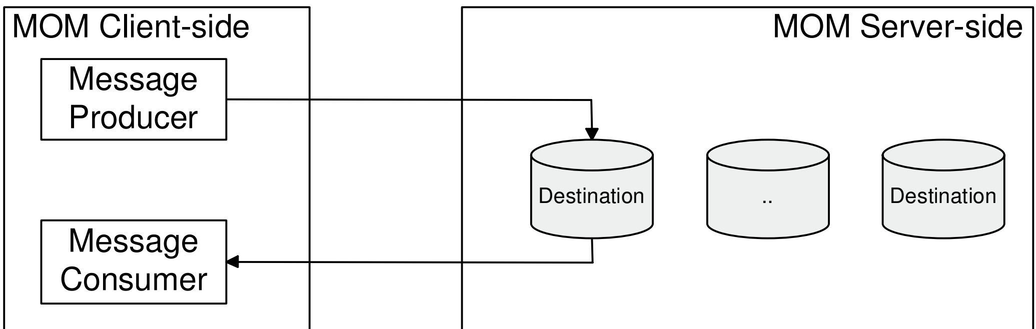


Figure 5.1 *Participants within a MOM interaction*

When examining a typical MOM deployment, illustrated in Figure 5.1, the key actors participating within a MOM interaction are MOM clients and the MOM provider/server. The role of the MOM provider is to offer messaging services to MOM clients. Given its central role, it is only natural the provider-side of the deployment is the main source of information for the meta-level; however, the client-side will also make a significant contribution to the meta-level. Note that the separation between actors in this deployment is logical, not physical. These roles are suitable to describe both federated and centralised MOM deployments.

A number of interdependencies exist between the client and provider, including provider specific libraries used by the client and configuration information. A client can also be semantically coupled to the configuration of the messaging constructs of a specific provider, such as a destination hierarchy or destination-naming schema. Even with these dependencies, both client and provider should be treated as independent entities and this separation must be recognised within any meta-level. With generic MOM participants identified, the next step is to examine their actions to identify common behaviour.

5.2.2 MOM Behaviour Identification

MOM participant behaviour is broken down along two lines, messaging behaviour and administration behaviour.

5.2.2.1 Messaging Behaviour

Messaging behaviour is classified as any activity that involves the transportation of messages between a message producer and consumer. The goal of this section is to identify a generic interface for a MOM. There are four key interactions associated with using a MOM [57, 78, 81, 86, 87].

Action	Description
SEND	Send a message to a specific queue.
RECEIVE (BLOCKING)	Read a message from a queue. If the queue is empty, the call will block until it is non-empty.
RECEIVE (NON-BLOCKING POLL)	Read a message from the queue. If the queue is empty, do not block.
LISTENER (NOTIFY)	Allows the message service to inform the client of the arrival of a message using a callback function on the client. The callback function is executed when a new message arrives in the queue.

Table 5.1 *General MOM API interface*

These actions, described in Table 5.1, reveal the core actions of a MOM. A survey detecting the presence of these capabilities within popular MOM implementations is available in Table 5.2. This survey shows that all MOM providers have the ability to receive a message using a listener, and also 7 out of 10 implementations can receive a message using a blocking or non-blocking receive command. The results of this survey reinforce the generality of the interface described in Table 5.1.

MOM Name	Send	Receive (Blocking)	Receive (Non-Blocking Poll)	Listen (Asyn)
CORBA Event Service	Yes	Yes	Yes	Yes
CORBA Notification Service	Yes	Yes	Yes	Yes
OpenJMS	Yes	Yes	Yes	Yes
ActiveMQ	Yes	Yes	Yes	Yes
SonicMQ	Yes	Yes	Yes	Yes
SIENA	Yes	No	No	Yes
TIBCO Rendezvous	Yes	Yes	Yes	Yes
REBECCA	Yes	No	No	Yes
Hermes	Yes	No	No	Yes
WebSphere MQ (formerly IBM MQ Series)	Yes	Yes	Yes	Yes

Table 5.2 *Survey of MOM messaging capabilities*

Developments within MOM platforms have evolved their capacity to include publish/subscribe capabilities. Within the publish/subscribe paradigm the concept of a subscription is used to define

the type of messages received by a message consumer [78, 87], this ability extends the core actions of a MOM provider with the additional subscription centric actions:

- *Create Subscription* – The ability to express a subscription
- *Delete Subscription* – The capability to remove a subscription

All MOM providers with publish/subscribe capabilities support these actions. Combining the generic actions described in Table 5.1 with these publish/subscribe actions would produce a generic messaging interface[1] to a MOM platform. With the messaging behaviour identified, the next step is to identify behaviour relating to the administration process of a MOM.

5.2.2.2 Administration Behaviour

Administration behaviour encompasses any activity related to the configuration or management of the infrastructure used to exchange messages between consumers and producers. Very little standardisation is provided for the definition of administration capabilities within a MOM. The JMS [78] specification does not define any administration capability or administration interface for a JMS provider or its destinations. Given the proprietary nature of administration interfaces within MOM providers, the identification of generic administration behaviour is an important step in the definition of a MOM meta- level.

To achieve this objective, common administration tasks must be identified. The diversity of MOM implementations requires the survey to be performed at an appropriate level of abstraction to identify similar capabilities. With this in mind, the key capabilities sought within the survey centre on destination administration:

- *Create Destination* – An ability to create a destination within the MOM
- *Update Destination* – A capacity to reconfigure a destination
- *Delete Destination* – The ability to delete a destination within the MOM

The results of the survey are provided in Table 5.3. The survey revealed that of the MOM platforms which posses the concept of a destination:

- All posses the ability to create a destination
- Four contain the ability to reconfigure a destination
- All except one posses the ability to delete a destination

The survey shows that these three capabilities are commonly found within MOM implementations and provide a good foundation for the creation of a generic MOM administration interface.

With the basic behaviours of a MOM in place, the focus turns to identifying generic state associated with MOM implementations.

[1] This interface is compliant with the JMS specification. Indeed, this may have well been the same process used to create the specification.

MOM Name	Create Destination	Update Destination	Delete Destination
CORBA Event Service	Yes	No	Yes
CORBA Notification Service	Yes	Yes	Yes
OpenJMS	Yes	No	Yes
ActiveMQ	Yes	No	No
SonicMQ	Yes	Yes	Yes
SIENA	-	-	-
TIBCO Rendezvous	Yes	Yes	Yes
REBECCA	-	-	-
Hermes	-	-	-
WebSphere MQ (formerly IBM MQ Series)	Yes	Yes	Yes

Table 5.3 *Survey of MOM administration capabilities*

5.2.3 MOM State Identification

The final element identified for the generic meta-level is MOM state. The objective of this task is to identify MOM state that would be useful within a self-management context. In particular, information that is not readily available in a standardised manner, such as via the JMS specification [78], is of immense benefit. Similar to the division used to identify MOM behaviour, MOM state can also be categorised into messaging state and administration state.

5.2.3.1 Messaging State

Messaging state is any information related to messaging activity. A major source of information within this category are the messages transported via the MOM. However, messages are not considered as suitable candidates for inclusion within the meta-level since they are readily available with the use of destination browsing specified within the JMS specification.

Within message centric systems, client demands are an important metric used to define current operating requirements. An accurate representation of such information would be invaluable to correctly gauge demands within the operating environment. With this in mind, the following states related to subscribers and message consumers have been identified:

- *Destination Name* - The destination associated with this subscriber/consumer
- *Filter/Selector State*
  - *Attribute Name* - Attribute identifier
  - *Operator* - Logical comparison operator used
  - *Value* - State used in comparison

MOM Name	Destination Name	Filter Attribute Name	Filter Operator	Filter Value
CORBA Event Service	Yes	-	-	-
CORBA Notification Service	Yes	Yes	Yes	Yes
OpenJMS	Yes	Yes	Yes	Yes
ActiveMQ	Yes	Yes	Yes	Yes
SonicMQ	Yes	Yes	Yes	Yes
SIENA	-	Yes	Yes	Yes
TIBCO Rendezvous	Yes	Message subject only	Subject-based addressing operators	-
REBECCA	-	Yes	Yes	Yes
Hermes	-	Yes	Yes	Yes
WebSphere MQ (formerly IBM MQ Series)	Yes	Yes	Yes	Yes

Table 5.4 *Survey of MOM messaging state*

The survey in Table 5.4 revealed that all MOM platforms that utilise a destination-like construct use a destination name to receive messages. All platforms that provided filtering capabilities contain the three filter states with the exception of TIBCO, which uses a form of subject routing that does not utilise a comparison value within the filter. The messaging state identified provides a very useful abstraction for the tracking of client demands across multiple MOM providers. The next step is to identify information related to the administration process.

5.2.3.2 Administration State

To maintain consistency, the identification of administration state follows the same abstractions used to identify administration behaviour by concentrating on the administration of destinations within a MOM. Therefore, the identification process of administration state involves pinpointing any state associated with destination administration behaviour. A survey of MOM administration capabilities sought the existence of the following state:

- *Destination Name* – A name associated with the destination
- *Destination Type* – The category of the destination (including queue, topic, and journal)

The results of the survey, available in Table 5.5, shows that for all systems utilising a destination abstraction, a name is associated with it. In addition, only two systems do not support multiple destination types.

Both messaging and administration state provide valuable information on the current configuration and environmental demands experienced by the MOM. As such, any worthwhile meta-level must capture this information. With the generic elements of a MOM meta-level identified, the

MOM Name	Destination Name	Destination Type
CORBA Event Service	Yes	No
CORBA Notification Service	Yes	No
OpenJMS	Yes	Yes
ActiveMQ	Yes	Yes
SonicMQ	Yes	Yes
SIENA	-	-
TIBCO Rendezvous	Yes	Yes
REBECCA	-	-
Hermes	-	-
WebSphere MQ (formerly IBM MQ Series)	Yes	Yes

Table 5.5 *Survey of MOM administration state*

next task is to examine the portability of the meta-level over multiple proprietary base-level MOM implementations. It is important to consider portability issues first in order to incorporate any useful design approaches into the ground-up design of the meta-level.

5.3 Designing a Portable Meta-Level

Traditional approaches to the development of a meta-level normally result in a tight coupling to the base-level. This work requires a meta-level to be portable between multiple base-levels. Generic portability presents a number of challenges for meta-level development. This section examines current approaches to meta-level design to highlight how facilitating portability requirements prompt a rethink in meta-level design and construction.

5.3.1 The Role of a Meta-Level

The meta-level of a system has a number of roles and responsibilities, however three key roles are desired:

- *Information* – The meta-level is responsible for maintaining information on the base-level's current state and the operational conditions experienced within its environment.
- *Examination* – The meta-level must provide an ability to access state information and perform relevant examinations on it.
- *Realisation* – The meta-level must be capable of altering the state information and updating the base-level to express these changes.

These meta-operations are key to providing an effective meta-level and are present within most current meta-levels including Open ORB [7], dynamicTAO [8], and K-Components [27].

5.3.2 Monolithic Meta-Level Design

With respect to the meta-operations identified, current meta-levels have taken a monolithic approach to their provision. Current implementations of self-managed systems utilise an approach using single objects or a crosscutting collection of objects to fulfil these meta-operations. This approach is illustrated in Figure 5.2 as a single object containing information, examination, and realisation operations. The example operations within this class diagram are taken from the Open ORB architectural meta-model[1].

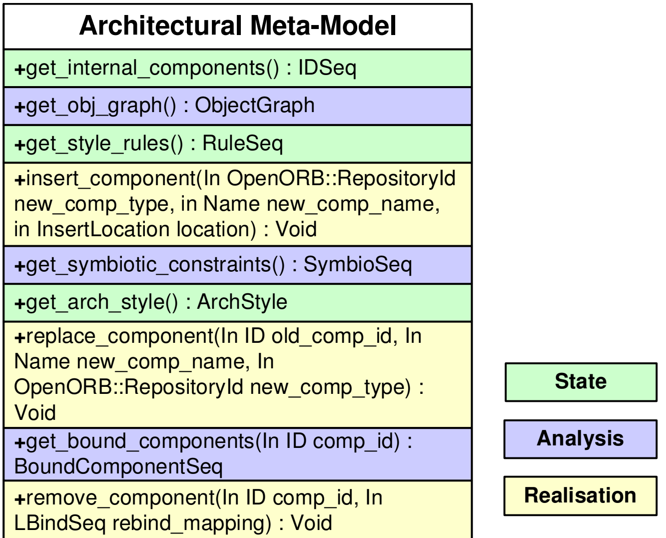


Figure 5.2 *A monolithic meta-level design*

5.3.2.1 Information

The one area where concern separation exists within current meta-levels is in the division of meta-information into multiple models. A number of projects have been developed that divide information concerns into multiple meta-models. For example, OpenCOM (Open ORB) separates the information between architecture, interface, interception, and resource models. Such division is very beneficial and simplifies meta-level interaction. Typically, this is the only dimension in which concerns are separated within a monolithic meta-level; other concern dimensions, such as meta-operations, are incorporated within the model.

5.3.2.2 Examination

Within a monolithic meta-level, the examination operations of the meta-level are included within the same objects as the meta-information and benefit from the division of information concerns. However, as the number of models increases with ever more-complex systems modelled, the complexity of the examination process will also increase. The need to consult multiple models to take a particular snapshot of the system will not only amplify the complexity of the meta-level interface, i.e. the Meta-Object Protocol (MOP), but also increase the effort required to construct

[1] This is a sample class used to illustrate the monolithic approach to meta-level design.

and analyse system snapshots. With such an approach, the MOP of a monolithic meta-level can quickly become bloated, complex, and difficult to manage and understand.

5.3.2.3 Realisation

Similar to the benefits obtained by the examination process, the division of information concerns is also beneficial to the realisation meta-operation. Within the monolithic approach, the same objects that contain the information and examination concerns perform ratification of the meta-level. Current meta-levels have only been required to realise changes to a single base-level implementation, in the case of Open ORB a single ORB implementation. A monolithic meta-level is tightly coupled to its base-level implementation, reducing the portability of the meta-level between alternative base-levels.

A requirement of this research is to produce a meta-level that is portable across multiple base-levels. While one cannot avoid the work required to ratify a meta-level to its base-level or the work required to examine the meta-level, some benefit can be achieved with a further separation of concerns along the meta-operation dimension within the meta-level.

To assist in the design of meta-levels that meets these challenges, a separation of concerns similar in spirit to the Model-View-Controller (MVC) design pattern is proposed for meta-operation concerns within a meta-level. Before discussing this new separation, a brief summary of the MVC design pattern is presented.

5.3.3 The Model-View-Controller Design Pattern

User interaction within Smalltalk-80 centres on a framework known as Model-View-Controller or MVC. This design pattern, illustrated in Figure 5.3, allows for the decomposition of an applications interface into three parts: the model, the view, and the controller.

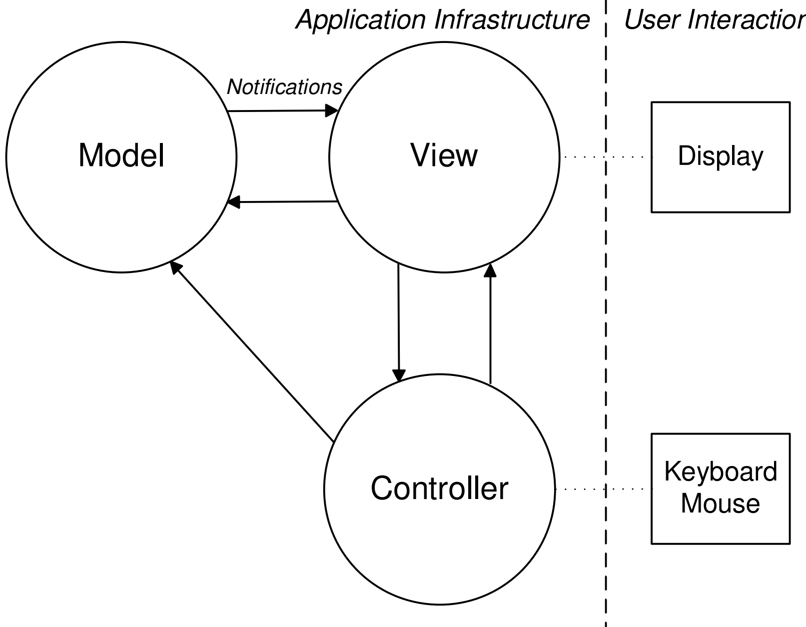


Figure 5.3 *The Model-View-Controller (MVC) design pattern*

"The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model" [112].

By separating the three elements of GUI interaction, the system is more competently able to cope with changes.

- Isolation of the data within the *Model* makes it independent from:
  - How it is viewed/rendered for the user
  - User input behaviour
- The V*iew* presents information contained in the model to the user. Multiple views of the model are possible to provide alternative representations of the model. The number and type of views will not affect the model.
- *Controllers* receive user input or commands to alter the model. The controllers contain the logic to alter the model. Any view that allows the model to be altered will use a controller to perform the relevant changes. Controllers may be general to all views or may be associated with a specific view.

5.3.4 Meta-State Analysis Realisation (M-SAR) Design Pattern

Motivated by the mobility challenges faced by a portable meta-level, the Meta-State Analysis Realisation (M-SAR) design pattern proposes a new separation of concerns along the meta-operation dimension within a meta-level. Inspired by the MVC design pattern, successfully used to simplify ever-increasing complexity with user interaction, M-SAR separates the three main operations of a meta-level, information, examination, and realisation, into distinct encapsulated entities. The proposed restructuring of the meta-level is presented in Figure 5.4.

With this approach the meta-level is broken into three parts, decomposing the meta-level into the following entities:

- *State* – Similar to the model objects within MVC these objects only contain meta-information in order to separate it from the realisation and examination procedures.
- *Analysis* - Examination of meta-state is performed within analysis objects. These entities have the responsibility to construct/render views of the meta-state from a particular snapshot. There is no restriction on the composition of analysis objects; they may be created using a mixture of information from multiple state objects. Analysis objects may also perform computations on the meta-state and augment it with additional information, allowing for a highly customised analysis of the meta-level.
- *Realisation* - The final part of the pattern covers the realisation meta-operation. Realisation entities are responsible for the ratification of the meta-level to the underlying base-level. The realisation process requires direct interaction with the base-level; as such, realisation objects are tightly coupled to a specific base-level. However, these objects encapsulate this base-level interaction coupling and decouple the rest of the meta-level (state and analysis objects) from

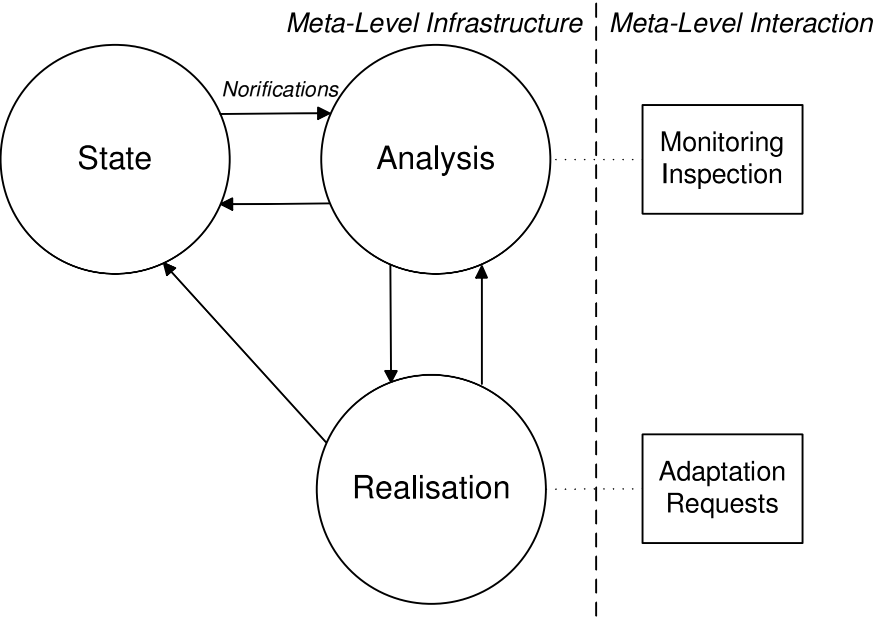


Figure 5.4 *The Meta-State Analysis Realisation (M-SAR) design pattern*

the base-level. Multiple realisation objects may be used to ratify the meta-level to a variety of base-level implementations.

The M-SAR design pattern is a vital tool in the design of portable meta-levels. The benefits of the M-SAR design pattern are discussed in the next section.

5.3.5 Benefits of a Separated Meta-Model Design

The main barrier to the portability of a meta-level is the ratification process. This part of the meta-level is tightly coupled to the base-level. Insulating as much of the meta-level from the realisation process produces a more controlled, looser coupling between the meta-level and underlying base-level. The M-SAR design encapsulates base-level interaction to minimise the changes required to replace the base-level implementation.

Within reflective systems, information analysis is paramount to directing meaningful and positive changes to the base-level; a successful adaptation is dependant on the quality of information available. As systems are modelled in more detail, the complexity of meta-levels and their examination will also increase. The use of multiple models within an examination can lead to inter-model dependencies that promote poor encapsulation. Analysis entities within the M-SAR pattern encapsulate the examination process within specific objects. This separation improves encapsulation within the meta-level, reduces the size of the basic MOP, and enables the creation of highly specialised views of the meta-level to describe specific parts of the system. This approach allows for comprehensive environment descriptions, without creating a bloated single-object interface (MOP).

The monolithic meta-level design discussed in Section 5.3.2 is recast using the M-SAR design pattern. The resulting meta-level is illustrated in Figure 5.5. Within this new design, the state, analysis, and realisation meta-operation concerns are separated into independent objects,

simplifying the interface to the meta-level and reducing coupling between concerns.

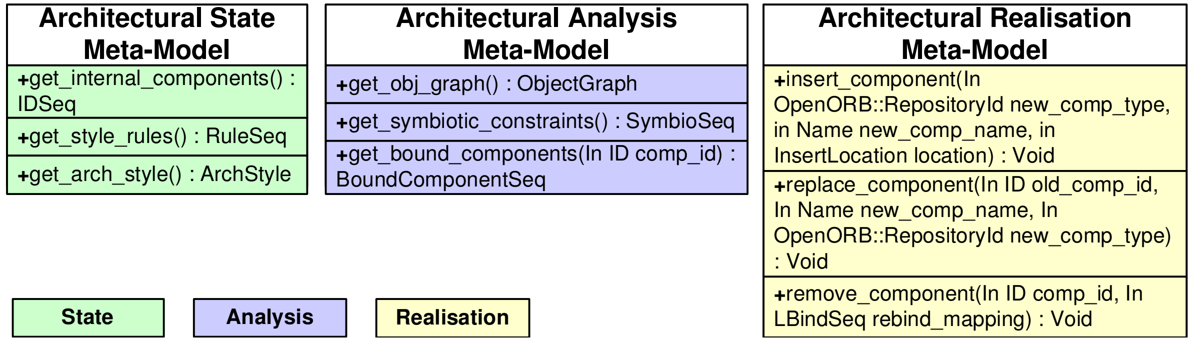


Figure 5.5 *An M-SAR based meta-level design*

5.4 GISMO: GenerIc Self-management for Message-Oriented middleware

The GenerIc Self-management for Message-Oriented middleware (GISMO) has been defined to provide a general-purpose meta-level that can be applied to a number of MOM providers. GISMO represents the generic participants, behaviour, and state identified for MOM within Section 5.2. The GISMO meta-level, illustrated in Figure 5.6, separates concerns along two dimensions, information concerns and meta-operation concerns. Information concerns are separated into three distinct sub meta-models covering destinations, subscriptions, and interception. The meta-level also contains an event model and reflective engine. The design of each sub meta-model uses the M-SAR design pattern to provide separation of meta-operation concerns. The remainder of this section examines each of these sub-models in detail, covering their state, analysis, and realisation processes. Before examining each of the sub-model, the roles of MOM participants are first discussed.

5.4.1 Client and Provider Roles

A central objective of a meta-level is to provide additional information on the operations of a base-level. Within the MOM domain, the two participants of the base-level are MOM clients and the MOM provider. While these two participants are independent of one another, they both contain vital information to the make-up of a useful meta-level. With this in mind, GISMO offers a meta-level that incorporates both the MOM client and provider. Using this approach, neither the client nor provider will be able to construct an entire GISMO meta-level without coordinating with one another. This unified approach enables a streamlined modelling of a MOM deployment and promotes the exchange of information and coordination between clients and providers. Exchange of meta-level information can be achieved with the use of OMIP introduced in Chapter 4; further information on OMIP support within GISMO is available in Section 5.5.

The core of the GISMO meta-level exists on the provider. In a typical MOM deployment, the provider is assumed to fully implement its side of a GISMO meta-level. While it is possible for

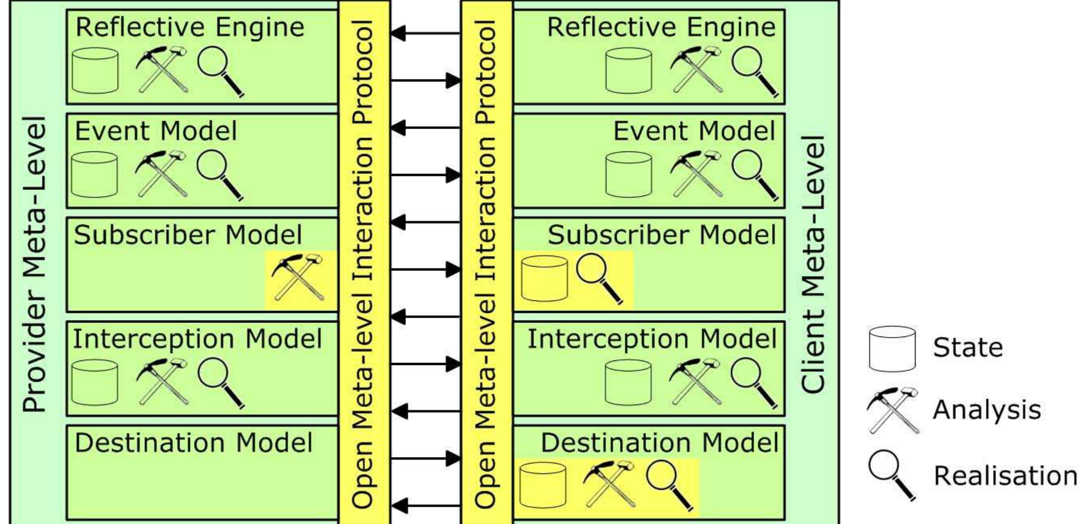


Figure 5.6 *The GISMO abstract meta-level design*

a client to interact with a provider that does not posses a GISMO meta-level, the capabilities of a client-side only meta-level are limited. However, unlike providers, clients have a choice of three levels of compliance:

- Non-Compliance - A traditional client that does not possess a meta-level.
- Minimum Compliance - The most basic level of compliance requires a client to inform the provider meta-level of its actions. Given that GISMO is designed to sit upon an underlying MOM implementation, client notifications are vital for the provider meta-level to be aware of the workings of the base-level.
- Full Compliance - A client that fully implements the client-side elements of the GISMO meta-level and provides support for open access to its meta-level via OMIP.

The following sections examine the sub-models within the GISMO meta-level, as each of these meta-models is discussed its relevance to the client or provider-side is highlighted.

5.4.2 Destination Meta-Model

The first sub-division within GISMO is the Destination meta-model. Destinations are the mechanism used for information exchange within a provider and a destination model is vital to the understanding of a provider's internal structure.

5.4.2.1 State

The model used to track destination state is the *Destination State Model* (DSM). In its basic form, the DSM tracks the existence and basic configuration of a destination. The basic unit of storage with the DSM is the *Destination* structure. This standard structure for destination state storage contains basic information about individual destinations, the information tracked in this structure is:

- *Destination ID* – A unique identifier for the destination
- *Destination Name* – The name associated with the destination
- *Destination Type* – Is the destination a queue of a topic
- *Destination Routing Condition* – The routing restrictions, if any, are placed on messages in the destination

The destination structure tracks information on either a queue or topic. If the destination forms part of a destination hierarchy, additional information needs to be captured by the DSM to track the intimate interconnected relationships between destinations within a hierarchy.

State on an individual hierarchy is stored within the *Hierarchy* structure; this simple structure contains a link to the root destination of the hierarchy and the collection of nodes within the hierarchy.

- *RootNode* – The root HierarchyNode of the hierarchy
- *Nodes* – The collection of HierarchyNodes for this hierarchy

Destinations within the hierarchy need to store additional relationship information. The *HierarchyNode* achieves this by extending the basic DSM with the following relationship information:

- *ParentNode* - The parent of this node, if any
- *ChildNodes* - The children of this node, if any

These two entries connect to other *HierarhcyNode* structures to allow the formation of a larger interconnected hierarchical structure to enable the modelling of destination hierarchies. This concept is illustrated in Figure 5.7.

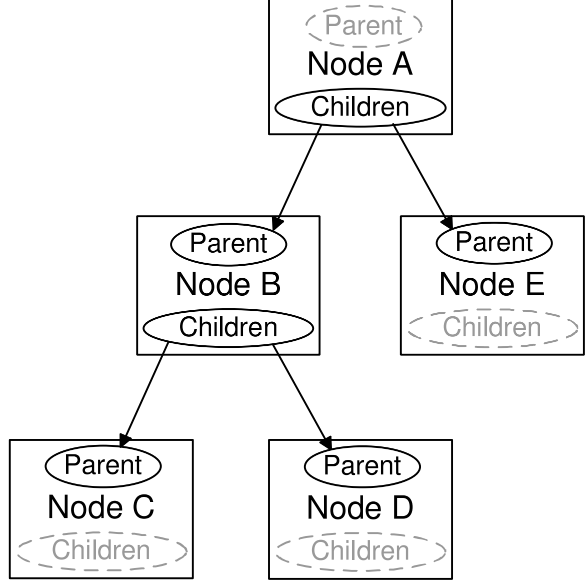


Figure 5.7 *A destination hierarchy example*

The DSM stores both Destination and Hierarchy structures within two groupings:

- *Single Designations* – Collection of destination structures (Queues or Topics)
- *Destination Hierarchies* – Collection of Hierarchies with embedded HierarchyNode structures (Topics in a Hierarchical Namespace)

With the use of these structures, the destination meta-model has the capability to represent relevant information for destination constructs within a MOM.

5.4.2.2 Analysis

The role of meta-analysis modules within GISMO is to provide an encapsulated method of examination, allowing highly specialised views of the meta-level to describe specific parts of the system. There is no limitation on the type of analysis performed within these modules. Modules obtain their information from one or more meta-state models and may gather additional information from injecting monitors at specific locations using the interception meta-model. To demonstrate these principles a selection of sample meta-analyses is presented for each sub-model.

Destination Search Analysis

The Destination Search Analysis (DSA) is a basic analysis designed to search the destination meta-model for the existence of a specific destination. The DSA illustrates how analysis entities can perform simple examinations of a meta-level. The analysis contains the following operation:

- *DestinationSearch(String destinationName*)

This analysis supports the use of wildcard to find similar named destinations, i.e. "Ed*" would return the destinations "EdInbox" and "EdDeleted".

5.4.2.3 Realisation

The primary rationale for meta-realisation objects is to encapsulate interaction with the base-level of the system. With this encapsulation in place, it is possible to change the base-level implementation while limiting changes to these objects. A standardised interface for meta-realisation would further simplify base-level interaction. With such an interface in place, base-levels are swappable without any disruption to the meta-level.

The central meta-realisation within GISMO is the Destination Administration Realiser (DAR). The DAR is responsible for making changes to the destination model and realising these changes to the underlying MOM provider. The DAR allows the creation, updating, and deletion of destinations and destination hierarchies (at both the hierarchy and hierarchyNode level). The basic DAR has the following operations:

- *CreateDestination(DestinationModel destModel*
- *UpdateDestination(DestinationModel currentDestModel, DestinationModel newDestModel)*
- *DeleteDestination(DestinationModel destModel)*

5.4.2.4 Client-side/Provider-Side Notes

Within a MOM deployment, destinations are located in the provider. The logical location of the DMS is the server meta-level. The destination meta-level may be accessed from the client meta-level with the use of OMIP.

5.4.3 Subscription Meta-Model

The second information concern separation within the meta-level is used to track message consumers. Message consumers are the main force that defines the current operating requirements and workload within a MOM environment; as such, the Subscription meta-model is an important source of information for obtaining the current operational demands of the MOM.

5.4.3.1 State

The *Subscription State Model* (SSM) is a state-model designed to contain information on client usage within a MOM. The objective of the SSM is to monitor client activity by tracking the subscription details of message consumers. The basic model tracks the following subscription details:

- *Subscriber ID* – Unique identifier for this subscription
- *Destination Name* – Identifier of the destination associated with this subscription
- *Filter/Selector State (collection)*
  - *Attribute Name* - Attribute identifier
  - *Operator* - Logical comparison operator used
  - *Value* - State used in comparison

Potentially, the model could be expanded with information in a number of areas such as client configuration, connection configuration (durability or transactional settings), and subscription history. Additionally, the model may also cover information specific to a JMS provider or application domain, such as mobile or ubiquitous environments. However, within the scope of this work, the SSM tracks only basic subscription information.

5.4.3.2 Analysis

Examination of the subscription meta-model offers a great deal of potential for constructing snapshots of the provider's currents environmental conditions. As an illustration of the benefits of multi-model analysis, the Destination Subscription Analysis is presented.

Destination Subscriptions Analysis

The Destination Subscription Analysis performs an examination of the subscribers to a specific destination by combining information from the *Destination* and *Subscriber* state-models. This simple correlation of information contained directly within the two models reveals the number of subscribers for a specific destination.

- *SubscriberCount(DestinationModel destModel)* – Returns the total number of subscribers to the destination

Given the close relationship between destinations within a hierarchy, the meta-analysis provides an additional query to preserve this relationship between HierarchyNodes. This analysis reveals the number of subscribers to a specific part of a hierarchy:

- *TotalNodeSubscriberCount(HierarchyNode hierNode)* - Returns total number of subscribers to the local node and all the subscribers of its descendants
- *LocalNodeSubscriberCount(HierarchyNode hierNode)* - Reveals the total number of local subscribers to a specific node
- *ChildrenNodeSubscriberCount(HierarchyNode hierNode)* - Returns the number of subscribers to the children of the specified node

Further subscription analysis is available with the examination of subscription filters. Snapshots of this information may be presented in a number of ways, two examples are provided to illustrate this concept:

- *DestinationFilterCount(DestinationModel destModel)* - Exposes the number of filters used in subscriptions on this destination
- *DestinationCommonFilterCount(DestinationModel destModel)* - Reveals the most common filters used in subscriptions on this destination

The queries presented in this section show the freedom available for creating highly specialised analysis. Each query type is encapsulated within its own object ensuring the basic MOP for the overall meta-level is kept to a minimum.

5.4.3.3 Realisation

The realisation process within the subscription meta-model is unique among the sub-models of GISMO. Message consumers maintain ownership and control over their subscriptions; as such, it is not possible for the provider to alter a client's subscription. This requires the realisation task to be located within the client-side meta-level. If the provider wishes to alter the subscription meta-state, it must request that the relevant subscription owner alters their subscription. This request is carried out using the OMIP implementation within GISMO. Using the interaction protocol, the realisation of the subscription meta-model is performed on the client-side of the deployment, with the client altering their subscription. Two other actions exist within the realisation process of the subscription meta-level, creating a new subscription, and deleting a current subscription. Both of these methods are available to subscription owners, allowing them to update the meta-level with their subscription activity. The complete realisation interface consists of:

- *CreateSubscription(SubscriptionModel subModel)*
- *UpdateSubscription(SubscriptionModel currentSubModel, Subscription newSubModel)*
- *DeleteSubscription(SubscriptionModel subModel)*

5.4.3.4 Client-side/Provider-Side Notes

The subscription meta-model is of particular interest when examined from the client/provider divide. The model and analysis is contained within the providers meta-level, for the client to access this portion of the meta-level it must perform an OMIP interaction with the providers meta-level. However, since each individual client maintains ownership on their subscriptions, any changes to the meta-level must be realised by the client. Thus, if the request is made from the

client-side of the subscription owner, the client need only inform the server of the update to its state. However, if the server wishes to alter the meta-level is must send an OMIP request to the client for the change. This process illustrates the need for an open interaction protocol at the meta-level; both systems are capable of requesting information and realisations from the other.

5.4.4 Interception Meta-Model

The next information concern within GISMO is the Interception Meta-Model (IMM). This meta-model details and tracks call-interception and functionality injection at specific locations or point cuts within the base-level. Unlike the previous sub-models covered the IMM does not track an internal state within the MOM, rather it provides information on the current state of interception operations. The tracking of this information is vital within a self-managed reflective system as interception is a flexible approach for monitoring, altering, and extending behaviour of a base-level at runtime. For example, code may be injected to execute every time a new subscriber is added to the system, or when a message is sent on the client.

Points of injection available within the model are located on both the client and provider-side. An interception point is a place of execution within a system where code can be triggered to inject functionality. A good quality injection model will identify appropriate locations for interception points, such as destination administration, subscription administration, and message processing, to allow functionality to be enhanced both in and out of-line with base-level execution.

5.4.4.1 State

Within GISMO, the Interception state-model is used to track information relating to interception within the base-level. At each interception point, the state-model maintains a list of the interceptors injected at the location. The structure used to store this information is:

- *Interceptor Name* – The name identifying this interceptor
- *Interception Point* – The interception point(s) associated with the interceptor
- *Interceptor Class Name* - The class defining the interceptor
- *Scope* – The execution scope of the interceptor
- *Configuration Options* – Collection of associated configuration information for the interceptor
  - *Option Name*, *Value*

Interceptors must be associated with an interception scope. Interception scopes are used to associate interceptors with specific types of call-interceptions, the following interception scopes may be associated with a handler attached to an interception point:

- *Global-Scope* - System-wide (all destinations)
- *Local-Scope* - per destination/destination group
- *Hierarchy-Scope* - per branch

Global-scoped interception points are design to operate on all calls intercepted within that point category. All globally scoped interceptors are triggered before interceptors of any other scope; the global scoping is designed for attaching system-wide handlers such as auditing, logging, or usage monitoring.

Local-scoped interception works on a per-destination basis, allowing interceptors to be attached to a single destination or a group of destinations. Once a call is intercepted (message published, subscription added, etc.) for a given destination, the interceptors attached to that destination (if any) will be triggered. These interception points allow functionality to be added/extended at the destination level

The hierarchy interception scope enables interceptors to be associated with a destination within a hierarchy. Similar to local-destination interception points, hierarchy interception points work on the principle that each branch (destination) of the hierarchy can have local interceptors associated with it. The absolute interceptor collection for a branch consists of the local interceptors and the absolute interceptor collection of its parent, this recursive approach to interception continues up the tree until it reaches the root channel. This results in a very powerful mechanism for processing messages submitted to a destination hierarchy structure. This mechanism is similar to inheritance within object-oriented programming, where an object (branch destination) inherits the functionality of its ancestors (parents handlers) and can augment this functionality with its local implementation (local handlers). Scoping of interception within a hierarchy allows for the subjective addition of interceptors throughout the hierarchy in a manner that is consistent with ancestry relationships within the hierarchy.

As listed in Table 5.6, seven suitable interception points have been identified within a MOM base-level, these points exists on both the client and provider.

Interception Point	Injects Functionality
CreateDestination	When a destination is added.
UpdateDestination	When a destination is reconfigured.
DeleteDestination	When a destination is deleted.
CreateSubscription	When a subscription is added.
DeleteSubscription	When a subscription is removed.
SendMessage	When a message is sent.
ReceiveMessage	When a message is received.

Table 5.6 *Interception points within a MOM base-level*

5.4.4.2 Analysis

The meta-analysis for the IMM is a simple query allowing the lookup of interceptors used at each point within the framework to reveal the current interception activity at that location:

- *AllInterceptors(InterceptionPoint iPoint)* – All interceptors at the interception point
- *GlobalInterceptors(InterceptionPoint iPoint)* – Only global interceptors at the interception

point

- *LocalInterceptors(InterceptionPoint iPoint, Destination destModel)* – Local interceptors for a specific scope at the interception point

With the use of this meta-analysis, it is possible to obtain an accurate snapshot of current interception activity within the meta-level.

Interception in itself is an extremely valuable tool in the creation of meta-analyses. The use of interception can greatly enhance the quality of the examination process, providing highly detailed analysis that would not be possible with a state-model only approach. To illustrate the usefulness of interception within the analysis process, a sample interception enhanced analysis is discussed.

Destination Usage Analysis

The Destination Usage Analysis illustrates how an analysis process within the meta-level can be enriched with additional information sources. With the use of interception, a simple message-monitoring utility called the destination profiler is inserted at the "SendMessage" interception point on the provider-side with a global scoping. The destination profiler provides the capability to track the message traffic of a particular or group of destinations within the provider. The profiler may be configured to log specific information about each message that is sent to a destination, such as message type, attribute information, sender, etc. With such information available, it is possible to build a profile of a destination's usage. Information recordable by the profiler includes:

- Message totals for:
  - Destinations
  - Hierarchies (Total, Local, Children)
- Time-base message throughputs for: (Destinations, Hierarchies)
  - Hourly, Daily, Weekly, Monthly

With such a large quantity of information the storage and analysis of the profile data becomes an issue. This analysis process can be enhanced with database access to support large volumes of message traffic; databases also provide a powerful query language SQL that may be used to perform analysis on the data. The encapsulation of such capabilities is a key benefit of the M-SAR design pattern. Use of this pattern in conjunction with interception provides a powerful method of providing highly customised analytical capabilities.

5.4.4.3 Realisation

The interception realisation process has responsibility for managing the use of interception throughout the MOM base-level. The realisation process may not only be responsible for administrating interception but may have a significant role in its implementation. If the MOM platform does not provide support for interception or only provides limited support, the responsibility for the provision of interception capabilities rests with the interception realisation process. The core interface for the Interception realisation is:

- *CreateInterceptor(Interceptor interceptor)*

- *DeleteInterceptor(Interceptor interceptor)*

When examined from the perspective of the base-level, all interceptors are assumed to be in-line with the synchronous execution of the system. However, no limitation on interceptor implementation exists, allowing the triggering of an out-of-line asynchronous execution.

5.4.4.4 Client-side/Provider-Side Notes

The interception meta-level provides the clearest divide between the client and provider with both entities providing independent interception meta-models. If either the client or the provider wishes to interact with the other's interception meta-model they must utilise the OMIP.

5.4.5 Meta-Level Event-Model

The GISMO meta-level provides a first-class event model with a range of event sources. As one would expect from such a model, it covers all the fundamental action within the MOM environment including messaging events, administrative events, and time-based trigger events.

Event consumers use the event model to register an interest in a specific event that may take place within the meta-level. When the event occurs, all registered consumers are notified of the occurrence. It is important to note that the event-model is not an in-line execution mechanism; it simply provides an asynchronous non-blocking notification of event occurrence. In-line execution can be facilitated using the interception meta-level. A full list of events contained in the GISMO event model is described in Table 5.7.

Event Category	**Event Name**	**Description**
Destination	CreateDestination	Event fires when a new destination is created on the provider.
Destination	UpdateDestination	Event fires when a destination is reconfigured.
Destination	DeleteDestination	Event fires when a destination's configuration is deleted on the provider.
User	CreateSubscription	Event fires whenever a new subscription is requested by a consumer.
User	DeleteSubscription	Event fires when a consumer unsubscribes from a destination.
Message	SendMessage	Event fires when a message is produced.
Message	ReceiveMessage	Event fires when a message is consumed.
Time-Triggered	Start/Stop	Run a task or operation between specified times. Fires a 'Start' event at time A, then fires a 'Stop' event at time B.
Time-Triggered	One-Shot	Run a task or operation once. Fires a 'Run' event at a specific time X.
Time-Triggered	Repeat	Repeat a task or operation at specific intervals. Fires a 'Run' event every X.

Table 5.7 *Event types within the GISMO event model*

Time-triggered events within the event model are not derived from a MOM event source and have been included to provide a simple integrated method of triggering time-sensitive tasks such as routine maintenance or monitoring activities.

5.4.5.1 Client-side/Provider-Side Notes

Similar to the interception meta-model, the event-model also has a clear separation between the client and provider with both possessing independent event models. Participants of client or provider meta-level can subscribe to the event model of the other with the use of the OMIP.

5.4.6 Reflective Engine

Reflective capabilities within the GISMO meta-level are encapsulated within the reflective engine. Present on both the client and provider-side, the reflective engine directs reflective actions within the meta-level. The objective of the reflective engine is to define locations within the meta-level where reflective computations may occur and to provide a standard interface for reflective policies. This section examines the reflective engine describing its policy structure and policy call sequence.

5.4.6.1 Reflective Policies

In a similar fashion to other meta-level realisations within GISMO, the reflective meta-level contains basic operations to add, update, and remove policies as well as simple analysis capabilities to search the state space. The reflective meta-level contains a state model used to configure policies:

- *Policy Name* – The name identifying the policy
- *Reflective Location* – The location(s) at which the policy is invoked
- *Class Name* – The class defining the policy
- *Scope* – The execution scope of the policy
- *Synchronisation* – Is the policy executed 'in-line' or 'out-of-line'
- *Configuration Options* – Collection of associated policy configuration
    - *Option Name*, *Value*

Reflective policies are an encapsulation mechanism for reflective computational logic. The structure of a policy is a straightforward interface with initialisation, execution, and cleanup operations as illustrated in Figure 5.8.

Once a policy implements this interface, it can be included within the policy repository and used within the meta-level. Reflective policies may be associated with a number of activities within the meta-level as detailed in Table 5.8.

Once attached to a reflective location, the next step is to decide the synchronisation of the policy. Reflective policies may be placed both in-line with system execution for synchronous reflection or placed out-of-line for asynchronous reflection. Similar to interceptors within the interception meta-level, reflective policies may also be associated with a global or local destination scoping. Once in position, reflective policies are able to perform their reflective duties utilising the full resources of the meta-level to effect change within the system.

Serializable
interface
ReflectivePolicy

+init:void
+executePolicy(refContext:ReflectiveContext):void
+onFault(refContext:ReflectiveContext):void
+cleanup:void
+setName(name:String):void
+getName:String
+setOption(name:String, value:Object):void
+getOption(name:String):Object
+setOptions(opts:Hashtable):void
+getOptions:Hashtable

Figure 5.8 *Reflective policy interface*

Reflection Location	Description
CreateDestination	Triggers reflective policies when a new destination is created on the provider.
UpdateDestination	Triggers reflective policies when a destination is reconfigured.
DeleteDestination	Triggers reflective policies when a destination is deleted on the provider.
CreateSubscription	Triggers reflective policies when a new subscription is requested by a consumer.
DeleteSubscription	Triggers reflective policies when a consumer unsubscribes from a destination.
SendMessage	Triggers reflective policies when a message is produced.
ReceiveMessage	Triggers reflective policies when a message is consumed.

Table 5.8 *Reflective locations within GISMO*

5.4.6.2 Policy Call Sequence

Given the number of options that can affect the execution of a reflective policy, the exact call sequence order for reflective policies is as follows:

- Associated event/behaviour occurrence
- Global Policies
  - Global synchronous policies invoked
  - Global asynchronous policies invoked
- Local Policies
  - Local synchronous policies invoked

– Local asynchronous policies invoked

An illustration of this call sequence is provided in Figure 5.9.

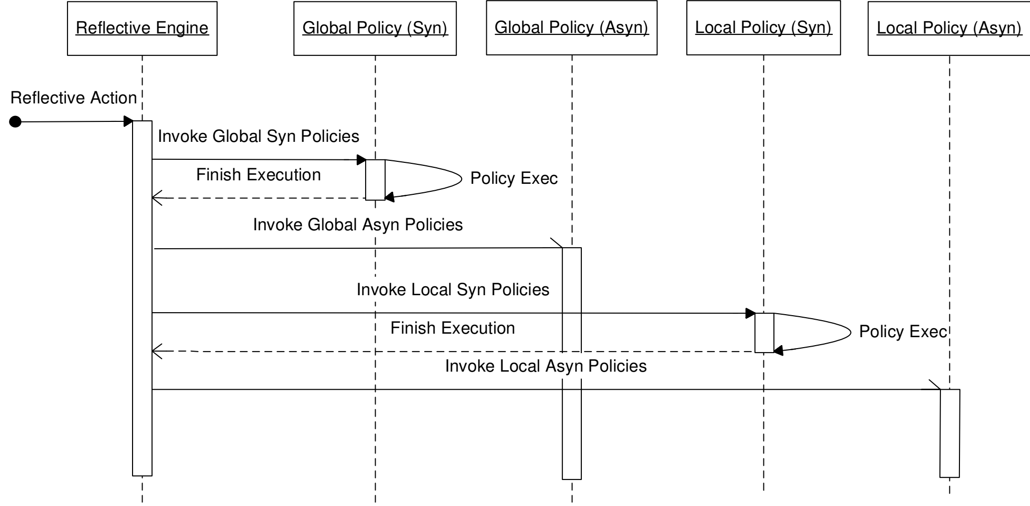


Figure 5.9 *Policy call sequence within the reflective engine*

5.4.7 Extending the Meta-Level

The overall objective of GISMO is to provide a generic meta-level for MOM. The majority of MOM implementations share common behaviours and capabilities; however, they may also contain some form of proprietary functionality. While the basic GISMO meta-level is not intended to cover such functionality, it can be easily extended in a controlled manner to include such functionality without affecting core portability.

The core of GISMO covers the basic common elements of a MOM within its sub-models and interfaces. These core interfaces and models may be extended to provide support for proprietary MOM behaviour and state. As long as the proprietary model or interface inherits from or implements the relevant core interface or model, portability is maintained. This process is illustrated with an example providing a SonicMQ specific extension of the destination meta-model. The core model is extended to include proprietary information specific to the configuration for a SonicMQ provider. The SonicMQ extension, illustrated in Figure 5.10, adds the following proprietary state to the destination state model:

- SonicMQ General Properties
  - *Enable Queue Cleanup* - Enable periodic checking of queues for expired messages
  - *Cleanup Interval* - Specify the approximate time interval in seconds between successive searches for expired messages
  - *Number of Delivery Threads* - Number of dispatch threads used for dequeueing messages from queues
  - *Maximum Temporary Queue Size* - Maximum size (in KBytes) of temporary queues

- Destination Specific
  - *Save Threshold* - Maximum total size in KB of messages that can reside in memory
  - *Maximum Size* - Maximum total size in KB of en-queued messages stored

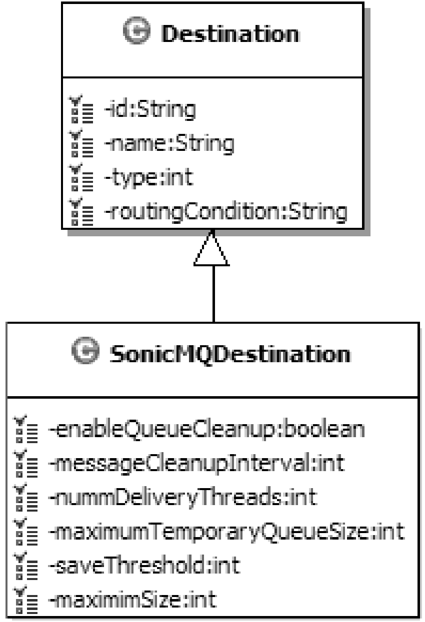


Figure 5.10 *SonicMQ proprietary state model*

A SonicMQ specific meta-realisation may utilise this information when creating the destination, while a non-SonicMQ meta-realisation simply ignores this information and uses the state from the standard GISMO destination meta-model.

5.5 MOM-DSL: Opening GISMO

The role of the Message-Oriented Middleware-Domain Specific Language (MOM-DSL) is to express the GISMO meta-level in a medium that is transferable between participants. Using the MOM-DSL, participants can request state and actions from a self-managed MOM platform. The objective of this section is to give an overview of the MOM-DSL and the role it plays in opening the GISMO meta-level. This demonstrates how the MOM-DSL expresses the GISMO meta-level and fulfils OMIP compliance. Section 5.6 presents a number of sample MOM-DSL interactions and a full definition of the MOM-DSL XML Schema is provided in Appendix B. Discussion starts with an overview of the message exchange infrastructure.

5.5.1 Message Exchange Infrastructure

Before a MOM-DSL is defined, it is important to characterise the medium used to express the MOM-DSL and the mechanism used to exchange messages between participants. The first of these issues addressed is the medium used to express messages.

5.5.1.1 Message Format

The format used to express MOM-DSL messages must do so in a technology neutral manner. Given the diversity of the MOM domain it is important that the message medium is compatible with as many environments as possible. XML is a popular medium for the exchange of information within heterogeneous environments and application integration scenarios. Given the standardisation and widespread acceptance and support for XML within MOM implementations it provides an ideal expression medium for the MOM-DSL.

5.5.1.2 Message Transport

Given that MOM provides communication services between participants, it is the ideal mechanism for the exchange of MOM-DSL messages. Utilising the underlying MOM simplifies the task of exchanging MOM-DSL messages and only requires the definition of relevant destinations to facilitate message exchange. Each participant needs a destination to receive MOM-DSL requests, to meet this need the "<ParticiapntName>.GISMO.Inbox" destination may be used by external entities to send MOM-DSL requests. Replies to a request are sent using the *Reply-To* field of the message containing the request; any valid reply-to destination is acceptable including temporary destinations.

Additional destinations may also be used to transfer MOM-DSL related information such as event notifications. A high-level overview of the message exchange infrastructure is illustrated in Figure 5.11. With the infrastructure of the protocol in place, the next step is to define the MOM-DSL.

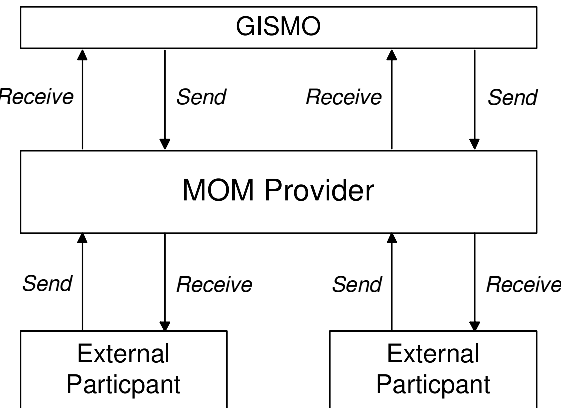


Figure 5.11 *MOM-DSL message exchange infrastructure*

5.5.2 State Structure

Representing GISMO state within the MOM-DSL is a simple process of expressing the state models within XML. An illustration of this process is presented in Table 5.9 using sample interceptor state. A full listing of all XML Schema for GISMO state models is available in Appendix B.

```
<Interceptor name="Log" interceptionPoint="CreateDestination"
              className="ie.nuigalway.ecrg.chameleon.interceptor.Log" scope="">
  <Option name="message" value="Destination Created"/>
</Interceptor>
```

Table 5.9 *GISMO interceptor state expressed in MOM-DSL*

5.5.3 Available Actions

Actions from the GISMO meta-level, such as realisations and analyses, are expressed as interaction commands within the MOM-DSL. As illustrated in Figure 5.12, the six interaction commands within the MOM-DSL are:

- *Capability Request* – Interaction command used to obtain the available capabilities
- *State Request* – Used to retrieve meta-state
- *Create* – Generic command to create an entity within a meta-level
- *Update* – Generic command to update a meta-level entity
- *Delete* –Generic command to delete an entity within a meta-level
- *Analysis* – Utilised to obtain the results of an analysis

The three realisation commands are of particular interest as they provide generic realisation operations for all meta-levels within GISMO. Examples of these and other commands in operation are provided in Section 5.6 and a full XML Schema of the commands is available in Appendix B.

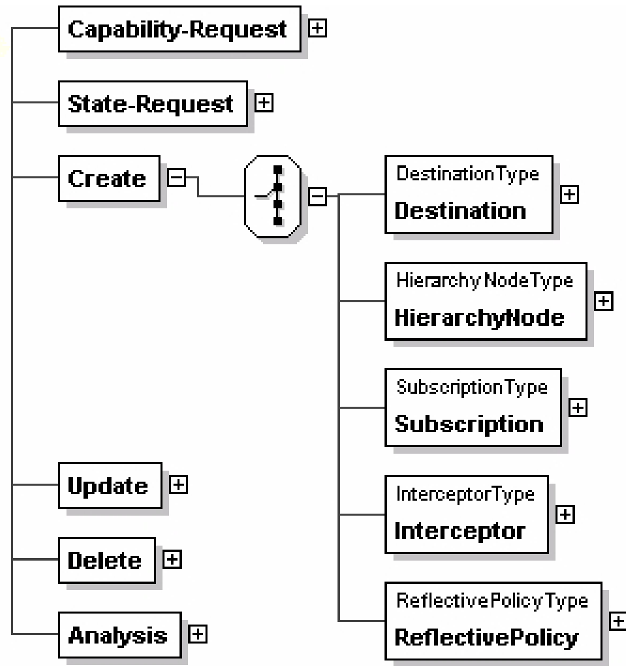


Figure 5.12 *MOM-DSL command structure*

5.5.4 Capabilities Request

The *Capability Request* of the MOM-DSL is required for OMIP compliance and is not a native part of the GISMO meta-level. The role of a capability-request is to inform a participant of its level of access to the meta-level. As illustrated in Figure 5.13, meta-level access is broken down along the same lines of division used for the GISMO meta-level. Access to each sub meta-model is then broken down into specific actions for that meta-model such as state and creation requests. This approach allows access control to be specified in a flexible granular manner. In addition, both event and analysis capabilities may be expressed in a capability request. A sample capability request is provided in Section 5.6 with full XML Schema detailed in Appendix B.

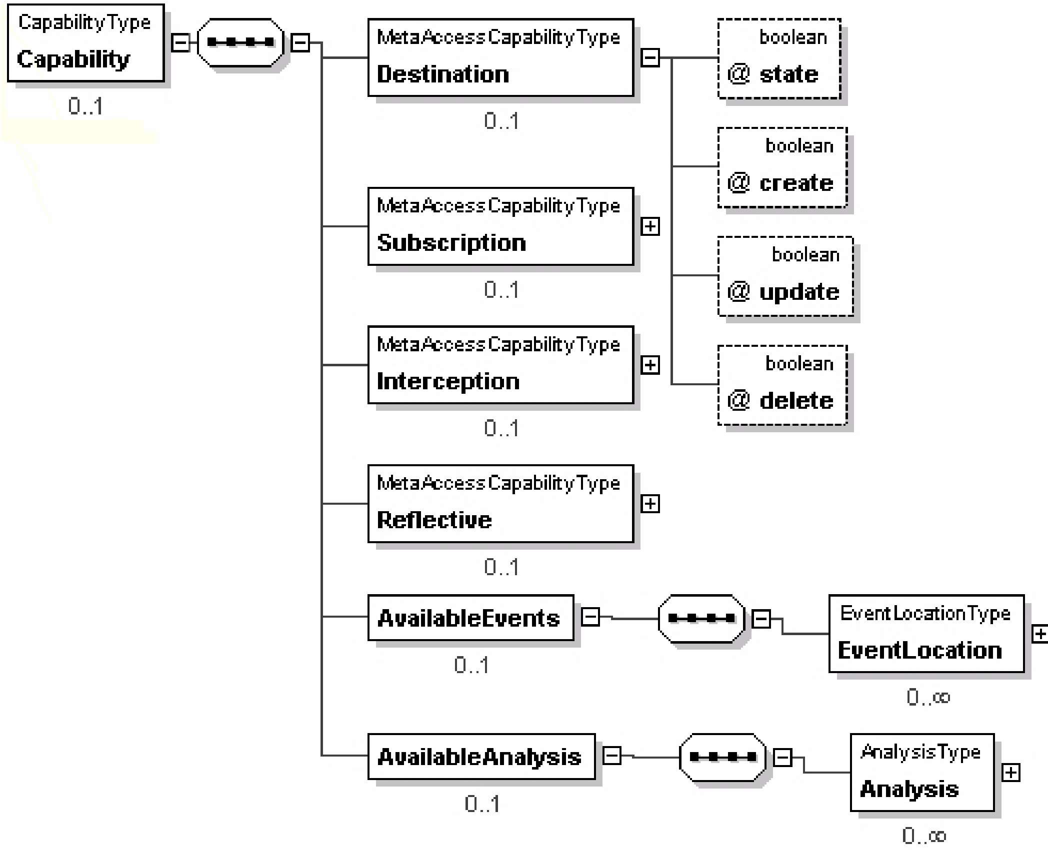


Figure 5.13 *MOM-DSL capability reply structure*

5.6 Example Interactions

The objective of this section is to provide some sample interactions using the MOM-DSL to demonstrate how it can facilitate coordination between self-managed systems. The examples also illustrate how the MOM-DSL expresses the GISMO meta-level and implements the OMIP. This section illustrates four interactions including state, update, and analysis requests. The first interaction examined is a capability request.

5.6.1 Request Capabilities

A capability request, see Table 5.10, within the MOM-DSL is a simple request command that includes an optional username and password for the requester.

```
<MOM-DSL>
  <Request requestID="1">
    <Capability-Request username="edcurry" password="rover">
  </Request>
</MOM-DSL>
```

Table 5.10 *Example MOM-DSL capability request*

The reply to the capability request, see Table 5.11, details the access available to requester. The reply details access to each of the four meta-models, details on any events that the requester has access to and a list of available meta-analyses.

```
<MOM-DSL>
  <Reply replyID="1" response="accept">
    <Capability>
      <Destination state="true" create="true" update="true"  delete="true"/>
      <Subscription state="true" create="false" update="true"  delete="false"/>     Sub meta-space
      <Interception state="true" create="false" update="false" delete="false"/>     access control
      <Reflective state="false" create="false" update="false"  delete="false"/>
      <AvailableEvents>
        <EventLocation event="CreateDestination" location="OMIP_Events.Create_Destination"/>     Event
        <EventLocation event="ReceiveMessage" location="OMIP_Events.Receive_Message"/>           locations
        <EventLocation event="SendMessage" location="OMIP_Events.Send_Message"/>
      </AvailableEvents>
      <AvailableAnalysis>
        <Analysis analysis="DestinationFilterCount" access="true"/>
        <Analysis analysis="DestinationSearch" access="true"/>      Available analyses
        <Analysis analysis="SubscriptionCount" access="false"/>
      </AvailableAnalysis>
    </Capability>
  </Reply>
</MOM-DSL>
```

Table 5.11 *Example MOM-DSL capability reply*

5.6.2 Request Destination State

The state request command within the MOM-DSL, Table 5.12, is a simple interaction command with a single attribute to identify the state requested.

```
<MOM-DSL>
  <Request requestID="1">
    <State-Request state="Destination"/>
  </Request>
</MOM-DSL>
```

Table 5.12 *Example MOM-DSL destination state request*

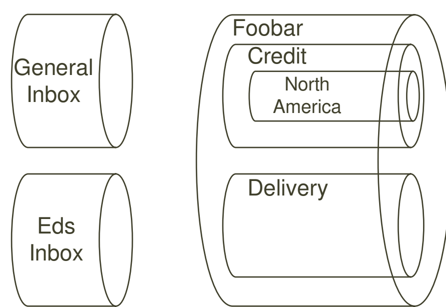


Figure 5.14 *Sample destinations used in state request*

The reply to the state request command, if accepted, returns the relevant meta-state model containing a straightforward description of the GISMO state objects expressed within XML. The reply in Table 5.13 details the sample destination state, illustrated in Figure 5.14, containing two queues and a single four-node destination hierarchy.

```
<MOM-DSL>
 <Reply replyID="1" response="accept">
  <DestinationState>
   <Single_Destinations>
    <Destination id="1" name="General_Inbox" type="queue"/>
    <Destination id="2" name="Eds_Inbox" type="queue">
     <Condition attribute="Recipient" operator="=" value="Ed"/>
    </Destination>
   </Single_Destinations>
   <Hierarchys>
    <DestinationHierarchy hierarchyID="foobar_company" root="foobar">
     <HierarchyNode id="foobar" name="foobar">
      <ChildNode childID="credit"/>
      <ChildNode childID="delivery"/>
     </HierarchyNode>
     <HierarchyNode id="credit" name="credit" parentNode="foobar">
      <Condition attribute="department" operator="=" value="credit"/>
      <ChildNode childID="north-america"/>
     </HierarchyNode>
     <HierarchyNode id="north-america" name="america" parentNode="credit">
      <Condition attribute="region" operator="=" value="north-america"/>
     </HierarchyNode>
     <HierarchyNode id="delivery" name="delivery" parentNode="foobar">
      <Condition attribute="department" operator="=" value="delivery"/>
     </HierarchyNode>
    </DestinationHierarchy>
   </Hierarchys>
  </DestinationState>
 </Reply>
</MOM-DSL>
```

Standalone destinations

Hierarchy root node

Child nodes

Hierarchy

Table 5.13 *Example MOM-DSL destination state reply*

5.6.3 Update Destination

The update request command is used to update state within the meta-level. The sample update command provided in Table 5.14 describes a request to update a destination by altering the name

and adding an extra condition to the destination. Each of the realisation commands (create, update, delete) work on the principle of using nested content to specify the nature of the request.

```
<MOM-DSL>
 <Request requestID="1">
  <Update>
   <UpdateDestination>
    <CurrentDestination id="2" name="Eds_Inbox" type="queue">
     <Condition attribute="Recipient" operator="=" value="Ed"/>
    </CurrentDestination>
    <NewDestination id="2" name="Eds_XML_Inbox" type="queue">
     <Condition attribute="Recipient" operator="=" value="Ed"/>
     <Condition attribute="Format" operator="=" value="XML"/>
    </NewDestination>
   </UpdateDestination>
  </Update>
 </Request>
</MOM-DSL>
```

Current configuration

New configuration

Table 5.14 *Example MOM-DSL destination update request*

The reply to this request, see Table 5.15, is a simple accept/reject response informing the requester as to the result of its request. In this case, the request was accepted.

```
<MOM-DSL>
 <Reply replyID="1" response="accept"/>
</MOM-DSL
```

Table 5.15 *Example MOM-DSL destination update reply*

5.6.4 Request Filter Analysis

The final interaction covered is a request for an analysis. In this request, see Table 5.16, the analysis interaction command is used to request the result of an analysis specified in the capability replay. The analysis requested counts the number and type of filters used by subscribers on a specific destination. The request must specify the destination to analysis.

```
<MOM-DSL>
 <Request requestID="1">
  <Analysis>
   <DestinationFilterCount destinationName="FooBar"/>
  </Analysis>
 </Request>
</MOM-DSL>
```

Table 5.16 *Example MOM-DSL filter analysis request*

The reply to this request, if accepted, returns the results of the analysis. In the example presented in Table 5.17, three filters have been returned for the "FooBar" destination, 15 filters compared the *Region* attribute to 'Europe', 10 filters compared the Department attribute to 'Accounts' and 5 filters checked the Priority attribute for a value of greater than '7'.

```
<MOM-DSL>
  <Reply replyID="1" response="accept">
    <AnalysisResult>
      <DestinationFilterCountResults>
        <Filter attribute="Region" operator="=" value="Europe" count="15"/>
        <Filter attribute="Department" operator="=" value="Accounts" count="10"/>
        <Filter attribute="Priority" operator=">" value="7" count="5"/>
      </DestinationFilterCountResults>
    </AnalysisResult>
  </Reply>
</MOM-DSL>
```

Analysis results

Table 5.17 *Example MOM-DSL filter analysis reply*

5.7 Summary

This chapter has shown a design for a general-purpose meta-level for MOM platforms. The central contribution of this work is the GenerIc Self-management for Message-Oriented middleware (GISMO). GISMO has been defined to provide a generic meta-level that can be used with a number of proprietary MOM implementations. The meta-level was designed by identifying generic elements, such as behaviour and state, common to MOM implementations. Portability of the meta-level across proprietary MOMs is achieved with the use of the M-SAR design pattern that also improves encapsulation and concern separation with a meta-level.

The GISMO meta-level is comprised of destination, subscription, and interception meta-models. The meta-level also includes a reflective engine, with pluggable reflective policies, and support for event notifications. GISMO is an open meta-level and provides full support for the Open Meta-level Interaction Protocol (OMIP) through the MOM-Domain Specific Language. The MOM-DSL is expressed using the eXtensible Markup Language and provides access to all aspects of the GISMO meta-level.

Chapter 6

Implementation of a GISMO

With the design for a generic MOM meta-level in place, the next step is to provide a concrete implementation of GISMO. This chapter describes an implementation of the GISMO meta-level using the Chameleon framework to augment the meta-level to multiple MOM base-levels in a non-invasive manner.

6.1 Introduction

A number of challenges exist when attempting to add a portable meta-level to a 3rd party base-level implementation. This chapter illustrates how to implement a portable meta-level with the use of the Chameleon framework. The Chameleon framework and the techniques it uses to augment functionality in a non-invasive manner are described. An overview of the realisation of the GISMO meta-level using Chameleon is discussed along with detailed highlights of the novel aspects of the implementation. The discussion commences with a review of the challenges faced with the transparent augmentation of a meta-level to an underlying base-level in a non-invasive manner.

6.2 Challenges in System Extension

The process of adding a meta-level to an existing MOM implementation requires the extension of the MOMs functionality and observation of its behaviour. This task presents many challenges and a number of techniques exist for the extension of a software system. However, the meta-level needs to be augmented in a non-invasive manner for it to be portable between MOMs. With this in mind, the evaluation of potential extension mechanisms needs to consider the invasiveness of the technique. Does it require source code? Will each change require recompilation and re-deployment? How flexible is the technique? How safe is it? The remainder of this section introduces a number of possible techniques for extension and provides a critical evaluation of their suitability for transparent functionality augmentation.

6.3 Options for Extension

The ever-increasing demands placed on software systems require them to often perform beyond the scope of their original requirements. Such behaviour may not always be anticipated during

their initial development phase, thus making it important to design systems that can be easily extended during their life-cycle. In order to tackle these challenges a number of software engineering techniques have emerged to allow the extension of a software system to function beyond its original design. These software engineering techniques also minimise the runtime of systems designed for large target audiences with diverse interests. Such systems often include functionality that is only utilised by a small percentage of users. While such functionality may be vital to some users, incorporating it into the core system makes it unnecessarily bloated and increases overhead for the majority of remaining users. In this section, the interceptor design pattern, aspect-orientated programming, programmatic reflection, and generative programming are evaluated for their suitability to transparently augment a meta-level.

6.3.1 Interceptor Design Pattern

The Pattern-Oriented Software Architecture (POSA) *Interceptor* design pattern [112] is a variant of the Chain of Responsibility pattern from the Gang of Four (GoF) [113] . This pattern enhances a system by increasing flexibility and extensibility. The pattern also enables functionality to be easily added to the system to dynamically change its behaviour. This seamless integration of functionality can be performed without the need to stop and recompile the system, allowing its introduction at runtime.

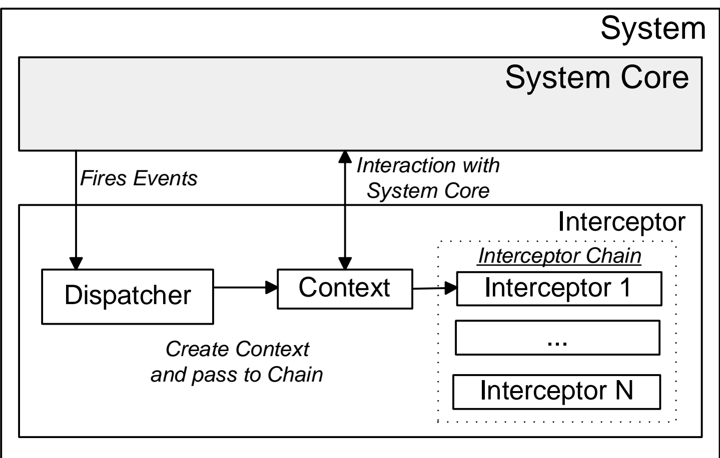


Figure 6.1 *The POSA interceptor design pattern*

The basic POSA interceptor pattern has four main elements:

- System Core
- Dispatcher
- Context
- Interceptors

The Interceptor pattern, illustrated in Figure 6.1, follows a straightforward sequence of events. Interceptors are registered with the system dispatcher; the system core may perform registration, the interceptors may self-register, or they may be registered from an external source (parent

server/master server). Once the interceptors are registered, the system core notifies the dispatcher of any events that have occurred. Upon receiving an event, the dispatcher examines the event to determine which interceptors need to be notified. The dispatcher then packages the event and any relevant information into a context provided by the system core. The dispatcher then notifies the relevant interceptors or interceptor chains (a ordered/unordered collection of interceptors) by passing them the context containing the event. When triggered, the interceptor examines the context and executes its related functionality. An optional addition to the pattern allows interceptors access to the internals of the core system state and provides a mechanism to control the system by altering its state.

Interceptors are utilised in a broad range of domains to increase flexibility and extensibility; such systems include CORBA ORBs (TAO, Orbix) for infrastructure and support services, web browsers (Microsoft Internet Explorer) for plug-in integration, and web servers (Apache 2.0) to allow modules to register handlers (interceptors) with the core server. The JBoss J2EE [114] application server also uses the interceptor design pattern to provide customised functionality in areas such as transactions, security, remoting, and life-cycle support.

6.3.2 Aspect-Oriented Programming (AOP)

The emergence of multifaceted software paradigms such as Aspect-Oriented Programming (AOP) and Multi-Dimensional Separation of Concerns (MDSOC) will have a profound effect on software construction. This section provides a brief overview of these new programming techniques.

A complex software system can be viewed as a combined implementation of multiple concerns, including business-logic, performance, logging, data and state persistence, debugging and unit tests, error checking, multithreaded safety, security and various other concerns. Most of these are system-wide concerns and are implemented throughout the entire codebase of the system, these system-wide concerns are known as crosscutting concerns.

One of the predominant techniques for implementing software systems is the Object-Oriented (OO) paradigm. The OO paradigm is a major advancement in the way developers think of and build software, but it is not a silver bullet and has a number of limitations. One of these limitations is the inadequate support for crosscutting concerns. The Aspect-Oriented-Programming (AOP) [43] methodology helps overcome this limitation. AOP complements OO by creating another form of separation that allows the implementation of a crosscutting concern as a single unit. With this new method of concern separation, known as an aspect, crosscutting concerns are more straightforward to implement. Aspects can be changed, removed or inserted into a systems codebase enabling the reusability of crosscutting code.

A brief illustration would be useful to explain the concept. The most commonly used example of a crosscutting concern is that of logging or execution tracking, this type of functionality is implemented throughout the entire codebase of an application making it difficult to change and maintain. AOP [43] allows this functionality to be implemented in a single aspect; this aspect can now be applied/weaved throughout the entire codebase to achieve the required functionality.

One of the founding works on AOP highlighted the process of performance optimisation that bloated a 768-line program to 35,213 lines. Rewriting the program with the use of AOP techniques reduced the code back to 1,039 lines while retaining most of the performance benefits. Grady Booch, while discussing the future of software engineering techniques, predicts the rise of multi-

faceted software i.e. software that can be composed in multiple ways at once. Booch cites AOP as one of the first techniques to facilitate a multifaceted capability [115].

6.3.3 Dynamic AOP for Reflective Middleware

The OO paradigm is widely used within the base-level of reflective platforms. However, a clearer separation of crosscutting concerns would be of benefit to reflective architectures. This provides the incentive to utilise AOP within reflective middleware platforms.

A major impediment to the use of AOP techniques within reflective systems has been the implementation techniques used by the initial incarnations of AOP [116]. Traditionally, when an aspect is inserted into an object, the compiler weaves the aspect into the objects code; this results in the absorption of the aspects into the objects runtime code. The lack of preservation of the aspect as an identifiable runtime entity is a hindrance to the dynamic adaptive capabilities of systems created using aspects. Workarounds to this problem exist in the form of dynamic system recompilation at runtime and load-time weaving, however this is not an ideal solution, and a number of issues, such as the transference of the system state, pose problems.

Alternative implementations of AOP have emerged which do not have this limitation. These approaches propose a method of middleware construction using composition filters [44] to preserve aspect as runtime entities. This method of creation facilitates the application of AOP for the construction of reflective middleware platforms. Another approach involving Java bytecode manipulation libraries such as Javassist provide a promising method of implementing AOP frameworks (JBossAOP) with dynamic runtime aspect weaving.

6.3.4 Multi-Dimensional Separation of Concerns

The key difference between dynamic AOP and MDSOC is the scale of multifaceted capabilities. AOP will allow multiple crosscutting aspects to be weaved into a program thus changing its composition through the addition of these aspects. Unlike AOP, MDSOCs multifaceted capabilities are not limited to the use of aspects; MDSOC allows for the entire codebase to be multifaceted enabling the construction of the software in multiple dimensions.

MDSOC also supports the separation of concerns for a single model [117], when using AOP you start with a base and use individually coded aspects to augment this base. Working from a specific base makes the development of the aspects more straightforward but also introduces limitations such as restrictions on aspect composition [117]; you can't have an aspect of an aspect. In addition, aspects can be tightly coupled to the codebase for which they are designed, this limits their reusability.

MDSOC enables software engineers to construct a collection of separate models, each encapsulating a concern within a class hierarchy specifically designed for that concern [117]. Each model can be understood in isolation, any model can be augmented in isolation and any model can be augmented with another model. These techniques streamline the division of goals and tasks for developers. Even with these advances, the primary benefit of MDSOC come from its ability to handle multiple decompositions of the same software simultaneously, some developers can work with classes, others with features, others with business rules, and others with services even though they model the system in substantially different ways [117].

To further illustrate these concepts an example is needed, a common scenario by Ossher [117] is of a software company developing personnel management systems for large international organisations. For the sake of simplicity assume their software has two areas of functionality, personal tracking which records employees personnel details such as name, address, age, and phone number, and payroll management which handles salary and tax information.

Different clients seeking similar software approach the fictitious company, they desire the software but have specific requirements, some clients want the full system while others do not want the payroll functionality and refuse to put up with the extra overhead within their system implementation.

Based on market demands the software house needs to be able to mix and match the payroll feature. It is extremely difficult to accomplish this sort of dynamic feature selection using standard object-oriented technology. MDSOC allows this flexibility to be achieved within the system using on-demand remodularisation capabilities, it also allows the personnel and payroll functionality to be developed almost entirely separate using different class-models that best suit the functionality they are implementing.

6.3.5 Programmatic Reflection

Refection capabilities are popular in programming languages, including Lisp, Smalltalk, Python, and Java. Within programming languages, reflection can offer the ability to alter the structure and behaviour of programs. The Java programming language allows a program to obtain structural information on the methods and attributes of a class using the *java.lang.reflect* package. The package also allows for classes to be dynamically loaded and for the discovery and execution of methods at runtime. One of the major drawbacks of packages like *java.lang.reflect* is the drastic difference, not only in format but also in complexity, between code used for program introspection and dynamic invocation when compared to regular code. Steve Vinoski, chief engineer of product innovation for IONA Technologies[1], sheds some light on this issue:

> To call a function with normal programming-language syntax, the developer simply writes the function name and passes arguments to it, also by name. With reflection, you must find the metadata for the function or operation to be invoked, and use it to dynamically construct appropriate values to pass as arguments. This can be arduous even for relatively simple types. Worse, you often have to perform the dynamic invocation using syntax or calls that look nothing like normal invocations. The relative complexity of reflective programming, which often results in applications consisting of one or two orders of magnitude more lines of code than similar static application code, tends to mean that only sophisticated programmers practice it. [118]

While these factors are a major drawback of using programmatic reflection, it is becoming a popular way of developing systems that need to be flexible and adaptive to meet their operating requirements.

[1] http://www.iona.com

6.3.6 Generative Programming

Generative programming [119] is the process of creating programs that construct other programs. The basic objective of a generative program, also known as a program generator [120], is to automate the tedious and error-prone tasks of programming. Given a requirements specification, a highly customised and optimised application can be automatically manufactured on demand. Program generators manufacture source code in a target language from a program specification expressed in a higher-level domain language. With the requirements of the system defined in domain language, the target language used to implement the system may be changed. The program generator could use Java, C, Visual Basic (VB), or any other language as the target language for implementation. The use of multiple program generators could offer the user a choice for the implementation of the program, such as a Java version and a C version.

Generative programming allows for high-levels of code reuse in systems that share common concepts and tasks, providing an effective method of supporting multiple variants of a program; this collection of variants is known as a program family. Program generation techniques may also be used to create systems capable of adaptive behaviour via program recompilation.

6.3.7 Evaluation

The main object of the Chameleon framework is to augment the base functionality of an underlying message provider in a non-invasive manner. The method used to achieve this needs to be flexible, dynamic, and capable of runtime changes. Moreover, the implementation must be straightforward to develop and intuitive to understand.

Each of the techniques has been chosen from a larger pool of candidates based on their established record of accomplishment within production environments. All the techniques have their own strengths and weaknesses depending on the requirements of a given situation. Two main requirements exist for Chameleon:

- Does it require source code?
- Can it perform runtime changes without the need for a recompile?

A summary of each technique's ability to meet these criteria is presented in Table 6.1.

Technique	**Require Source Code**	**Runtime Change Requires Recompile**
Interceptor Design Pattern	No	No
Aspect-Oriented Programming	Yes	Partial (Newer frameworks allow load-time weaving)
Programmatic Reflection	Yes	No
Generative Programming	Yes	Yes

Table 6.1 *Summary of extension techniques*

AOP, programmatic reflection, and the interceptor design pattern can all perform dynamic runtime changes without the need for a system recompile. Such a capability is vital to the operation of a framework like Chameleon. Further examination of these techniques reveal that programmatic

reflection requires possible changes to be expressed in the compiled code. This reduces the ability of the code to perform unanticipated changes. These unforeseen adaptations would require the code to be altered and recompiled, thus limiting its adaptive capability. Vinoski [118] suggests a potential, but limited, solution to this problem with the use of a "simple - although not quite uniform - service interface". Essentially, he proposes that any object introduced at runtime would need to conform to a specified interface, thus removing the need for code changes and recompilation. The two remaining techniques allow runtime adaptations with the use of a runtime/load-time weaver for AOP or the addition of a single or multiple interceptors in the Interceptor design pattern.

The portability of any framework that is designed to work on a specific non-standardised codebase would require the framework to be customised to each specific codebase, thus negating the portability benefits gained through use of the JMS specification. In addition, only a handful of the MOM providers available are open-sourced. Any framework that requires intimate knowledge of a providers code base would reduce its potential user-base considerably. These considerations lead to the second major requirement for the Chameleon framework, the ability to extend a platform without the need for access to the platform's source code. Out of the reviewed approaches, the Interceptor design pattern is the only technique that does not require the source code of the underlying MOM provider.

Based on this evaluation process the interception design pattern is the most suited approach to meet the requirements of the Chameleon framework. AOP was also given serious consideration in the final decision but the code-centric nature of the approach was deemed less suitable for this project. As previously noted, a major drawback of programmatic reflection is code bloat that "results in applications consisting of one or two orders of magnitude more lines of code" [118]. This made the approach unsuitable for a lightweight framework placed upon an underlying platform. Generative Programming, while very useful in creating adaptive and flexible applications, is code and complication centric; such characteristics diminish its suitability for the framework.

The interceptor design pattern was chosen as the method of extension. However, the pattern is not a silver bullet to the problem. The interceptor pattern has a number of advantages and disadvantages. Benefits of the pattern include the decoupling of communications between a sender and receiver of an interceptor request; this permits any interceptor to fulfil the request and allows interceptors to change system functionality, even at runtime.

The pattern also has a number of drawbacks that if left unresolved may lead to a number of issues in the system design. One of the main drawbacks is increased complexity in design, the more interceptors can hook into the system the more bloated its interface becomes. The inherent openness of the pattern also introduces potential vulnerabilities into systems. With such an open design, the introduction of malicious or erroneous interceptors may result in system error or corruption.

Another important issue to consider is the possibility of incompatibilities between interceptors and potential infinite interceptor loops whereby an event produced by an interceptor triggers another interceptor that in turn generates an event that triggers the original interceptor. Such errors will only occur at runtime and may be difficult to locate.

When used within the messaging domain the abstract generic interceptor pattern is implemented using a customised context and interceptor, these are commonly referred to as *'Message Context'* and '*Message Handler'* respectively.

6.4 Chameleon

The goal of the Chameleon framework is to allow the transparent extension of functionality onto a base MOM platform. Towards this objective, the Java Message Service (JMS) Application Protocol Interface (API) [78] is used as an interface to the underlying core messaging platform. MOM services/features are then packaged as interceptors or message handlers and deployed to add functionality on top of the base service, enhancing its functionality. The remainder of this section examines the architecture of Chameleon and discusses its ability to extend functionality on both the client-side and provider-side of a MOM platform.

6.4.1 Call Capture Proxy

The Chameleon framework will require a mechanism to capture the actions and events that occur within the underlying MOM implementation. These events and behaviour include messaging and administration activity. To clearly define the challenge faced by this mechanism, a simple messaging interaction is presented in Figure 6.2. Within the call sequence, a message is produced and consumed.

Figure 6.2 *MOM call sequence*

The objective of the framework is to receive notifications of the actions within the call sequence and extend the functionality.

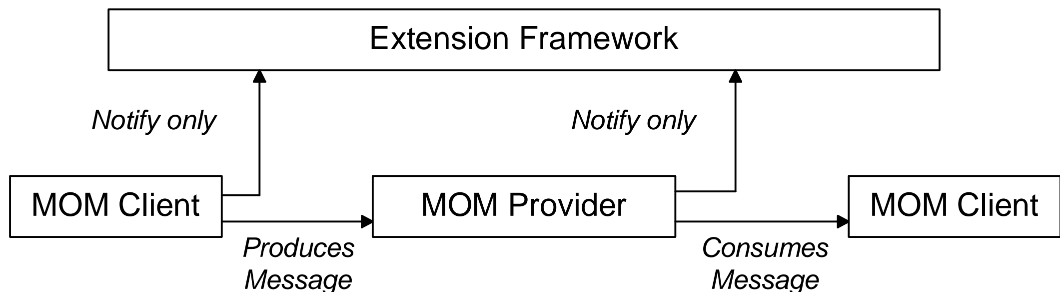


Figure 6.3 *Notification call sequence*

Receiving notifications from the MOM is illustrated in Figure 6.3. Within this call sequence, the client delivers its message directly to the MOM provider and sends a notification to the extension framework informing it of its action. A similar notification is sent to the framework when the message is consumed.

The process of intercepting and extending the functionality of the call sequence is illustrated in Figure 6.4. Within this sequence, the produced message is intercepted and redirected to the extension framework where functionality may be injected. Once completed, the message is forwarded onto the message provider. Message consumption follows a similar process. The challenge faced

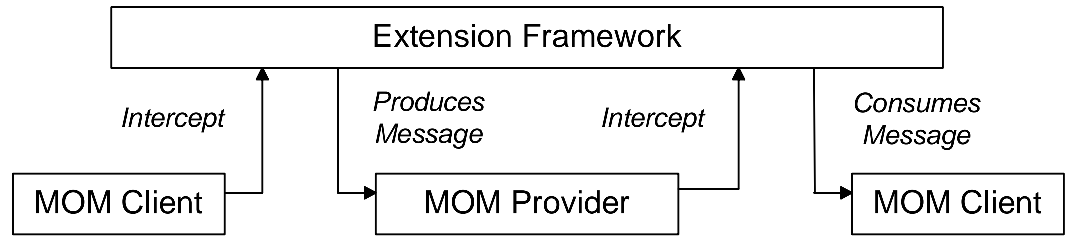


Figure 6.4 *Interception call sequence*

with these tasks is to facilitate them in a transparent manner from the perspective of the MOM clients while maintaining portability across MOM implementations.

Given the large number of proprietary MOM implementations in existence and the desire for Chameleon to be compatible with as many as possible, a generic solution to this problem is vital to support portability. Fortunately, the JMS specification is widely accepted as an industrial standard, providing an ideal solution for capturing MOM interaction. The JMS API provides a universal interface to proprietary MOM implementations. A MOM provider's client-side library implements the JMS API. The position of the API within the call sequence is illustrated within Figure 6.5.

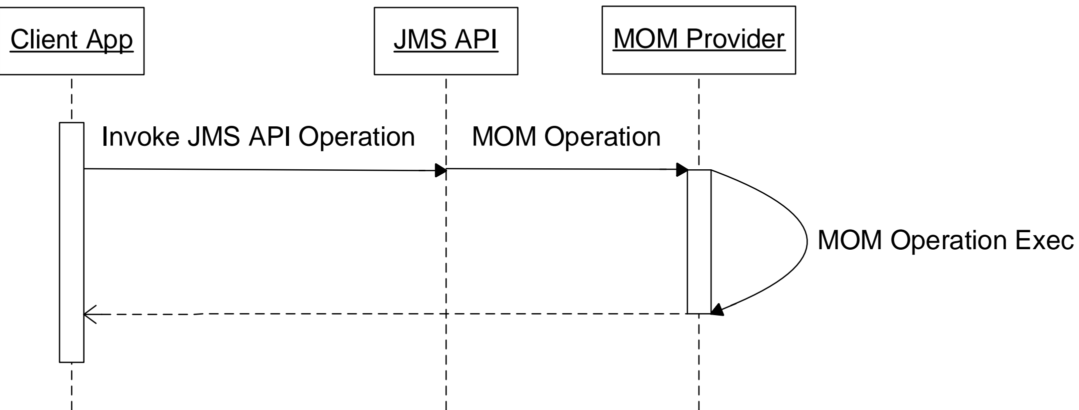


Figure 6.5 *JMS API client/provider interaction sequence*

In effect, the API acts as an intermediary between the client and the provider, decoupling a client application from the MOM. This allows the provider to be changed without affecting the client application. This portability presents an opportunity to introduce an additional intermediary into the call sequence to capture client-provider interaction. If this intermediary or proxy also implements the JMS API, it can seamlessly replace the MOM provider's client-side library from the client's perspective. Once in place the proxy can seamlessly intercept client-provider calls, send notification, or inject functionality before forwarding the request onto the provider's client-side library. The proxy enhanced call sequence is illustrated in Figure 6.6.

The proxy allows for the actions of the client to be seamlessly captured and conveyed to Chameleon in a non-invasion transparent manner. With the capability to capture MOM interactions in place, the next step involves extending functionality once a call is captured.

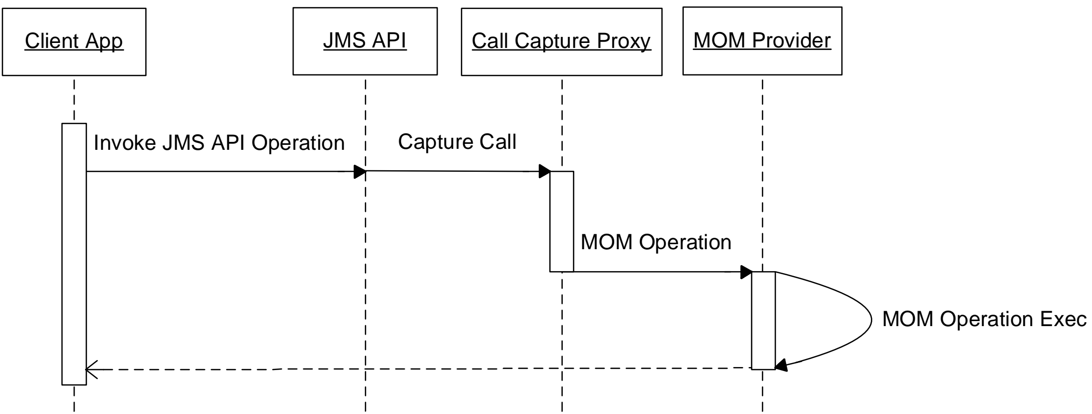


Figure 6.6 *JMS API client/provider captured interaction sequence*

6.4.2 Interception

Once a call has been captured, for functionality to be extended, a mechanism must be in place to inject this functionality to alter the system's runtime operation. In order to achieve this, the Chameleon framework provides a full-scale interception mechanism. The architecture of the interception framework is illustrated in Figure 6.7. The requirements set out for GISMO interception capabilities, detailed in Chapter 5, define a powerful general-purpose interception mechanism for systems within the MOM domain. The requirements, including global and local interception with support for destination scopes, are incorporated within the design of the interception mechanism within Chameleon. The implementation of this mechanism covers the following areas, server-side interception, client-side interception, and the context used to pass information between both client- and server-side interceptors and mobile interceptors. The remainder of this section describes a high-level overview of the operation of these areas.

6.4.2.1 Server-side

The core of the Chameleon framework exists in its server-side deployment. When the framework initialises, it first registers any handler chains present in its start-up configuration. When a message is intercepted, Chameleon first checks for the existence of a client-side message context. If one exists it uses it as the basis to create a server-side context. Alternatively, it creates a new context for the inbound message. Once the context is ready, it is passed to any relevant global server-side chains. After the global dispatcher has completed evaluating the message it is passed on to the local-chain dispatcher. This local-dispatcher is responsible for triggering any local-chains associated with specific destinations.

6.4.2.2 Message context

The message context is an interceptor's main point of interaction with the framework. Message contexts act as a medium to store data, to communicate with other handlers, or to interact with the framework. Handlers are able to store and retrieve information within the context using the *setProperty()* and *getProperty()* methods. These methods can save any serializable object and the basic Java primitive types within the context.

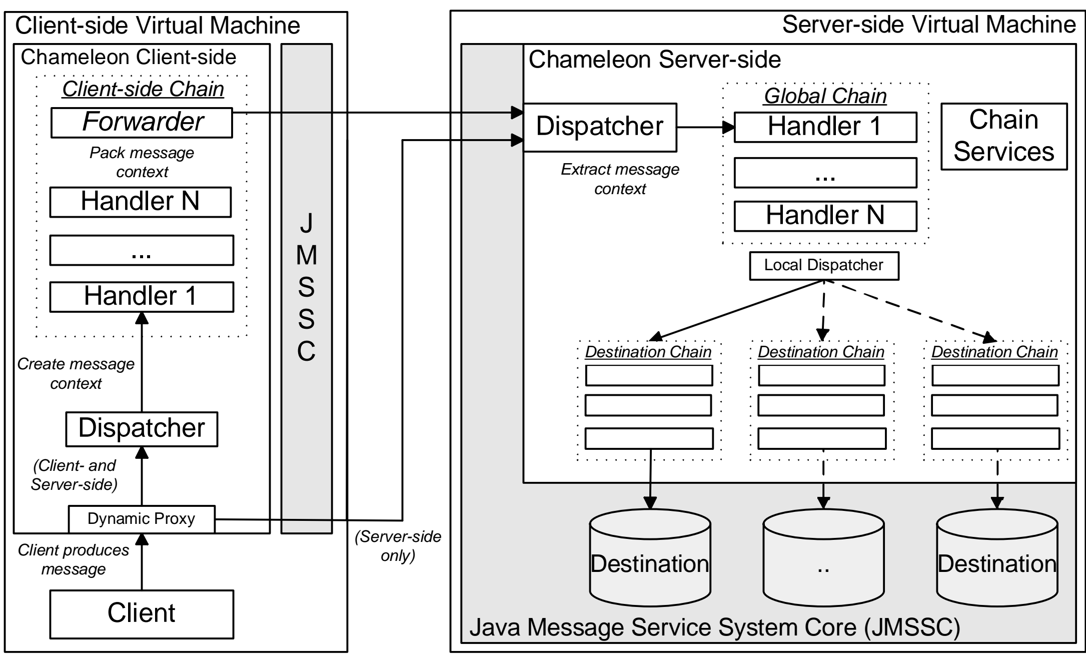


Figure 6.7 *Chameleon interception architecture*

Contexts can communicate information between client-side and server-side handlers. This allows for the behaviour on the server-side or client-side to be dynamically altered based on events and information from either side.

6.4.2.3 Client-side

Client-side handlers offer the ability to extended and enhance the functionality of the MOM provider on the client. Client-side chains allow the message service to dynamically alter the behaviour of its client at runtime. This capability has a number of advantages by easily distributing computational tasks and behaviour to client machines. With this support framework in place, advanced features may be developed with cooperation and coordination between both the client and server-side of the platform. Such capabilities can increase the scalability of centralised deployments by distributing tasks to the clients, such as message transformation or filtering.

The operation of the client-side interception framework is as follows. Once client-side chain initialisation has completed, the outbound/inbound message is placed into a message context and passed to the client-side dispatcher. As soon as the message has passed through the chain, the message context is packaged into the JMS messages and sent to the server-side. Upon receipt of a new message, the server checks for the existence of a client-side context and uses this in the construction of the server-side context. This process allows client-side handlers to communicate with the server-side, allowing data exchange between them, such as the results of a distributed task or the results of any computations or pre-processing carried out on the message by the client-side.

The inverse of this process is also possible, whereby server-side handlers wish to pass message specific information to the client-side.

6.4.2.4 Mobile Chains

Client-side handlers can be constructed in two manners, the first approach is to build the chain from local handlers which reside on the client machine, this approach requires the machine to have the relevant handlers installed or the use of a distributed classloader. The second approach, known as mobile chains, involves the construction of chains on the server-side and transferring it to the client; this removes the need to have application-specific stubs that need to be pre-installed on the client machine.

The dynamic retrieval and configuration of client-side handlers has a number of benefits; the deployment of services to clients can now take place without any special arrangements on the client-side. This streamlined distribution of services reduces the amount of administration needed to alter a deployed system, making frequent changes to its behaviour and configuration more feasible. A service can adapt itself into a more optimal state based on its current operating conditions. Clients may now connect to multiple servers and retrieve their specific client-side chain without the need for extra configuration or intervention by a system administration.

The major benefit of mobile chains is the ability for a simple clean base install of Chameleon to connect to a previously unknown message broker and download the related code needed to interact with any non-standardised parts of the broker. To illustrate this ability imagine that a service requires messages submitted to be in a specific format or a specific categorisation was used to sort and store messages on the service. Using Chameleon, message transformation or sorting logic can be autonomously sent to the client to allow seamless interaction with the service.

The prospect of deploying functionality to the client-side is an interesting proposition, however due diligence must be taken when considering the use of this “mobile code”. If configuration of a client-side chain is directed from a remote location or if the configuration is downloaded, it presents a number of security issues and potential vulnerabilities to the client system. Such issues are covered in more details in [121].

6.4.3 Non-Invasive Extension of Functionality

In order to demonstrate the extension capabilities of the Chameleon framework, a small case study is provided to illustrate how functionality may be added to a MOM provider. When choosing a candidate for this study, a number of possibilities were considered including, message routing and filtering, message security, and message encryption. In the end, a decision was taken to package enhanced message transformation capabilities, and to allow such transformation to be performed on the clients-side (i.e. using XSL/T and XPath filters to transform XML payloads).

Message transformation takes place when two disparate systems need to interconnect; such requirements are commonplace within Enterprise Application Integration (EAI) environments. Tasks of this nature are typically performed by integration platforms, such as IONA’s ARTiX, Microsoft’s BizTalk Server, and SAP’s NetWeaver. In this case study, the motivational scenario is that of an information service which accepts information from message producers in a single format, *Format A*, and provides this information to message consumers in a number of formats, *Formats A, B and C*. In a conventional deployment, the information service upon receiving a message in format A would be required to transform the message into the two alternative formats (formats B and C) for consumers. This process is illustrated in Figure 6.8.

Depending on the amount of work required to perform the transformation, this centralised

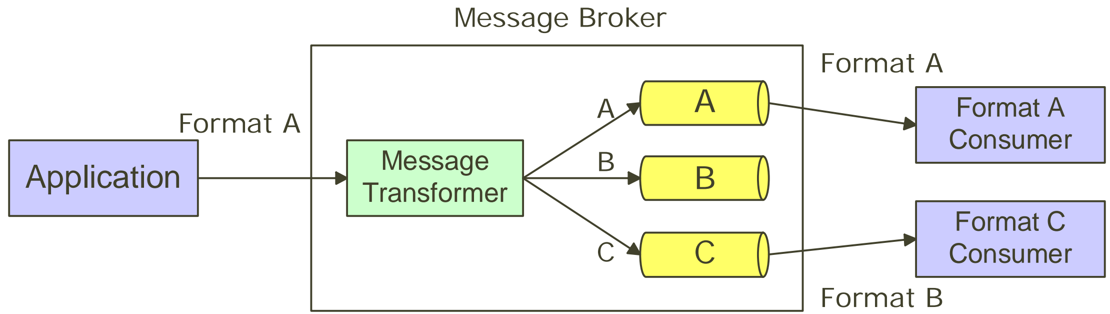


Figure 6.8 *Centralised message transformation*

approach could have limited and expensive scalability. With the use of Chameleon, it is possible to decentralise the message transformation task among the participating clients. This is achieved by intercepting the message send action from the message producer and performing the two message transformations before the message is sent. Once the transformations are complete three messages, one in each format, would be sent to the information service.

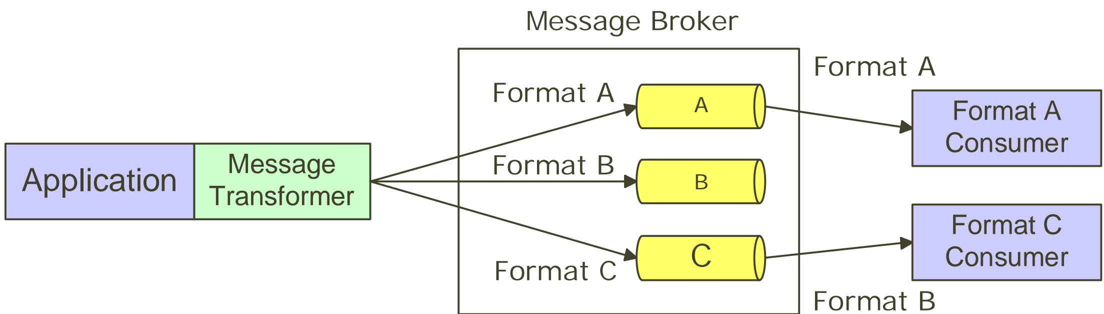


Figure 6.9 *Decentralised message transformation*

The decentralised message transformation service[1] is illustrated in Figure 6.9. Within this service the message send action for *Format A* messages is intercepted by a dynamic proxy and passed to client-side chains. Message transformation handlers for format B and C clone the message and transform the clone into their respective formats. Messages in formats A, B, and C are sent to the information service

This trivial example illustrates the simplicity of the Chameleon mechanism for functionality extension and the ease of its deployment. Once the Chameleon infrastructure is in place, the message transformation capability can be deployed on-top of any JMS compatible MOM provider. This capability is of particular use to the academic community as it provides a platform for academia to expose their MOM related work to a wider audience. This concept is demonstrated in the Chapter 8 case study where the techniques of covering and merging from the SIENA [67] research project are added to a JMS compliant MOM. Within the case study, message participants choose the most

[1]It is important to note that this solution requires three messages to be sent to the MOM and is only beneficial in scenarios where the cost of message transformation (i.e. video compression) is greater that the cost of message transport.

appropriate destination for their messages using a derivative of the SIENA filtering libraries.

6.5 Realisation of a GISMO

With the Chameleon framework in place, the next step is to implement the GISMO meta-level using the framework. This process involves implementing the GISMO abstract design from Chapter 5 using the Chameleon framework. An overview of the layers within the GISMO implementation model are illustrated in Figure 6.10.

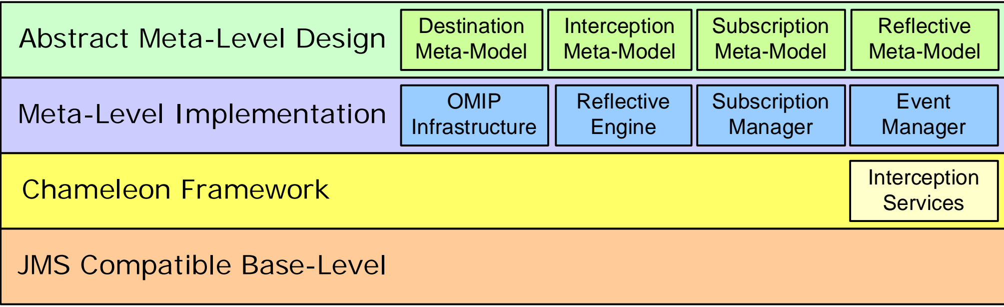


Figure 6.10 *The GISMO implementation model*

An examination of this model from the top-down shows the abstract design of the MOM meta-level that encompasses the concepts of its constituent sub meta-models. The abstract design is implemented by the three lower levels in the architecture. The next level provides concrete implementations of the concepts from abstract meta-level design including a reflective engine, subscription manager, and event manager. This layer uses Chameleon to augment itself upon a JMS compliant provider. Chameleon is capable of augmenting a JMS provider in a transparent manner by injecting new functionality with the use of interception. The bottom layer, a JMS compliant provider, is responsible for the provision of messaging services within the architecture and is the target benefactor of the three layers above it.

The remainder of this section covers the implementation of GISMO using Chameleon, starting with destination realisation. To avoid duplication with the description of the abstract design from Chapter 5 the discussion omits straightforward tasks and focuses on the novel aspects of the implementation.

6.5.1 Destination Realisation

The task of implementing the destination meta-model is for the most part a straightforward process of ratifying changes made to the meta-model in the configuration of the base-level. The most interesting part of this solution is the use of the M-SAR design pattern to streamline portability between multiple base-levels.

As identified within Chapter 5, no standardised administration interface is available for MOM platforms. To this end, the destination meta-model defines a generic administration interface that includes common administration actions. With an interface in place, the next step is to ratify these changes to the base-level.

Each base-level (MOM platform) possesses a different proprietary administrative interface, resulting in a different realisation process for each base-level. The M-SAR design pattern promotes a separation of concerns between the state, analysis and realisation concerns within a meta-level. By encapsulating the realisation process within an object, it simplifies the process of ratifying the meta-level to different base-levels. The current implementation of the destination realisation process provides support for three JMS providers, OpenJMS, ActiveMQ, and SonicMQ. The role of M-SAR in assisting this portability is illustrated in Figure 6.11.

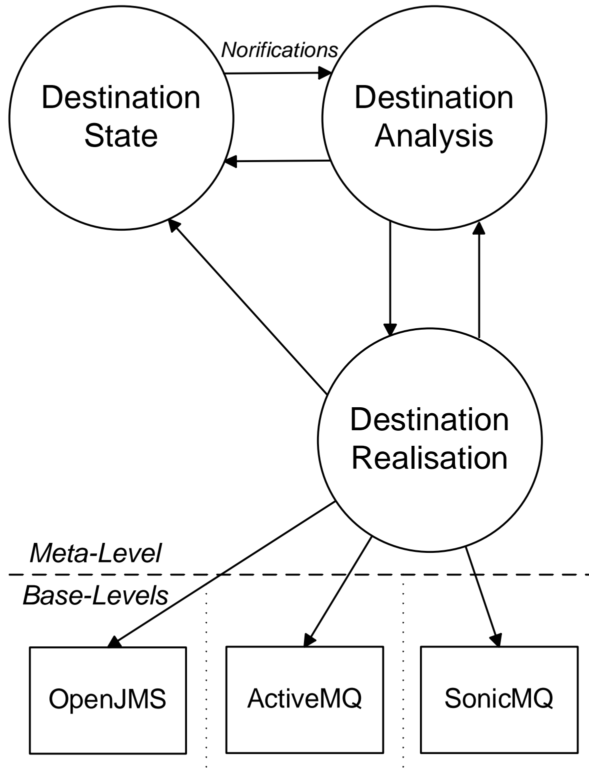


Figure 6.11 *Multiple base-level realisations*

6.5.2 Interception Provision

The next sub meta-model to be examined is the interceptions meta-model. Where a message provider does not provide native support for interception, the implementation of GISMO must fulfil this requirement. Given that very few message providers include interception support and the fact that Chameleon provides a fully functional interception mechanism, the decision was taken to assume responsibility for interception provision. With the use of Chameleon-based interception, the implementation of the interception meta-model is simply a process of ratifying the meta-model to the configuration of the Chameleon interception mechanism. Further details on the techniques used to intercept base-level interaction are provided in Section 6.4.1 with details of the interception mechanism within Chameleon available in Section 6.4.2.

6.5.3 Subscription

An interesting aspect of the subscription meta-model, illustrated in Figure 6.12, is its realisation process. Only the client owning the subscription may change it. In order for the server to change a subscription, it must request the client to perform the realisation using the OMIP. Given the potential for a large quantity of subscriptions, the implementation of the subscription meta-level uses a database to store subscription state. The information analysis and search capabilities of a database also provide a useful mechanism to simplify meta-analysis.

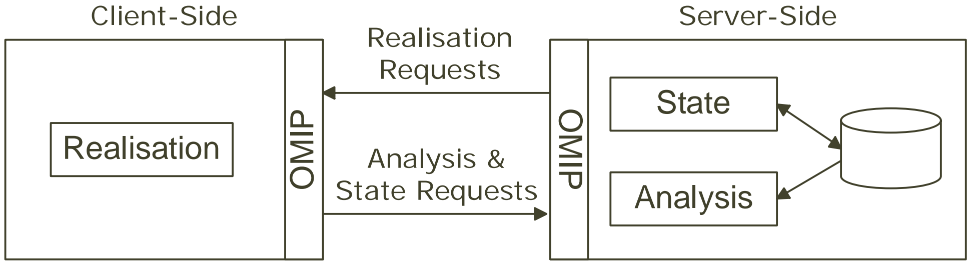


Figure 6.12 *Subscription meta-model architecture*

6.5.4 Reflective Engine

Within the reflective engine, illustrated in Figure 6.13, reflective computational logic is encapsulated within pluggable reflective policies. These policies contain the reflective logic that direct change within the meta-level. Reflective policies are stored within a repository and are created and maintained by the policy manager. The policy manager is responsible for the loading and unloading of policies from the policy container

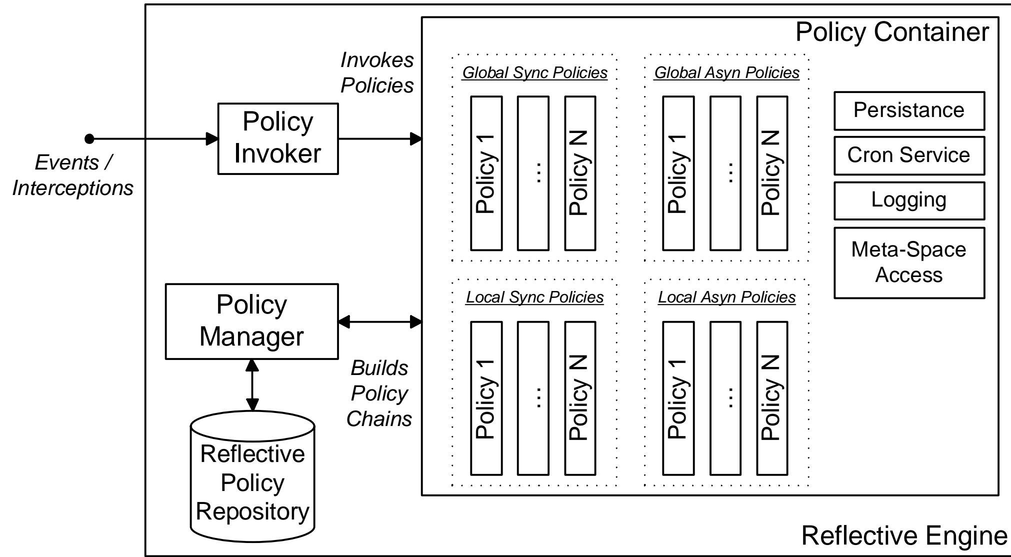


Figure 6.13 *GISMO reflective engine architecture*

Once set up, policies are executed by the *Policy Invoker*. The invoker receives notification and

interceptions from the base-level and executes the corresponding reflective policies. Within the reflective engine, a number of policy support services are provided, including persistence, logging, and cron triggers. Policies also have full access to the GISMO meta-level to perform inspections and adaptations. With a high-level overview of the reflective architecture in place, the next section provides a description of the structure of reflective policies.

Triggers for reflective policies are attained by placing interceptors at each interception point within Chameleon to supply trigger information to the reflective engine. These interceptors inform the *PolicyInvoker* within the reflective engine of the activity of the base-level. The PolicyInvoker has responsibility for invoking reflective policies and facilitating policy execution synchronously or asynchronously with base-level execution.

6.5.5 Event System

The implementation of the GISMO event system, illustrated in Figure 6.14, is similar to that of the reflective engine. An interceptor is attached at each interception location and is responsible for informing the event system of the activities of the base-level. The event engine is responsible for maintaining a list of subscribers and the events in which they are interested. Upon receiving a report from an interceptor, it is the responsibility of the event engine to inform the relevant local subscribers of the event. In addition, the engine is also responsible for publishing the events to the relevant destinations used by remote OMIP event listeners.

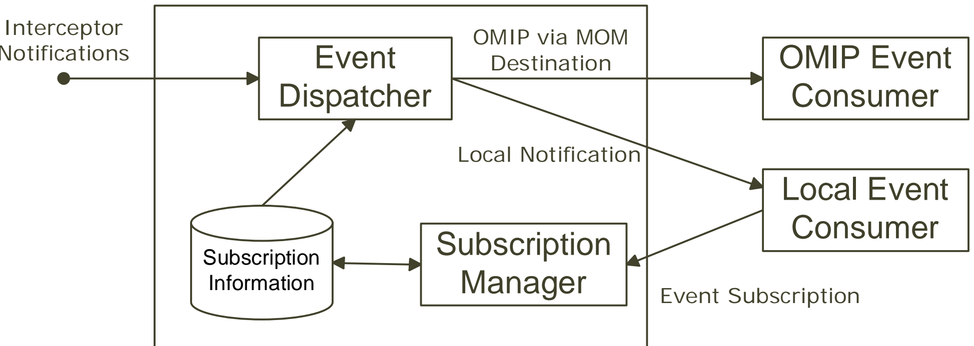


Figure 6.14 *Event system architecture*

6.5.6 OMIP Infrastructure

The infrastructure used to handle OMIP interactions follows the ARMAdA architecture, described in Chapter 4, using an inbox listener to process incoming MOM-DSL requests. Once a message is received, the request is evaluated and appropriate action is taken within the meta-level. The Facade design pattern can be used to encapsulate OMIP interactions to simplify the process of creating, sending, and receiving requests and responses. The facade is useful for both the GISMO client and provider meta-levels. Chapter 5 provides further detail on the message format and infrastructure used to exchange OMIP messages.

6.6 Summary

The Chameleon framework is designed to support the use of message handlers (interceptors) with a JMS compatible MOM platform. Chameleon provides a non-invasive message-centric method for augmenting any JMS compatible message service without needing access to its source code for mass-refactoring and recompilation.

The Chameleon framework uses a call capture proxy to intercept MOM interaction and may inject functionality with the use of interceptors. The framework was used to implement a full GISMO meta-level with support for interception in a portable manner. The implementation is also OMIP-compliant, facilitating interaction and coordination of self-management capabilities.

Part III

Evaluation

Chapter 7

Case Study – Coordination-Based Integration

The objective of this case study is to evaluate the benefits of coordination between self-managed systems and how it may improve the level of service delivered to an environment. In particular, the case study examines the benefits of information exchange within the problem domain of system integration.

7.1 Introduction

Integration is a common challenge within the software community. Whether connecting two simple applications or a collection of complex distributed systems the process of integration is demanding. Integrating two systems involves connecting together the outputs of one system to the inputs of another system, and vice versa. Within distributed systems, Message Oriented-Middleware (MOM) is an effective and flexible mechanism for interconnecting systems. A primary benefit of MOM is loose coupling between participants in a system - the ability to link applications without having to adapt the source and target systems to each other. This results in a highly cohesive, decoupled approach to connecting multiple systems [56].

MOM has been very successful in reducing both interface and temporal forms of coupling experienced with synchronous Remote Procedure Call (RPC) based mechanisms. This simplifies the integration of multiple applications and enables the creation of flexible and adaptable deployments. However, many forms of coupling exist within the integration process, such as message format and semantic coupling. MOM has even introduced its own form of coupling: Message Infrastructure Coupling (MIC).

This case study examines the cause and effects of MIC to highlight the limitation of current centralised solutions with respect to maintainability, scalability, and robustness. It then introduces a new decentralised approach based on coordinated self-management and evaluates it against the current solution.

7.2 Message Infrastructure Coupling

Successfully integrating two systems requires that the semantics of both systems be reconciled or bridged. Within a large-scale heterogeneous integration effort, this is one of the greatest challenges faced. To achieve a successful integration, all participants must have a common conceptualisation of the problem domain. David Orchard, a technical director at BEA Systems[1] [122], provides a description of the effects of semantic coupling on system integration:

> Imagine a Purchase Order system. A sender sends Purchase Orders to a receiver, who responds with successful completion of the order or failures. The receiver must understand all the nuances and details of the purchase order messages. Any interface or type change - such as changing the authentication structures, changing the timing of the authentication step, changing the purchase order messages, etc - well require that the sender change. And that means a programmer must perform the change.[2]

Semantics have an effect on a number of areas of application development. Most prominently from an integration perspective, they direct the definition of messages used to exchange information, which entity should receive the information, and how it should be received. However, semantic coupling can extend beyond these concerns to include the infrastructure used to exchange the messages. This form of coupling is referred to as infrastructure coupling.

Within the MOM domain, a number of constructs such as queues, topics, journals, and destination hierarchies are used for message exchange. These constructs or destinations are configured to meet the demands of a particular application domain. In many cases, the semantics of the domain will heavily influence their configuration. The resulting configuration will dictate how applications use the MOM to exchange messages, creating a tight coupling to a specific destination configuration. In order for an application to utilise the MOM, a programmer must alter the application to suit the chosen configuration, coupling the application to the MOM infrastructure. A motivational scenario is presented to illustrate this form of coupling in more detail.

7.3 Motivational Scenario

MOM may be utilised within a diverse set of deployment environments such as news and weather services, online auctions, marketplaces, enterprise application environments, and a wide range of information dissemination systems. To illustrate infrastructure coupling an Enterprise Application Integration (EAI) deployment scenario is examined, EAI is a domain where MOM is commonly utilised. While this scenario is EAI specific, the issues faced are relevant within any messaging scenario where an application developer is required to "wire up" the application to the messaging infrastructure. The scenario assumes the presence of a message standard such as ebXML, Universal Business Language (UBL), or RosettaNet to provide a common messaging format.

The scenario comprises two multi-national companies, Foo and Bar. Both companies operate their own independent MOM for communications. In order for an application to communicate with message consumers of either company, it must place its messages in the relevant destination within the MOM of the respective company. When designing their communications infrastructure, both

[1] BEA Systems are a key player in the integration domain.

[2] http://dev2dev.bea.com/pub/a/2004/02/orchard.html

companies must create destinations relevant to the needs of their respective message consumers. In order to maintain clarity within the scenario the interests of consumers are limited to the regional division of messages. However, communications may be routed along a number of lines such as departmental divisions, business units, document types (Invoice, Purchase Order, etc.), message priorities, and so on. Each additional division increases the complexity of the infrastructure and the effort required to connect to the system.

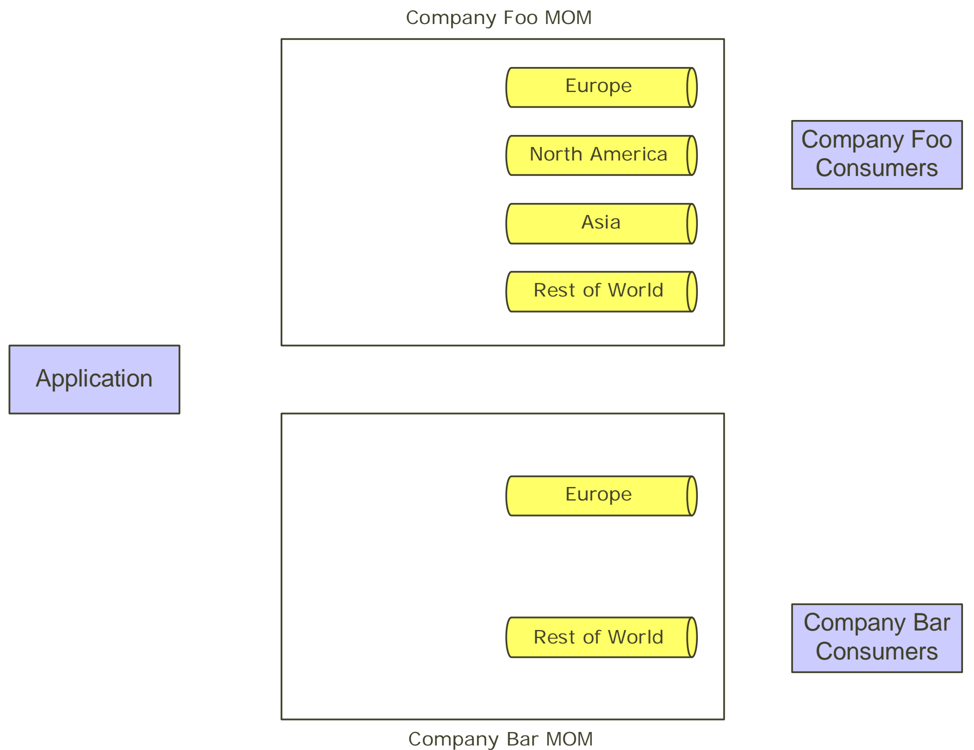


Figure 7.1 *A sample deployment scenario*

In this scenario, illustrated in Figure 7.1, the application on the left produces messages for the consumers on the right; messages are delivered using the MOM. Company FOO has four possible message destinations; one for each of Europe, North America, and Asia with a fourth catchall destination for remaining messages. Company Bar has a simpler infrastructure with only two destinations: one for Europe, and a catchall destination for all other regions.

As illustrated in Figure 7.2, the most straightforward method of integration available is to hardcode the application to both communication infrastructures, connecting directly to the destinations within the MOM. The main advantages of this solution are its straightforward implementation and decentralised message delivery. Distributing message delivery responsibility to the message producers will increase the scalability of the overall deployment. While it may be trivial to hardcode two simple messaging infrastructures, the complexity and cost of the process will grow geometrically as the number and intricacy of the messaging infrastructures increase. Hard coding the application directly to the MOM infrastructure creates tight coupling. Changes to the MOM infrastructure will require reciprocal changes in the application. This act nullifies some of the benefits gained by using an intermediary message exchange layer, effectively re-creating the interface-type coupling

experienced with RPC-based distribution, as infrastructure coupling within MOM.

Figure 7.2 *Hard-coded integration*

Application/business-logic is the main driving force in the configuration of the MOM infrastructures. Given that this form of logic can be prone to frequent changes to meet current business demands, it is reasonable to assume that the MOM's configuration will also need to change to meet current needs. It is vital that the integration approach used can accommodate these changes in a flexible manner. The challenge within this scenario is to integrate with each company's infrastructure in an effective scalable manner while minimising infrastructure coupling.

7.4 Integration 101

To evaluate integration options for the motivational scenario, it is important to examine the latest practice within integration. Experience gained within the EAI domain offers a number of important lessons when integrating multiple systems.

A key lesson from the perspective of this case study is that integration should be configured rather than coded. The latest practice within EAI is the development of Service-Oriented Architecture (SOA) using an Enterprise Service Bus (ESB). SOA is message/document-centric architectural style designed to achieve loose coupling amongst interacting software services; a service is a set of input messages sent to a single or composition of objects, with the return of causally-related output messages. An ESB is a messaging infrastructure designed to support the interconnection of services within an SOA by providing cross-protocol messaging, message transformation, and message routing capabilities. With these capabilities in place, it is possible to interconnect services

within an SOA by configuring the ESB. The concepts behind ESBs are a mindset for developing SOAs and are not specific to a particular product, although many do exist including Artix[1], Sonic ESB[2], and BEA AquaLogic Service Bus.

Traditional integration involves programmatically connecting two systems. Configuration-Oriented Integration (COI) works on the notion of connecting the inputs and outputs of two systems by utilising configuration files to direct a middleware platform to provide the connection. The concept of COI is an important lesson from the SOA/ESB approach to integration. COI can connect multiple systems in a flexible manner that can accommodate changes in application/business-logic. The second lesson is to minimise centralisation when integrating systems [123]; integration should be an edge case solution. In particular, scalability within a messaging solution can be improved by performing routing as close to the producer as possible [124].

7.5 Integration solutions

With COI in mind, a number of options are available for integrating the application within the deployment scenario. A common integration solution is introduced to the scenario with its benefits and limitations highlighted. An enhanced version of the solution is then presented. This new approach captures infrastructure configuration information allowing its exchange between systems. This exchange streamlines integration and reduces infrastructure coupling.

7.5.1 Centralised Content-Based Routing Integration Pattern

As a means to overcome the shortcomings of the hard-coded integration solution, a Content-Based Routing (CBR) integration pattern is used to deliver messages between producers and consumers. CBR transports messages to consumers based on the contents of the message. This is the equivalent of COI within the messaging domain; the routing rules are the configuration mechanism used to integrate applications.

CBR capabilities may be offered as a service via the MOM or by a message router within the messaging solution. MOM-based CBR delivers message from the producer to consumer without the use of a destination and is available in a limited number of MOM implementations using the publish/subscribe messaging model, including SIENA [62], REBECCA [97], Elvin [125], and ECO [104]. The main drawbacks with MOM-based CBR include its lack of support within the point-to-point messaging model and a lack of support within messaging standards. The widely adopted Java Message Service [78] only provides limited support for consumer-side CBR with message selection on destination subscriptions; both consumers and producers must still locate the relevant destination for their messages. Due to this lack of standardisation, MOM-based CBR can result in lock-in to a proprietary MOM implementation. To avoid this lock-in, we exclude MOM-based CBR as a viable option and focus on CBR provided within the messaging solution.

The CBR is a fundamental [16] design pattern commonly used within messaging solutions. This pattern enhances the hard-coded approach with the inclusion of a message router. The message router uses a rule-base to store routing instructions that direct messages to the relevant destination(s). A number of variations on the pattern are available including the Dynamic Router

[1] http://www.iona.com/products/artix/
[2] http://www.sonicsoftware.com/products/sonic_esb/index.ssp

[15] and the more generic Message Router [14].

Figure 7.3 *Centralised content-based routing integration pattern*

In Figure 7.3, the CBR pattern is introduced within the motivational scenario. Messages within the deployment take the following steps:

1. Application produces the message and sends it to the router
2. Router evaluates the message against its rule-base to match it to relevant destination(s) based on its content
3. Router forwards the message to the relevant destination(s)
4. Message consumer receives the message

The centralisation of the routing responsibility relieves the tight coupling of the hard-coded solution, decoupling the application from the destinations and allowing the messaging infrastructure to change without affecting the application. Integration from the application's perspective is simplified with the application delivering all messages to the router of the relevant company. The cost of integrating the application with further companies is minimal; no matter how complex their messaging infrastructure is the application simply forwards its messages to the message router of the company.

Advantages of the pattern emanate from its centralisation of the message sorting process. However, centralisation is also a weakness when it comes to the scalability and robustness of the de-

ployment. When examined from a message delivery perspective, a centralised delivery mechanism has the potential to form a scalability bottleneck, and can affect the robustness of the messaging solution by creating a single point of failure.

7.5.2 Decentralised Content-Based Routing Integration Pattern

The choice to centralise (router) or decentralise (hard-coded) the integration process has a major impact on the characteristics of the messaging solution. Centralisation simplifies integration while decentralisation improves performance. An ideal routing solution would maintain the benefits of a both the centralised and decentralised approaches while minimising their limitations. Thus providing the benefits of distributed message delivery, while preserving the maintenance benefits of a centralised rule-base.

Such an approach would be possible if the centralised rule-base was shared with message producers, enabling them to bypass a centralised router and distribute their messages directly to relevant destinations. This could be achieved by expressing the routing rules in an open and portable format, allowing the task of message sorting to be located at the edge (producer-side) of an integration scenario. This enhancement to the CBR approach, known as Decentralised-Content Based Routing (D-CBR), is illustrated in Figure 7.4.

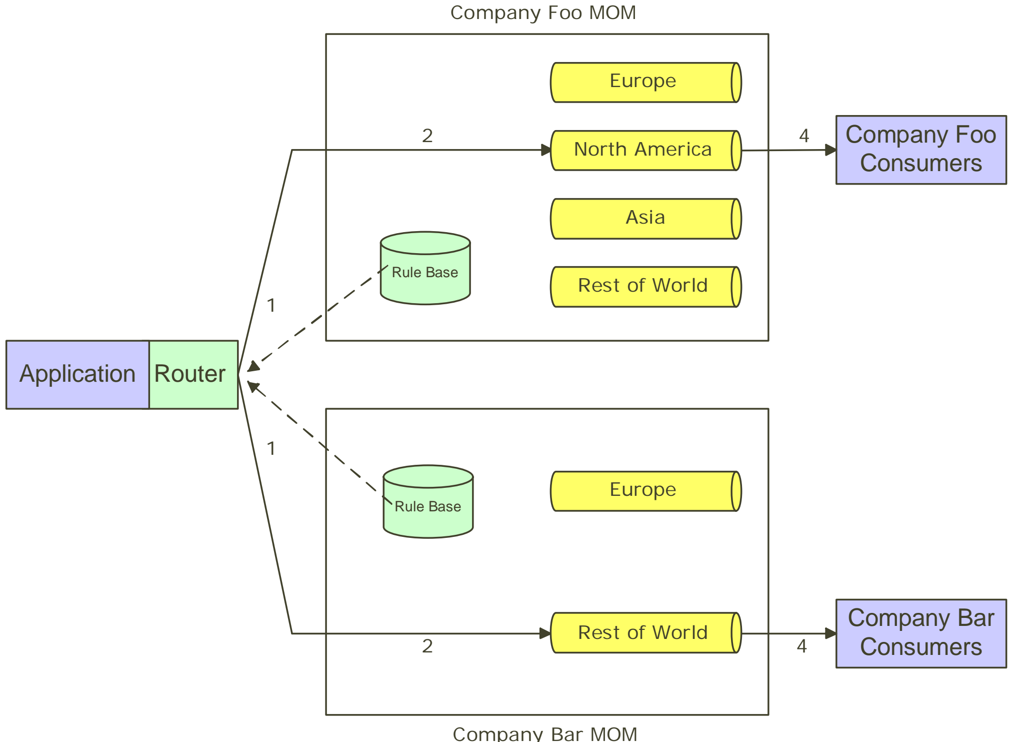


Figure 7.4 *Decentralised content-based routing integration pattern*

The motivational scenario utilising the new D-CBR pattern operates as follows:

1. Message producer receives routing rules from the centralised rule-base of each company
2. Message producer matches the message to the relevant destination(s) based on the rule-base

3. Producer sends message directly to destination(s) within each messaging infrastructure
4. Message consumer receives the message

This solution provides the best of both worlds by maintaining a centralised rule-base, increasing scalability with decentralised message delivery, and removing the single point of failure from the messaging solution. The major drawback of this integration solution is the initial setup of the support framework needed to facilitate the exchange of routing information [126, 127]. Within simple or low-volume messaging environments, this support framework may be overkill, negating any benefits gained. In addition, the approach may not be suitable for mobile clients with limited computational or power capabilities. These devices need to conserve their local capability and place the burden on less restricted systems utilising an approach similar to the centralised CBR integration pattern.

7.6 Achieving Decentralised-CBR

Evaluating the benefits gained from the D-CBR pattern requires the implementation and evaluation of both integration solutions. The centralised CBR solution can be created using standard messaging facilities and a rule-powered router. Decentralised-CBR has two additional requirements; routing rules must be expressed in a portable format, and a mechanism to exchange the rules must be in place.

Implementing the CBR design pattern using a traditional closed self-managed system is possible. However, as demonstrated in Figure 7.5, implementing the D-CBR design pattern is not feasible within a closed self-managed system as no mechanism exists to exchange rule information.

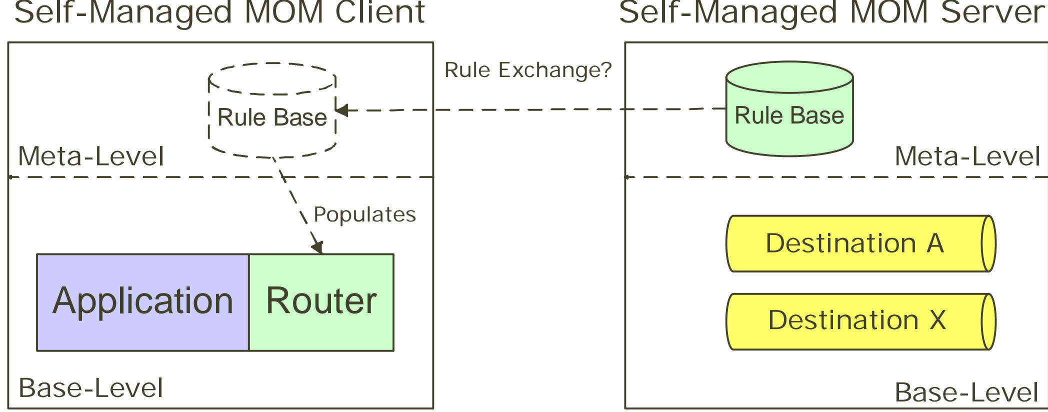


Figure 7.5 *D-CBR implemented within a closed self-managed MOM*

The requirements needed to implement the D-CBR design pattern can be provided by an open self-managed MOM platform such as the GenerIc Self-management for Message-Oriented middleware (GISMO). The requirement for a portable rule format is easily solved as GISMO may express its destination meta-model within the MOM-Domain Specific Language, an open XML based format. The second requirement is fulfilled by the Open Meta-level Interaction Protocol (OMIP), allowing MOM-DSL messages to be easily exchanged between participants within the deployment, as illustrated in Figure 7.6.

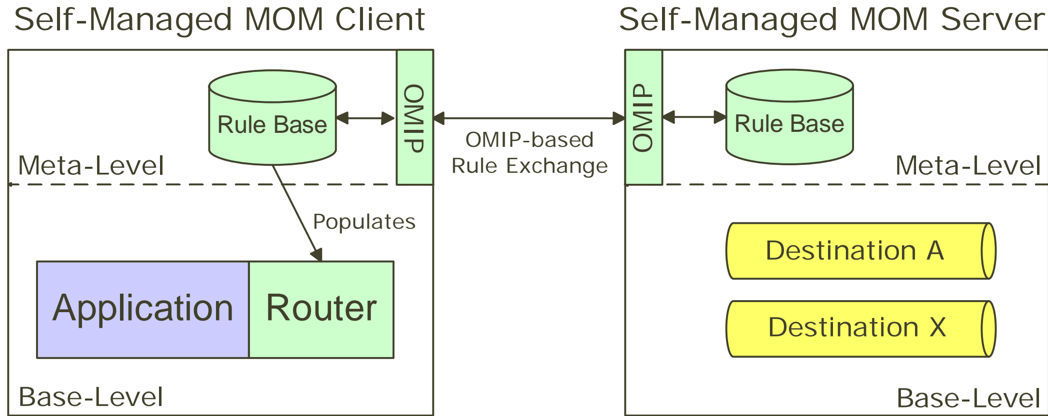


Figure 7.6 *D-CBR implemented within an open self-managed MOM*

This deployment demonstrates that a self-managed system's ability to track meta-state is not sufficient to implement the D-CBR pattern by itself. The key mechanism needed is an ability to exchange management information (meta-information) between systems in an open manner. The D-CBR design pattern demonstrates the benefits of an open interaction protocol such as OMIP for the coordination of self-managed systems.

7.6.1 MOM-DSL Messages for Decentralised-CBR

Routing instructions for the rule-base are expressed with the condition attribute of the destination state model within GISMO. Once captured in this model, the rule-base can be expressed within the MOM-DSL format and exchanged between participants using the OMIP. A sample rule-base for the deployment scenario expressed in MOM-DSL is provided in Table 7.1.

Using these MOM-DSL messages, producers within the deployment are able to obtain the purpose of destinations with the MOM infrastructure of each company and route their messages to the relevant destination. The D-CBR integration pattern demonstrates the possibility for information exchange and coordination of message participants at the self-management level. However, the exchange of routing rules is only one such example of information exchange, other forms of information may also be exchanged to describe different aspects of a messaging solution.

7.7 Evaluation

In order to compare the integration solutions, both approaches are implemented and performance evaluation test are executed. The motivational scenario is simulated using a test case with message producers sending messages through the routing solution to message consumers. The test cases replicate diversity within the deployment environment by adjusting the number of message producers and consumers, varying their ratio, and adjusting the size of the rule base (number of routing rules). The alternation of these characteristics produces a varied set of deployment environments to stretch the scalability of the routing solutions along each of these dimensions. A full discussion of MOM benchmarking techniques, including a description of the testbed used, is provided in Appendix A.

(a) Routing rules for company A expressed in destination meta-model

```
<MOM-DSL>
  <Reply replyID="1" response="accept">
    <DestinationState>
      <Single_Destinations>
        <Destination id="CompanyA_Inbox1" name="Europe" type="queue">
          <Condition attribute="Region" operator="=" value="Europe"/>
        </Destination>
        <Destination id="CompanyA_Inbox2" name="NorthAmerica" type="queue">
          <Condition attribute="Region" operator="=" value="North America"/>
        </Destination>
        <Destination id="CompanyA_Inbox3" name="Asia" type="queue">
          <Condition attribute="Region" operator="=" value="Asia"/>
        </Destination>
        <Destination id="CompanyA_Inbox4" name="RestOfWorld" type="queue">
          <Condition attribute="Region" operator="NOT IN"
                value="'Europe','North America', 'Asia'"/>
        </Destination>
      </Single_Destinations>
    </DestinationState>
  </Reply>
</MOM-DSL>
```

Europe
North America
Asia
Rest Of World

(b) Routing rules for company B expressed in destination meta-model

```
<MOM-DSL>
  <Reply replyID="1" response="accept">
    <DestinationState>
      <Single_Destinations>
        <Destination id="CompanyB_Inbox1" name="Europe" type="queue">
          <Condition attribute="Region" operator="=" value="Europe"/>
        </Destination>
        <Destination id="CompanyB_Inbox2" name="RestOfWorld" type="queue">
          <Condition attribute="Region" operator="NOT IN" value="Europe"/>
        </Destination>
      </Single_Destinations>
    </DestinationState>
  </Reply>
</MOM-DSL>
```

Europe
Rest Of World

Table 7.1 *Sample destination state model for deployment scenario*

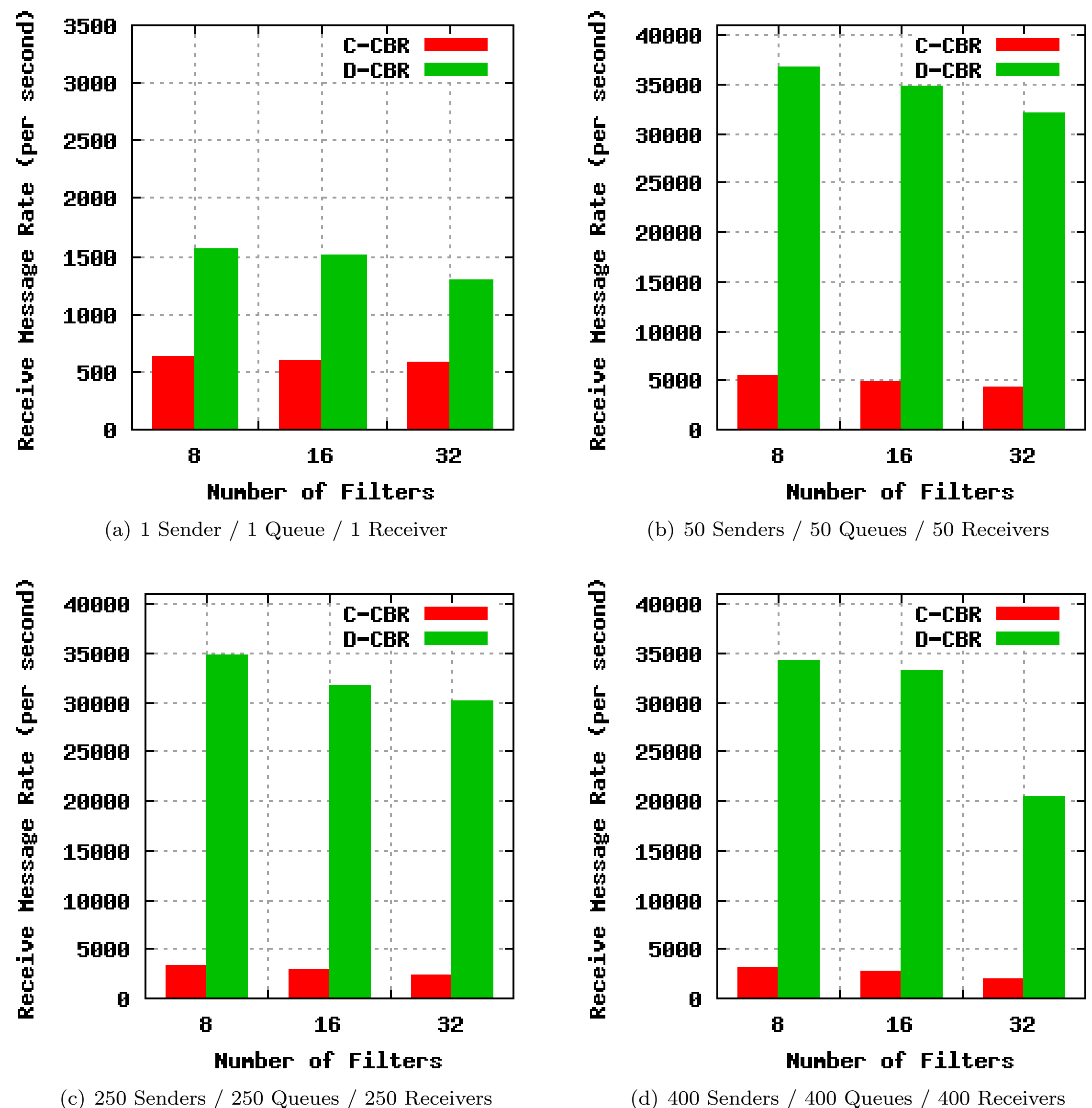
The benchmarking process used to evaluate the centralised and decentralised CBR integration patterns was extensive. A total of sixty test cases were performed, each test was run with producer and consumer clients evenly deployed across the client machines; each client was run under a independent thread and connection. Benchmarking took over 34 hours to execute with the largest test case involving 800 participants (400 producers and 400 consumers). The test cases investigate the scalability of both integration patterns in a wide range of deployment scenarios. Each scenario was careful selected to expose a variety of messaging requirements by adjusting the number of producers, consumers, varying their ratio, and adjusting the size of the rule-base. Test cases used the point-to-point messaging model (once-and-once-only delivery). The report metric chosen is the total number of messages received by all consumers per second (msg/sec). The test case results are detailed in Table 7.2, with the remainder of this section providing comprehensive analysis.

Test Case	Sender	Queue	Receiver	Filter	CBR (msg/s)	D-CBR (msg/s)	Increase	Increase (%)
One-to-One								
OtO-1	1	1	1	8	627.68	1,566.54	938.86	149.58%
OtO-2	1	1	1	16	602.96	1,507.07	904.11	149.95%
OtO-3	1	1	1	32	573.13	1,291.34	718.21	125.31%
OtO-4	50	50	50	8	5,488.75	36,774.69	31,285.94	570.00%
OtO-5	50	50	50	16	4,761.47	34,716.73	29,955.26	629.12%
OtO-6	50	50	50	32	4,252.82	32,032.17	27,779.35	653.20%
OtO-7	250	250	250	8	3,210.24	34,760.83	31,550.59	982.81%
OtO-8	250	250	250	16	2,889.58	31,596.91	28,707.33	993.48%
OtO-9	250	250	250	32	2,371.99	30,126.95	27,754.96	1170.11%
OtO-10	400	400	400	8	3,103.09	34,222.14	31,119.05	1002.84%
OtO-11	400	400	400	16	2,650.08	33,186.58	30,536.50	1152.29%
OtO-12	400	400	400	32	2,016.47	20,362.26	18,345.79	909.80%
Few-to-Many								
FtM-1	1	1	5	8	3,141.40	8,142.15	5,000.75	159.19%
FtM-2	1	1	5	16	2,899.35	7,373.00	4,473.65	154.30%
FtM-3	1	1	5	32	2,794.35	6,382.15	3,587.80	128.39%
FtM-4	50	50	250	8	5,250.59	32,407.64	27,157.05	517.22%
FtM-5	50	50	250	16	4,650.14	29,706.42	25,056.28	538.83%
FtM-6	50	50	250	32	4,191.88	29,364.28	25,172.40	600.50%
FtM-7	100	100	500	8	4,491.64	35,552.29	31,060.65	691.52%
FtM-8	100	100	500	16	3,929.08	32,456.32	28,527.24	726.05%
FtM-9	100	100	500	32	3,275.32	30,605.27	27,329.95	834.42%
Many-to-Few								
MtF-1	5	1	1	8	632.95	1,638.32	1,005.37	158.84%
MtF-2	5	1	1	16	595.59	1,556.07	960.48	161.27%
MtF-3	5	1	1	32	560.32	1,402.29	841.97	150.27%
MtF-4	250	50	50	8	4,664.75	29,906.83	25,242.08	541.12%
MtF-5	250	50	50	16	4,140.88	26,027.54	21,886.66	528.55%
MtF-6	250	50	50	32	3,659.33	25,168.73	21,509.40	587.80%
MtF-7	500	100	100	8	3,778.95	32,110.38	28,331.43	749.72%
MtF-8	500	100	100	16	3,381.29	27,982.82	24,601.53	727.58%
MtF-9	500	100	100	32	2,813.52	26,764.15	23,950.63	851.27%

Table 7.2 *Coordination-based integration case study benchmark results*

7.7.1 One-to-One Evaluation

The objective of the One-to-One test cases is to benchmark both integration patterns in a message producer/consumer balanced deployment scenario. The twelve test cases for this evaluation are test case OtO-1 to test case OtO-12. These tests expose the solutions to an ever-increasing quantity of message participants with the largest test involving 800 participants (400 producers and 400 consumers).



(a) 1 Sender / 1 Queue / 1 Receiver

(b) 50 Senders / 50 Queues / 50 Receivers

(c) 250 Senders / 250 Queues / 250 Receivers

(d) 400 Senders / 400 Queues / 400 Receivers

Figure 7.7 *Benchmark results integration patterns within the one-to-one test cases*

7.7.1.1 Results Summary

The minimum relative performance increase observed within the one-to-one benchmarks is experienced in test case OtO-3. In this test case, the CBR integration pattern processed 573.13 msg/sec (34,387.80 msg/min), while the D-CBR integration pattern processed 1,291.34 msg/sec (77,480.4 msg/min), an increase of 718.21 msg/sec (43,092.6 msg/min) or a 125.31% improvement in throughput.

The maximum relative performance increase was in test case OtO-9. During this test case, the CBR integration pattern processed 2,371.99 msg/sec (142,319.40 msg/min) and the D-CBR integration pattern processed 30,126.95 msg/sec (1,807,617 msg/min), an increase in throughput of 27,754.96 msg/sec (1,665,297.60 msg/min) or an 1170.11% throughput improvement.

The results of these test cases, illustrated in Figure 7.7, show that the D-CBR integration solution outperforms the centralised CBR solution in all test cases. As test cases become more strenuous, the margin of improvement increases. This trend is sustained in each test until the last three test case. In test cases OtO-10 and OtO-12, a small decrease is observed in the growth of the overall performance increase trend within the test cases.

7.7.2 Few-to-Many Evaluation

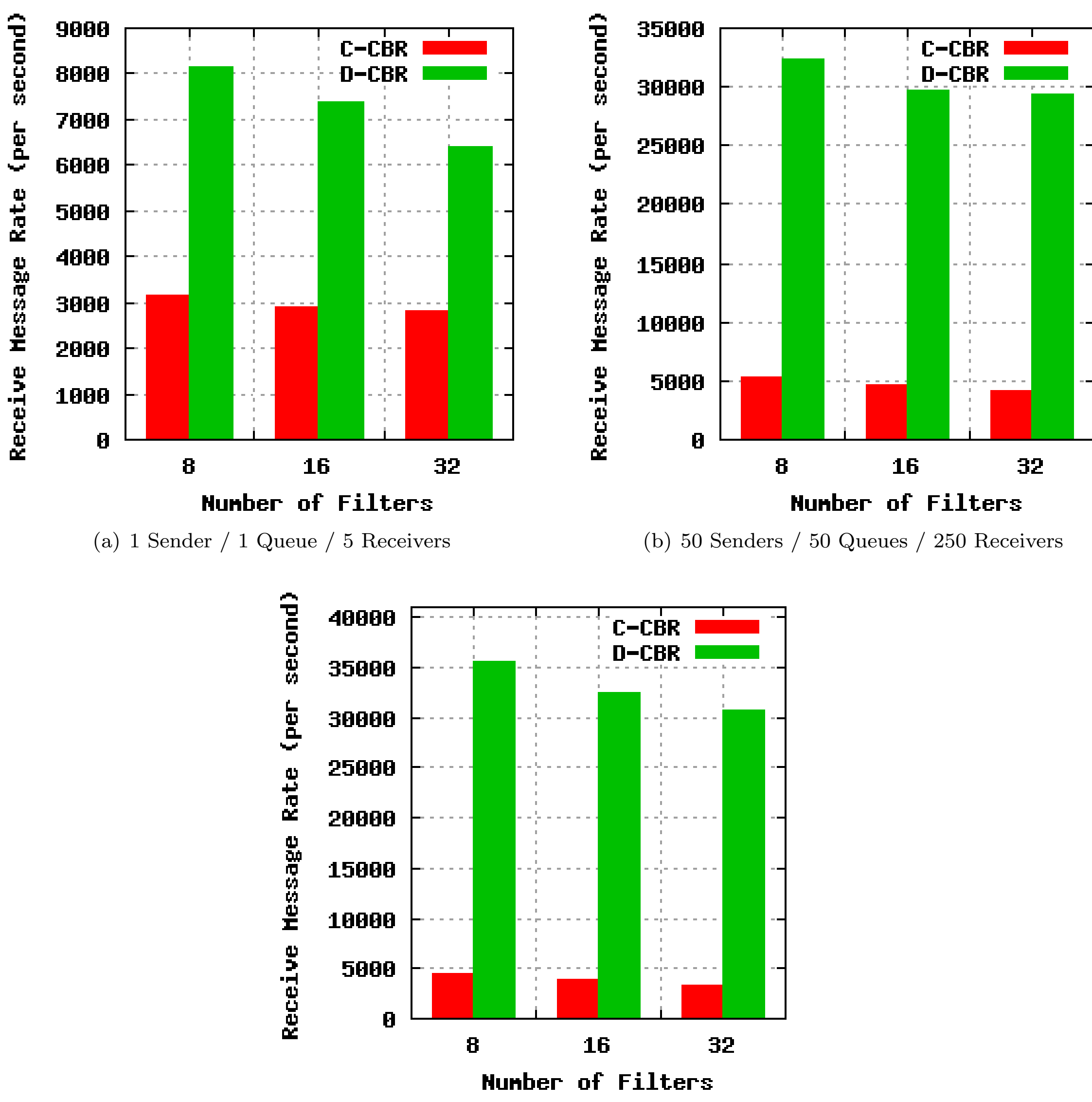
The objective of the Few-to-Many benchmarks is to test the integration patterns in a message consumer dense environment. The nine test cases for this evaluation are FtM-1 to FtM-9. These tests expose the patterns to an increasing quantity of message consumers with the largest test featuring 600 participants (100 producers and 500 consumers).

7.7.2.1 Results Summary

The minimum performance increase experienced within the benchmark set is in test case FtM-3. During this test case, the centralised CBR integration pattern processed 558.87 msg/sec (33,532.20 msg/min) and the D-CBR integration pattern processed 1,276.43 msg/sec (76,585.80 msg/min), an increase of 717.56 msg/sec (43,053.60 msg/min) or a 128.39% improvement.

The maximum relative performance increase was in test case FtM-9. Over the course of this test case, the CBR integration pattern processed 3,275.32 msg/sec (196,519.20 msg/min) and the D-CBR integration pattern processed 30,605.27 msg/sec (1,836,316.20 msg/min), an increase in throughput of 27,329.95 msg/sec (1,639,797 msg/min) or an 834.42% improvement.

The D-CBR integration pattern performs well in this type of environment with an increase in performance as demand intensifies with each successive test case. The trend is consistent for all test cases within the set, illustrated in Figure 7.8.



(a) 1 Sender / 1 Queue / 5 Receivers

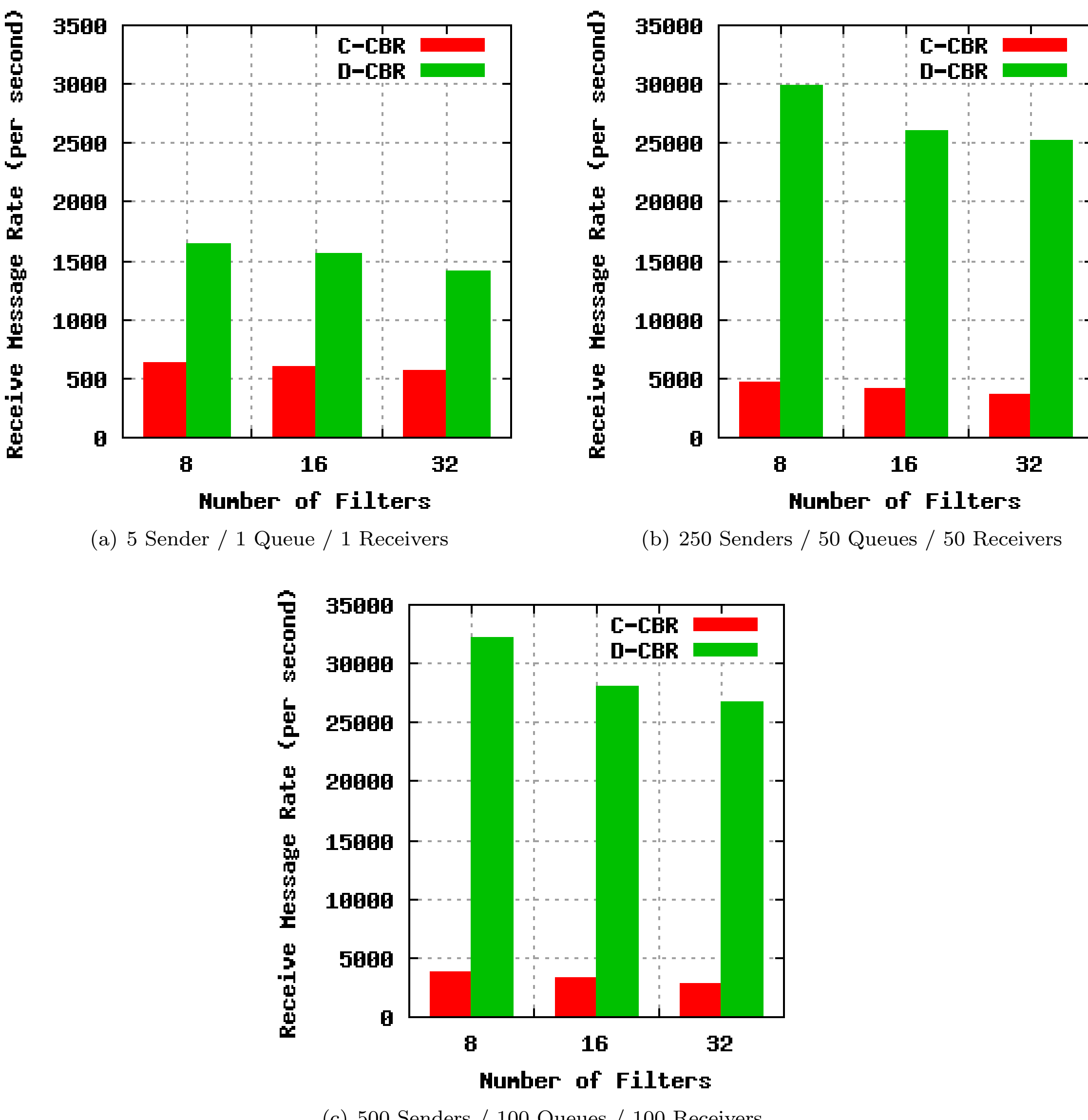
(b) 50 Senders / 50 Queues / 250 Receivers

(c) 100 Senders / 100 Queues / 500 Receivers

Figure 7.8 *Benchmark results for integration patterns within the few-to-many test cases*

7.7.3 Many-to-Few Evaluation

The purpose of the Many-to-Few test cases is to expose the integration patterns to message producer centric environments. The nine test cases for this evaluation are MtF-1 to MtF-9. These tests expose the solutions to an increasing quantity of message producers with the largest test containing 600 participants (500 producers and 100 consumers).



(a) 5 Sender / 1 Queue / 1 Receivers

(b) 250 Senders / 50 Queues / 50 Receivers

(c) 500 Senders / 100 Queues / 100 Receivers

Figure 7.9 *Benchmark results for integration patterns within the many-to-few test cases*

7.7.3.1 Results Summary

The minimum relative performance increase experienced within these benchmarks is in test case MtF-3. In this test case, the centralised CBR integration pattern processed 560.32 msg/sec

(33,619.20 msg/min) while the D-CBR integration pattern processed 1,402.29 msg/sec (841,37.40 msg/min), an increase of 841.97 msg/sec (50,518.2 msg/min) or an increase of 150.27%

The maximum relative performance increase was in test case MtF-9. During this test case the CBR integration pattern processed 2,813.52 msg/sec (168,811.20 msg/min) and the D-CBR integration pattern processed 26,764.15 msg/sec (1,605,849 msg/min), an increase in throughput of 23,950.63 msg/sec (1,437,037.80 msg/min) improving performance by 851.27%.

These test cases once more demonstrate the dominance of the D-CBR integration pattern in terms of scalability. The results of the many-to-few test cases, illustrated in Figure 7.9, in conjunction with the one-to-one and few-to-many test cases, demonstrate the capacity of the D-CBR integration to perform well under a wide range of challenging conditions.

7.7.4 Evaluation Summary and Discussion

The benchmarking process to evaluate the CBR and D-CBR integration patterns was extensive. Sixty test cases were performed on a private network of 12 machines, taking over 34 hours to execute. The test cases were designed to investigate the scalability of both patterns in a wide range of deployment scenarios; carefully selected to expose the patterns to a variety of messaging requirements.

In all the test cases executed, the Decentralised-CBR integration pattern outperformed the centralised CBR pattern. This was a predictable outcome of the benchmark process; decentralisation of the distribution process increases the scalability of the overall deployment. However, the level of performance increase was not expected, in almost all test cases the D-CBR pattern exhibited over a five fold improvement; over six test cases demonstrated a nine fold improvement, with two an eleven fold improvement.

The smallest scaled scenario was achieved in the few-to-many test cases. Test case FtM-3, consisted of a single producer and consumer with 32 routing rules. In this test case, the centralised CBR integration pattern consumed 558.87 msg/sec (33,532 msg/min, 2,011,920 msg/hour), and the D-CBR integration solution consumed 1,276.43 msg/sec (76,586 msg/min, 4,595,160 msg/hour), a throughput increase of 128.39%.

Test case OtO-4 produced the largest scaled scenario with the CBR integration pattern producing 5,488.75 msg/sec (329,325 msg/min, 19,759,500 msg/hour) and the D-CBR solution producing 36,774.69 msg/sec (2,206,481 msg/min, 132,388,860 msg/hour) an improvement in throughput of 31,285.94 msg/sec (1,877,156 msg/min, 112,629,360 msg/hour).

A comparison of the few-to-many and many-to-few test sets reveals that the few-to-many scenarios produce larger scaled deployments, however the many-to-few scenarios produce greater relative improvements when using the D-CBR integration pattern.

In conclusion, the evaluation process has validated the assertion of the benefits of the D-CBR pattern and the benefits of an open interaction protocol for the coordination of self-managed systems..

7.8 Summary

Message Infrastructure Coupling (MIC) can drastically affect the maintenance and performance of a MOM-based distributed deployment. Content Based Routing (CBR) is a popular approach used

to minimise the affects of MIC. However, current approaches to CBR are based on a centralised mindset that improves the maintainability but limits the scalability and robustness of a messaging solution. The Decentralised-CBR (D-CBR) pattern developed in this case study maximises maintainability and scalability by combining the advantages of a centralised rule-base with a distributed message delivery.

The key mechanism needed in the implementation of the D-CBR design pattern is the ability to exchange management information (meta-information) between systems in an open manner. The D-CBR design pattern demonstrates the benefits of an open interaction protocol such as Open Meta-level Interaction Protocol (OMIP) for the coordination of self-managed systems.

The benchmarking process to evaluate the CBR and D-CBR integration patterns was extensive. Sixty test cases were performed on a private network of 12 machines, taking over 34 hours to execute. The test cases were designed to investigate the scalability of both patterns in a wide range of deployment scenarios; carefully selected to expose the patterns to a variety of messaging requirements. The Decentralised-CBR integration pattern significantly outperforms the centralised CBR pattern. In almost all test cases the D-CBR pattern exhibited over a five fold improvement; over six test cases demonstrated a nine fold improvement, with two an eleven fold improvement. The evaluation process validates the assertion of the benefits of the D-CBR pattern and the benefits of an open interaction protocol for the coordination of self-managed systems.

Chapter 8

Case Study – Coordinated Self-Managed MOM

The purpose of the case study is to illustrate the benefits of self-management capabilities for MOM and to support the case for the coordination of such capability. Within this case study, GISMO adapts the configuration of a MOM destination hierarchy to efficiently service its deployment environment. The resulting reflective destination hierarchy is then empirically compared to a contemporary non-reflective destination hierarchy to reveal the more efficient and scalable solution.

8.1 Motivational Scenario

As a means to provide background rational for this case study, a motivational scenario of a factitious "Movie Information Service" (MIS) is presented. The MIS provides information on newly released movies, in-production movies, and a back-catalogue of classic films. Services such as the MIS can potentially contain a large collection of information. To maintain clarity within this scenario the MIS tracks a limited subset of information consisting of movie genera, cast, director, rating, and release date as summarised in Table 8.1.

Information	**Description**	**Sample Values**
Name	The title of the film	Star Wars, Indiana Jones, Clerks, Dogma, James Bond, Schindler's List, Raising Arizona, The Lord of the Rings, ...
Genera	The classification of the film	Horror, Sci-Fi, Action, Comedy, Drama, ...
Cast	The leading actors and co-actors in the movie	Harrison Ford, Sean Connery, Tom Hanks, David Prowse, Liz Taylor, ...
Director	The director of the movie	George Lucas, Kevin Smith, The Coen Brothers, Steven Spielberg, Peter Jackson, Michael Moore, ...
Rating	Average critic rating	(* | ** | *** | **** | *****)
Release Date	Date of release	19/05/2005

Table 8.1 *A sample movie service message structure*

The MIS requires a broad dissemination mechanism and is an ideal candidate for the powerful one-to-many publish/subscribe-messaging model. Within this messaging model, message consumers declare their interest in messages by providing a subscription to the MOM. Subscriptions define a consumer's interest in a particular type of message by defining a constraint on the attributes of a message. Some sample subscriptions for the MIS are:

- All Sci-Fi movies (Genera = "Sci-Fi")
- All movies that are director by Kevin Smith (Director = "Kevin Smith")
- All Sci-Fi films that are directed by Steven Spielberg (Genera = "Sci-Fi" AND Director = "Steven Spielberg")

Providing a subscription to the MOM allows it to limit messages a consumer receives to ones that match its interests, avoiding the delivery of any unwanted messages.

The MIS scenario is a common messaging challenge within the MOM domain. The main challenges faced within these scenarios is balancing the expressability of the subscription with the scalability of the messaging solution [65, 124]; the more expressive the subscription mechanism the less scalable the solution.

A number of techniques can minimise this effect. One of these, filter covering, discussed in Chapter 3, attempts to minimise subscription constraint evaluation by grouping common filters together. Another approach to increase scalability is to filter the message as close to the message producer as possible. Both of these techniques are natural characteristics of destination hierarchies.

Destination hierarchies are a destination grouping mechanism within the publish/subscribe messaging model. This type of structure allows destinations to be defined in a hierarchical fashion, so that they may be nested under other destinations. Each sub-destination offers a more granular selection of the messages contained in its parent. Clients of destination hierarchies subscribe to the most appropriate level of destination to receive the most relevant messages.

The objective of this case study is to investigate the combination of coordinated self-managed techniques with a destination hierarchy. The next section introduces the current routing scenarios for the test case, Section 8.2.3 introduces reflective destination hierarchies, and the remainder of the chapter discusses the benchmark evaluation of both static and reflective destination hierarchies and commentary on the results.

8.2 Routing Scenarios

To recreate a scenario like the MIS, the demands experienced within the deployment must be replicated. These demands are dictated by the requirements of the message participants within the deployment; the more diverse the requirements are the harder the messaging solution must work to meet them. Requirements are replicated within the test case by using groups of consumers and producers. To create diversity, message participants are assigned to an interest group with a predefined set of messaging requirements (subscription constraints). Message producers of an interest group produce messages that match the subscriptions of the group's message consumers. The use of multiple groups enables the creation of diverse deployment scenarios allowing a test case to benchmark the proficiency of different routing solution in dealing with messages and subscriptions of varying types.

The proposed interest groups used within this case study employ an exponentially increasing subscription complexity. To show the effect of filter covering, each group builds on the constraints of the previous group(s). This relationship is illustrated as a Venn diagram in Figure 8.1. These relationships and their effect on message delivery ratios are an important consideration when examining the results of the case study. Not only does the size of the subscription increase with each group but the ratio of messages delivered also increases.

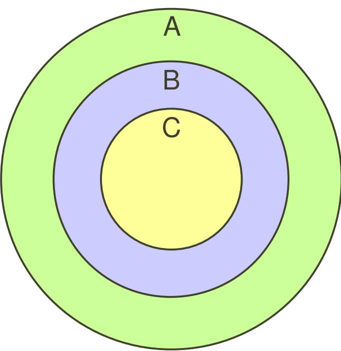


Figure 8.1 *Venn diagram of relationships between interest groups*

Since this case study is run within the publish/subscribe messaging model, topics destinations are used, resulting in a one-to-many message delivery. This has an effect on the number of messages sent and delivered. Table 8.2 details the correlation with respect to message delivery ratios between message groups.

Message Producer	Simulated Messaging Requirements	Message Recipients
Group A	Ability to cope with subscriptions of interest group A (8 Filters = 8 new constraints)	Group A Only
Group B	Ability to cope with subscriptions of interest group B (16 Filters = Group A + 8 new constraints)	Groups A & B
Group C	Ability to cope with subscriptions of interest group C (32 Filters = Group A + Group B + 16 new constraints)	Groups A, B, & C

Table 8.2 *Summary of message delivery relationships between interest groups*

The message delivery ratios between each interest group is broken down as:

- Group A messages will only be received by group A consumers. A 1:1 delivery ratio.
- Group B messages will be received by consumers from groups A & B. A 1:2 delivery ratio.
- Group C messages will be received by consumers from groups A, B, & C. A 1:3 delivery ratio.

With the messaging participants introduced, the remainder of this section discusses possible routing solutions.

8.2.1 Single Destination

The most straightforward routing solution available is to use a single destination to route messages. This dissemination mechanism uses a single topic for all interest groups, forcing the maximum amount of work on the MOM, requiring it to filter individual subscriptions on each message sent to the topic. A single destination routing solution is illustrated in Figure 8.2.

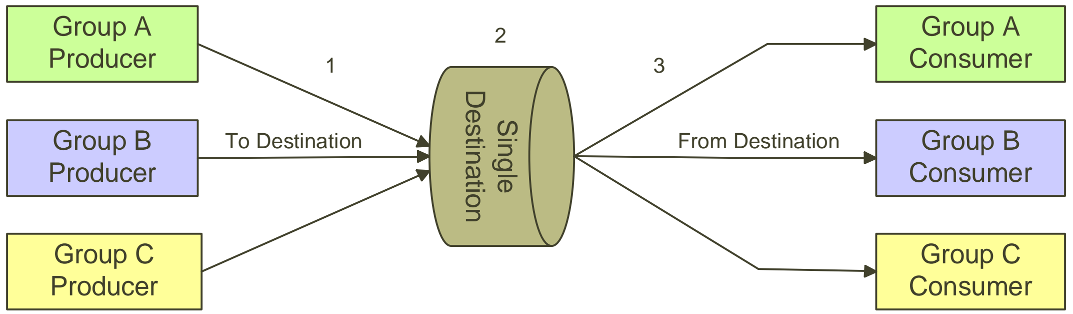


Figure 8.2 *Single destination routing scenario*

1. Each interest group's message producers send their messages to the single destination.
2. Message consumers from each interest group subscribe to the single destination. Each group's subscription constraints are evaluated on the messages at the destination.
3. Messages that match subscription constraints are delivered to relevant consumers.

The single topic approach is a simple routing solution for a MIS type service. However, its simplistic structure does not take advantage of techniques such as pre-filtering (source filtering) and filter covering [124] to increase scalability and throughput. Such capability may be implemented within a proprietary MOM but it is not a natural characteristic of the messaging solution. This forces the maximum amount of exertion on the MOM provider limiting the scalability of the deployment. To overcome this limitation more complex structures such as destination hierarchies were developed.

8.2.2 Static Destination Hierarchy

Destination hierarchies are a key technique to facilitate an efficient message pre-filtering in a provider neutral manner. Destination hierarchies alleviate the burden of extensive filtering within large-scale deployments. This type of destination structure allows the definition of topics in a hierarchical fashion, so that topics may be nested under other topics. Each sub-topic offers a more granular selection of the messages contained in its parent topic (filter covering). Clients of hierarchical destinations subscribe to the most appropriate level of destination to receive the most relevant messages. In large-scale systems, the grouping of messages into related types (i.e. into sub-topics) helps to manage large volumes of different messages [64].

Hierarchical destinations require that the destination namespace schema be both well defined and universally understood by the participating parties. This requires the hierarchy to be pre-defined to meet requirements for a specific deployment scenario. Due to this prerequisite, hier-

archical destinations are used in routing situations that are more or less static and have limited capabilities to deal with new messaging requirements.

Illustrated in Figure 8.3 is a static hierarchy routing solution. Within this routing solution, the hierarchy has two destinations, “A” and “B”, designed for interest groups A and B respectively. Interest group C has been omitted from the hierarchy design and is used to replicate an unanticipated messaging requirement within the environment. Group C demonstrates the limitations of a static hierarchy, enabling the observation of the hierarchy’s behaviour when it encounters an unexpected requirement.

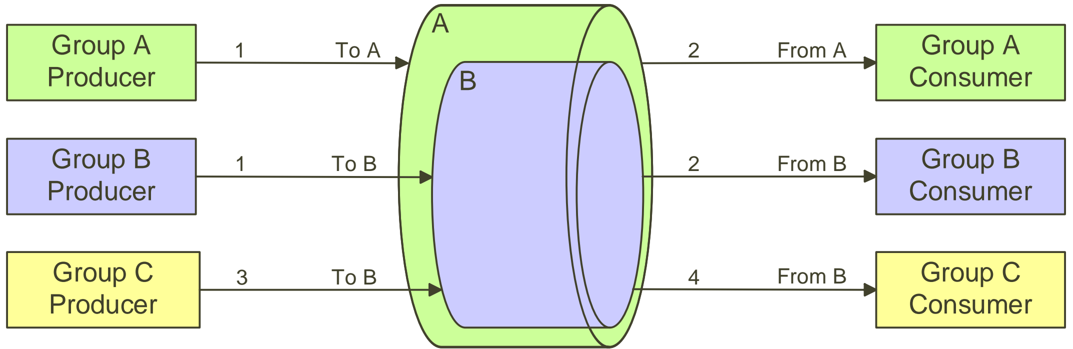


Figure 8.3 *Static destination hierarchy routing scenario*

1. Interest group A & B’s message producers send their messages to their respective destinations A and B within the hierarchy (pre filtering).

2. Message consumers from interest group A and B pre-empt constraint evaluation by subscribing directly to destinations A and B within the hierarchy.

3. Since the hierarchy is not designed for interest group C, no destination C exists. The most appropriate destination within the hierarchy for messages from group C is destination B (destination B provides filter covering for the constraints of group A & B which constitute 50% of C’s constraints).

4. Consumers from group C subscribe to destination B. Half of group Cs subscription constraints are covered by the hierarchy, the remaining constraints of group C are evaluated on all messages within destination B (all messages from group B & C). All messages that match the group C subscriptions are delivered to group C consumers.

The pre-filtering associated with the static destination hierarchy is a very effective method for minimising the workload associated with subscription evaluation for the pre-defined requirements of groups A and B. However, once the unanticipated requirement of group C is encountered, the hierarchy has no way of dealing with it and requires the MOM provider to supplement it with constrain evaluation. Reflective destination hierarchies are designed to overcome this limitation.

8.2.3 Reflective Destination Hierarchy

A Reflective Destination Hierarchy (RDH) [33, 128] is designed with the ability to autonomously self-adapt to its deployment environment. The objective of the adaptation is to maximise the ben-

efits from pre-filtering and filter covering while minimising constraint evaluation. This is achieved by creating appropriate sub-topics within the hierarchy at runtime to meet current operating conditions.

The routing scenario in Figure 8.4 introduces a RDH. The initial configuration of the hierarchy (destinations A and B) is the same as the preceding static hierarchy. This enables a direct comparison between the hierarchies to assess the benefits of reflective destination hierarchies.

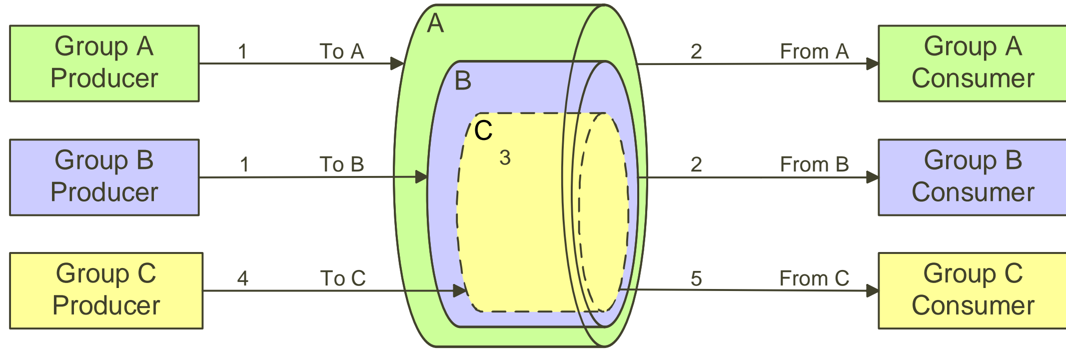


Figure 8.4 *Reflective destination hierarchy routing scenario*

1. As with the static hierarchy, interest group A & B's message producers send their messages to the destinations A and B within the hierarchy.
2. Again, message consumers from interest group A and B pre-empt constraint evaluation by subscribing directly to destinations A and B within the hierarchy.
3. Unlike the static hierarchy, a reflective hierarchy will recognise a need for a destination for interest group C and create destination C. Message participants are informed of this change.
4. Interest group C's message producers send their messages to the newly created destination C.
5. Message consumers from interest group C remove the need for constraint evaluation by subscribing directly to destination C, improving the performance of the hierarchy.

The ability of a RDH to identify current operating conditions and adapt the hierarchy structure to meet these requirements is vital to maximise the benefits of pre-filtering and filter covering, minimising filter constraint evaluation. A RDH allows message participants to choose the criteria, allowing a hierarchy to evolve (or grow) and constantly adapt into one customised specifically for the deployment environment. Starting from a single destination (seed) a customised destination hierarchy (tree) can grow based on the expressed requirements of the environment; relieving the need to predefine the hierarchy. The design of a RDH is now discussed.

8.3 Creating Reflective Destination Hierarchies

The GISMO reflective framework provides an ideal foundation for the creation of RDH with its support for coordinated self-managed MOM, a destination meta-model, and pluggable reflective policies. All of these capabilities are necessary to implement RDHs.

An important part of creating an effective reflective destination hierarchy is defining the criteria that will direct the adaptation of the hierarchy. A number of potential reflective criteria were identified for the creation of a reflective hierarchy. The criteria identified analysed different usage metrics for hierarchy adaptation; these criteria included the analysis of message traffic submitted to the service and consumer subscription analysis. When evaluating each potential candidate it is important to consider the quantity of information that must be tracked and the cost associated with its collection at runtime. As discussed in Chapter 2, the cost of the reflective process is closely connected to its success.

Consumer subscription analysis was the metric chosen to prototype reflective hierarchies due to its requiring a smaller amount of information tracking with simple runtime collection requirements, offering a greater chance of success. The presence of a subscription meta-model within GISMO was also a factor. Subscription monitoring alters the hierarchy based on current subscription constrains. The remainder of this section discusses the design of the subscription monitoring reflective policy including policy triggers, analysis process, adaptation algorithm, and the role of coordination within the policy.

8.3.1 Subscription Monitoring Policy - Overview

The subscription monitoring policy observes consumer subscription constraints on destinations within the hierarchy. The examination of subscriptions exposes common constraints and their most frequent values. If a large number of similar constraints exist, the policy reacts by adapting the destination hierarchy to provide a new destination for these constraints, pre-empting the need for their evaluation, increasing the filter covering and pre-filtering range of the hierarchy.

8.3.2 Policy Triggers

Policy invocation can be achieved with the use of time-based triggers or by attaching the policy to any of the subscription related reflective locations (Create and Delete). Once the trigger(s) method is chosen, the next step is to decide how often the policy should be executed.

Self-managed systems aspire to reap the maximum benefit possible from the minimum amount of cost (analysis and adaptation). Within this or any scenario, it would not be appropriate for the policy to be executed every time a subscription is created or deleted. In order for the policy to be effective, the state must have changed a reasonable amount for an adaptation to be of any real benefit. With this in mind, the policy implementation allows the restriction of its execution to a function of its invocation frequency; limiting its execution to every 10, 100, or N times it is invoked. This allows a reasonable quantity of change to have occurred before any reflective activity executes, increasing the likelihood of a beneficial adaptation to cover the cost of the reflective process.

8.3.3 Analysis Process

The primary analysis tool used by the policy examines the subscription and destination meta-model to identify common subscription constrains for destinations within the hierarchy. With the use of this information, it is possible to identify and create new sub-destinations to meet the needs of common subscription constraints, such as the destination for group C in the test case. The next section describes the reflective logic used within this decision making process.

8.3.4 Adaptation Algorithm

With the analysis information on hand, the next action of the policy is to decide on suitable adaptations to perform, if any. In many cases, it may not be appropriate to make an adaptation and it is the responsibility of the adaptation algorithm to make this decision. This algorithm must balance the cost against the potential benefits of the adaptation; the logic and control used for this decision making process is illustrated in Figure 8.5.

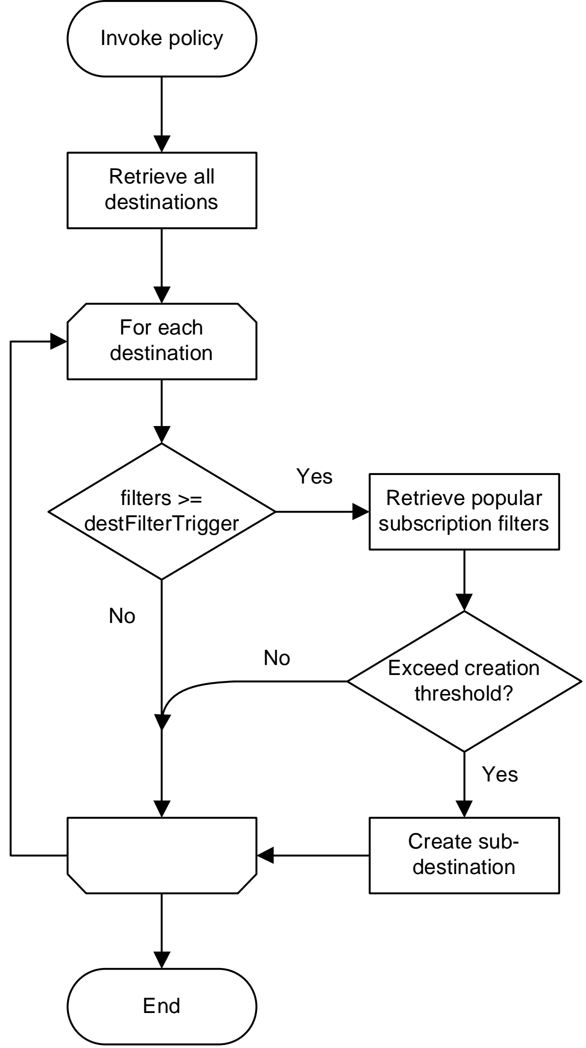


Figure 8.5 *Reflective destination hierarchy adaptation algorithm*

Within this algorithm, two control variables set the criteria for the creation of a new sub-destination. The 'destFilterTrigger' control variable is use to specify the number of filter constraints a destination must posses in order for it to be considered for adaptation. Once a destination with a sufficient number of subscription constraints is identified, its subscriptions are examined to expose common constraints. The common constraints exposed are evaluated against the 'creation_-threshold' control variable to decide if a new sub-destination is required.

Within the current implementation of this algorithm, the administrator specifies the value of

both of the control variables. However, the algorithm could be enhanced with the use of artificial intelligence techniques to self-optimise and refine the boundaries within the algorithm, possible techniques include reinforcement learning [129] and collaborative reinforcement learning [27].

8.3.5 Realisation

The realisation process of the policy has two parts. The most obvious realisation involves changing the destination hierarchy using the destination meta-model described in Chapter 5. The second part of the realisation process involves informing relevant consumers and producers of any new sub-destination(s). This is achieved with the use of MOM-DSL event notifications to inform external participants when the destination meta-model changes.

8.3.6 Coordination

Coordination plays an important role in the realisation of RDH. As described in Chapter 7, coordination can be used to reduce message infrastructure coupling, allowing the adjustment of the hierarchy to suit the current environment. The role of coordination within RDH is to inform message participants as to the structure and purpose of the hierarchy. Initial OMIP interactions between participants exchange destination information to inform consumers and producers of the structure and role of the hierarchy. Further interactions inform message participants of changes to the hierarchy via MOM-DSL event notifications. When the destination hierarchy is adapted, any participants subscribed to these events are informed of the change and can adjust their subscriptions accordingly.

The coordination interactions with RDH provide an exemplar of how coordinated self-managed systems can improve the overall service provision within an environment.

8.4 Benchmarking Dynamic Environments

The objective of this case study is to compare a reflective destination hierarchy against a static destination hierarchy to reveal any performance benefits. As discussed in Appendix A, the traditional approach to benchmarking MOM solutions is a test case with a fixed number of message producers communicating through a MOM to a fixed number of messages consumers using a fixed number of filters. These conditions are static (remaining constant) for the duration of the test case. This is the common form of MOM test case and has been used by a number of industrial and academic benchmarking efforts [92, 130, 131, 132, 133]. This approach is known as static benchmarking.

Static benchmarking is of limited use for evaluating reflection-based self-managed systems. While it is possible to test the performance of a self-managed system under one set of conditions, static benchmarking has no capacity to test the system for its ability to adapt to new and changing conditions: the key objective of a self-managed system. With this deficiency in mind, a new approach to test case design for self-managed systems is purposed.

The key to successfully benchmarking a reflection-based self-managed system is to gauge its reflective capability. To achieve this, the test case must simulate varying conditions within dynamic environments. An effective method of achieving this is to stage "environmental" changes over the duration of the test case to replicate a non-static environment. With these environmental changes, it is possible to gauge the ability of the self-managed system to deal with them, thus benchmarking

the reflective ability. With this change in approach to test case design, a new metric is needed to measure performance.

The common metric used within static benchmarks is the *message-received per interval* rate. Typically, this is reported as an average for all time intervals during the duration of the test case, i.e. 15,543 msg/sec. However, within a dynamic benchmark, there is an important relationship between an intervals message rate and the activity of the dynamic environment during that interval; a single throughput figure for the test case does not represent this relationship. It is vital to observe the reaction of a self-managed system to an environmental change; how it behaved before the change, how it reacted to and during the change, and how it performed after the change. To capture these characteristics a temporal performance trend may be used to highlight environmental changes and the system's reaction for the duration of the test case. Appendix A provides a full discussion of MOM benchmarking techniques, including a description of the testbed used. The remainder of this section provides a systematic walkthrough of the dynamic MOM test case developed for this case study.

8.4.1 Dynamic MOM Test Case Walkthrough

In order to simulate a dynamic MOM environment, the dynamic conditions within the environment must first be identified. From the perspective of the MOM, the three main dynamic conditions are the number of message producers, message consumers, and subscription complexity.

Creating a dynamic test case for a self-managed MOM can be achieved by altering these conditions at runtime with the staggered introduction of new message producers and consumers into the environment. Messaging conditions can also be altered by varying the subscriptions (interests) of message participants over the duration of the test case.

Within the test case scenario three interest groups exist, groups A, B, and C. To replicate a dynamic environment these groups will join the benchmark in a staggered manner. This will allow the observation of the messaging solutions proficiency in dealing with each group. Each test case used for these benchmarks will execute for the duration of one hundred 60-second intervals. Test cases are run with the following interest groups joining at staggered times:

- Interest Group A: Join at interval 1, participant until interval 100
- Interest Group B: Join at interval 25, participant until interval 100
- Interest Group C: Join at interval 50, participant until interval 100

The timeline for the test cases builds on the default test case timeline, discussed in Appendix A. The dynamic test case timeline is enhanced with the staggered joining of interest groups; the enhanced timeline is illustrated in Figure 8.6.

The test case is specifically designed to replicate operational conditions to benchmark both hierarchies. Intervals 1-50 are limited to interest groups A and B to allow the examination of the behaviour of both hierarchies under anticipated operating conditions. The test case then introduces an unanticipated requirement, interest group C, from interval 50-100, to demonstrate their ability to cope with unanticipated operating conditions.

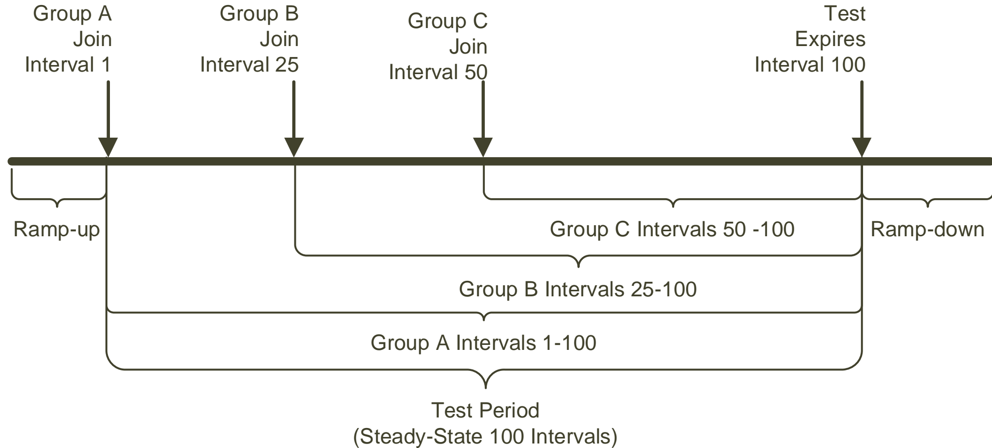


Figure 8.6 *Dynamic test case timeline*

8.5 Evaluation

With the use of the dynamic benchmark described in Section 8.4.1 it is possible to benchmark both reflective and static destination hierarchies. However, to comprehensively test both hierarchies, a range of test cases must be executed with varying numbers of message producers, message consumers, and subscription complexity. In the evaluation a total of 26 test cases were executed, each of these test cases follow the same timeline as described in the previous section. Listed in Table 8.3 is a full list of all test cases within the evaluation. Within each test case, message publishers and message subscribers are comprised of equal numbers from each interest group (i.e. test case FtM-3 contains 20 publishers and 100 subscribers from interest group A, B, and C). The remainder of this section provides a detailed analysis of the results from the benchmarking process.

8.5.1 One-to-One Evaluation

The one-to-one test cases benchmark both hierarchies in a producer-/consumer-balanced deployment. This set of tests contains five test cases ranging from 6 to 600 messaging participants.

8.5.1.1 3 Publishers / 3 Subscribers

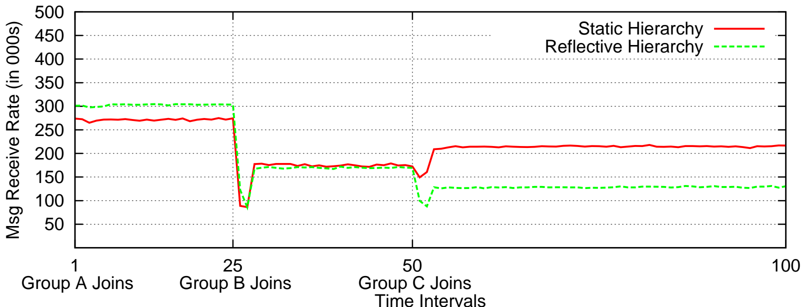


Figure 8.7 *Result of test case OtO-1 (3 publishers and 3 subscribers)*

Test Case ID	Publishers	Subscribers
One-to-One		
OtO-1	3	3
OtO-2	30	30
OtO-3	90	90
OtO-4	150	150
OtO-5	300	300
Few-to-Many		
FtM-1	3	15
FtM-2	30	150
FtM-3	60	300
FtM-4	90	450
Many-to-Few		
MtF-1	15	3
MtF-2	150	30
MtF-3	300	60
MtF-4	450	90

Table 8.3 *Coordinated self-management case study benchmark test cases*

The first of the one-to-one test cases, illustrated in Figure 8.7, is the smallest test case within the evaluation process using only six participants. At the most basic stage of this test case, 0-25 intervals with participants from only group A, the static hierarchy outperforms the reflective hierarchy even though both hierarchies are identical. This performance difference disappears when group B joins the test case at interval 25. Once group C joins at interval 50, the static hierarchy outperforms the reflective hierarchy for the remainder of the test by an average for 40%. At this very early stage of the benchmarking process, the prospects for a reflective hierarchy within a small-scale environment do not look promising.

8.5.1.2 30 Publishers / 30 Subscribers

During this test case, shown in Figure 8.8, both hierarchies have near identical performances over the first 50 intervals with an average deviation of –1.5% (intervals 0-25) and 0.5% (intervals 25-50). When group C joins the test case at interval 50, the reflective hierarchy outperforms the static hierarchy by an average of 26.96% for the duration of the test case. The amount of time taken by the reflective hierarchy to reach optimal throughput is minimal.

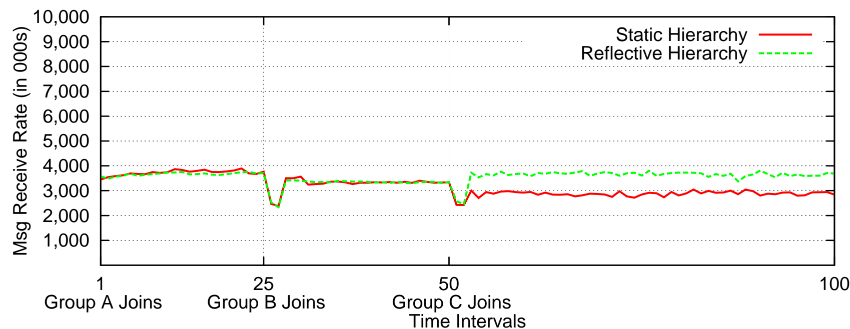


Figure 8.8 *Result of test case OtO-2 (30 publishers and 30 subscribers)*

8.5.1.3 90 Publishers / 90 Subscribers

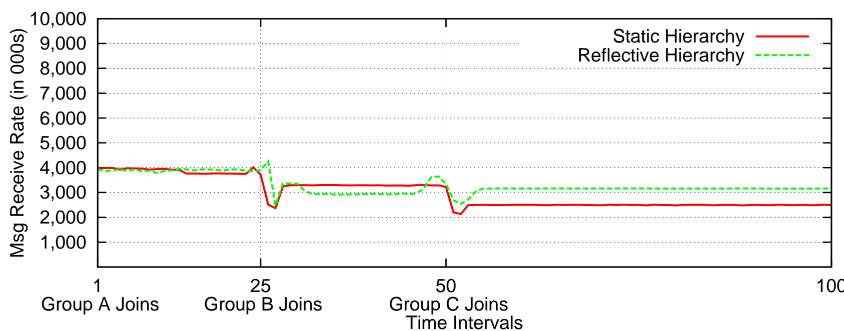


Figure 8.9 *Result of test case OtO-3 (90 publishers and 90 subscribers)*

In this 180 participant test case, presented in Figure 8.9, there is a slight difference between the performances of both hierarchies over the first 50 intervals. The reflective hierarchy slightly underperforms for 15 intervals between intervals 30 and 45, with an average underperformance of –4.69% between intervals 25 to 50. When group C joins at interval 50, the reflective hierarchy outperforms the static hierarchy by an average of 26.06% for the remainder of the test case.

8.5.1.4 150 Publishers / 150 Subscribers

During the first 50 intervals of this 300 participant test case, illustrated in Figure 8.10, both hierarchies have near identical performance and an average deviation of less that 1%. At interval 50, when group C joins, the reflective hierarchy outperforms the static hierarchy by an average of 19.88%, reaching its optimal performance within 5-10 intervals.

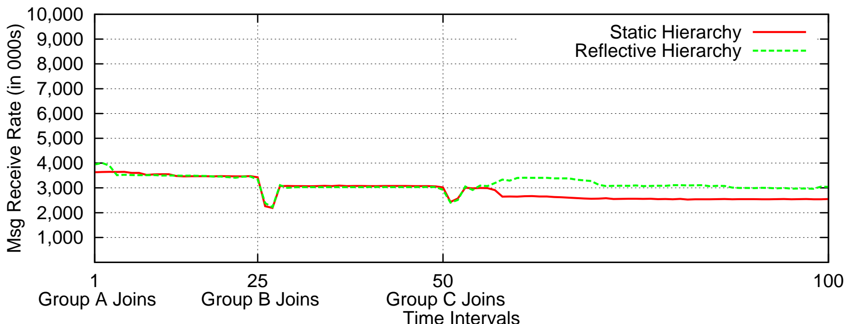


Figure 8.10 *Result of test case OtO-4 (150 publishers and 150 subscribers)*

8.5.1.5 300 Publishers / 300 Subscribers

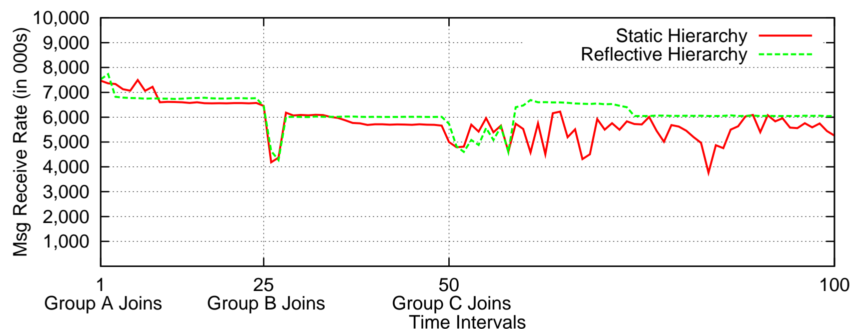


Figure 8.11 *Result of test case OtO-5 (300 publishers and 300 subscribers)*

The largest of the producer/consumer balanced test cases, with over 600 messaging participants, is shown in Figure 8.11. Within this test case, both hierarchies provide a similar performance over the first 50 intervals with a deviation of between 0.37% and 2.94%. Upon the introduction of group C, the reflective hierarchy outperforms by an average of 11.73%. However, this test case reveals more than a simple throughput increase. The static hierarchy has large spikes in the level of service it is able to provide within this environment. During some intervals, the static hierarchy is able to match the performance of the reflective hierarchy, at other intervals it is only able to provide between 65% and 80% (62.27% at interval 83) of the service offered by the reflective hierarchy. In contrast, the reflective hierarchy offers a smooth level of service for the duration of the test case and reaches optimal performance at interval 60.

8.5.1.6 Summary of One-to-One Evaluation

Within the one-to-one test cases, the reflective approach outperforms the static hierarchy in all but one small-scale test case, OtO-1. Performance increases range from 11.73% to 26.96%, a full throughput comparison of these test cases in available in Table 8.4.

Participants	Test Case	Intervals 0-25	Intervals 25-50	Intervals 50-100
3/3	**Static**	6,421,565	4,339,314	10,728,557
	Reflective	7,173,751	4,089,262	6,421,455
	Difference	752,186 (11.71%)	-250,052 (5.76%)	-4,307,102 (40.15%)
30/30	**Static**	90,510,510	81,925,644	144,330,987
	Reflective	89,158,339	82,168,329	183,239,234
	Difference	-1,352,171 (1.49%)	242,685 (0.30%)	38,908,247 (26.96%)
90/90	**Static**	93,883,811	79,926,804	124,898,961
	Reflective	96,346,100	76,178,453	157,441,631
	Difference	2,462,289 (2.62%)	-3,748,351 (4.69%)	32,542,670 (26.06%)
150/150	**Static**	85,614,437	75,660,895	130,639,962
	Reflective	86,007,078	74,561,951	156,617,335
	Difference	392,641 (0.46%)	-1,098,944 (1.45%)	25,977,373 (19.88%)
300/300	**Static**	165,259,616	143,267,485	272,929,447
	Reflective	165,867,999	147,474,961	304,937,015
	Difference	608,383 (0.37%)	4,207,476 (2.94%)	32,007,568 (11.73%)

Table 8.4 *Message receive throughput comparisons for the one-to-one set of test cases*

8.5.2 Few-to-Many Evaluation

The few-to-many test cases evaluate the static and reflective hierarchies in a consumer heavy deployment. The set contains four test cases ranging from 18 to 540 message participants.

8.5.2.1 3 Publishers / 15 Subscribers

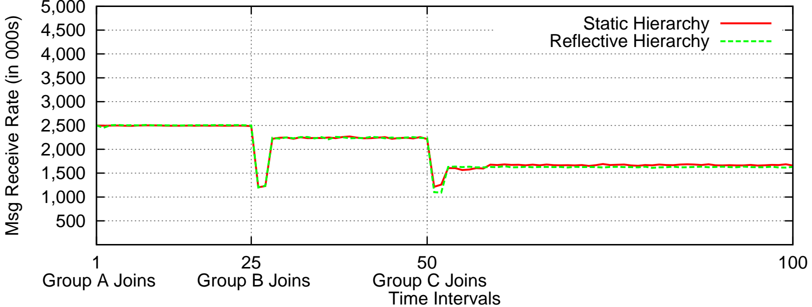


Figure 8.12 *Result of test case FtM-1 (3 publishers and 15 subscribers)*

The first of the consumer heavy environments, illustrated in Figure 8.12, results in a similar throughput for both hierarchies throughout the entire 100 intervals. The effect of group C was

minimal with the static hierarchy slightly outperforming the reflective hierarchy by an average of 2.02%.

8.5.2.2 30 Publishers / 150 Subscribers

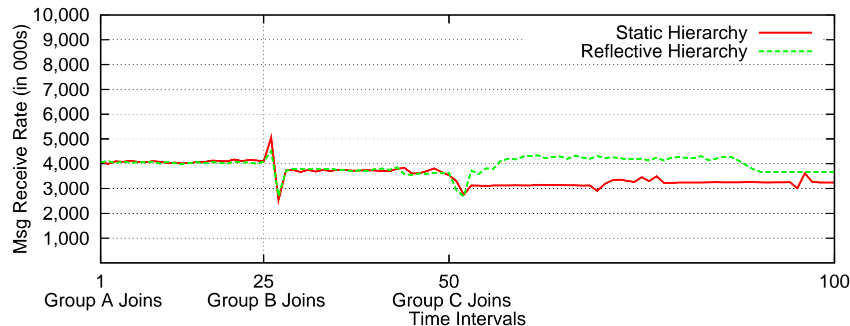


Figure 8.13 *Result of test case FtM-2 (30 publishers and 150 subscribers)*

The second test case in this group, presented in Figure 8.13, shows both hierarchies performing similarly, with less than a 1.17% deviation until group C is introduced. At this point, the reflective hierarchy starts to outperform the static hierarchy by an average of 26.01%.

8.5.2.3 60 Publishers / 300 Subscribers

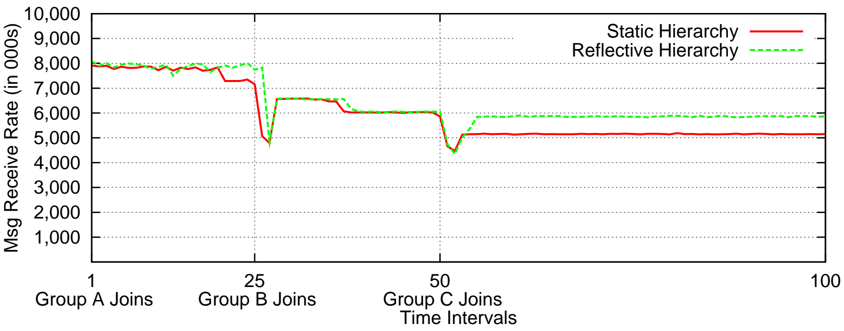


Figure 8.14 *Result of test case FtM-3 (60 publishers and 300 subscribers)*

This test case also follows the pattern from the previous two tests of near identical performance over intervals 0-50, even though the static hierarchy does tail off slightly around interval 20. Upon the introduction of group C participants, as shown in Figure 8.14, the reflective hierarchy provides an average increase of 13.22% in throughput over the static hierarchy.

8.5.2.4 90 Publishers / 450 Subscribers

The result of this 540 participant test case is illustrated in Figure 8.15. Within the test case, both hierarchies perform alike for the first 50 intervals with a small spike for the reflective hierarchy

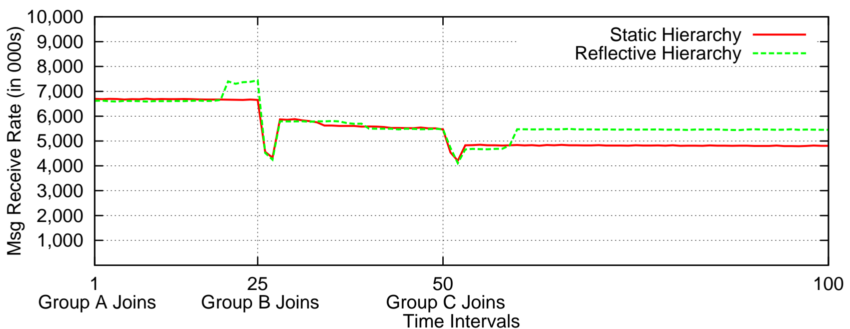


Figure 8.15 *Result of test case FtM-4 (90 publishers and 450 subscribers)*

between intervals 20-25. When group C joins, the reflective hierarchy outperforms the static hierarchy by an average of 11.15%, taking seven intervals to reach optimal performance.

8.5.2.5 Summary of Few-to-Many Evaluation

Participants	Test Case	Intervals 0-25	Intervals 25-50	Intervals 50-100
3/15	**Static**	59,842,913	53,983,997	83,082,569
	Reflective	59,990,406	53,765,752	81,401,873
	Difference	147,493 (0.25%)	-218,245 (0.40%)	-1,680,696 (2.02%)
30/150	**Static**	101,547,580	91,476,470	160,870,337
	Reflective	100,359,240	91,120,011	202,717,099
	Difference	-1,188,340 (1.17%)	-356,459 (0.39%)	41,846,762 (26.01%)
60/300	**Static**	186,905,713	152,224,346	257,563,307
	Reflective	193,873,534	153,497,873	291,600,559
	Difference	6,967,822 (3.73%)	1,273,527 (0.84%)	34,037,253 (13.22%)
90/450	**Static**	162,519,779	138,235,824	240,884,371
	Reflective	164,687,640	138,431,858	267,753,562
	Difference	2,167,862 (1.33%)	196,035 (0.14%)	26,869,191 (11.15%)

Table 8.5 *Message receive throughput comparisons for the few-to-many set of test cases*

In all but one test case in the few-to-many scenarios, the reflective hierarchy outperforms the static hierarchy. In the one test case it under performed, the reflective hierarchy had an average throughput of 2.02% less than the static hierarchy did. In the remainder of the test case, the reflective hierarchy out performed the static hierarchy from 11.15% to 26.01%. A full throughput analysis is presented in Table Table 5 Throughput comparisons for the few-to-many set of .

8.5.3 Many-to-Few Evaluation

In this group of test cases, the converse of the few-to-many test cases are run, creating producer rich environments ranging from 18 to 450 message participants.

8.5.3.1 15 Publishers / 3 Subscribers

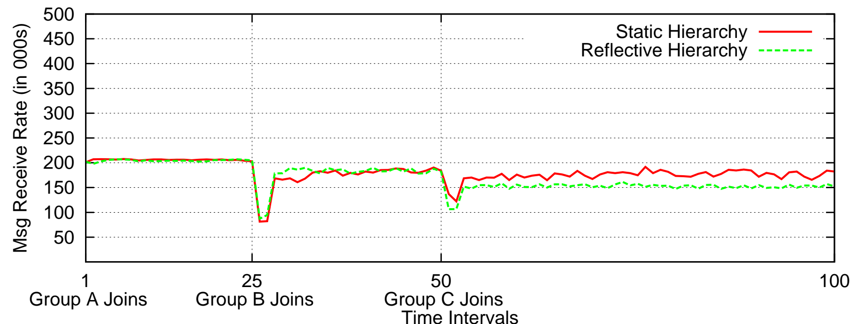


Figure 8.16 *Result of test case MtF-1 (15 publishers and 3 subscribers)*

As shown in Figure 8.16, this is the third test case in which the static hierarchy outperforms the reflective hierarchy. In a similar fashion to the previous tests (OtO-1 and FtM-1), a small number of message participants (less than 20) are involved. Both hierarchies perform similarly with an average deviation of less than 1.57%, until the introduction of group C. At this stage, the static hierarchy has an average improved performance of 13.23%.

8.5.3.2 150 Publishers / 30 Subscribers

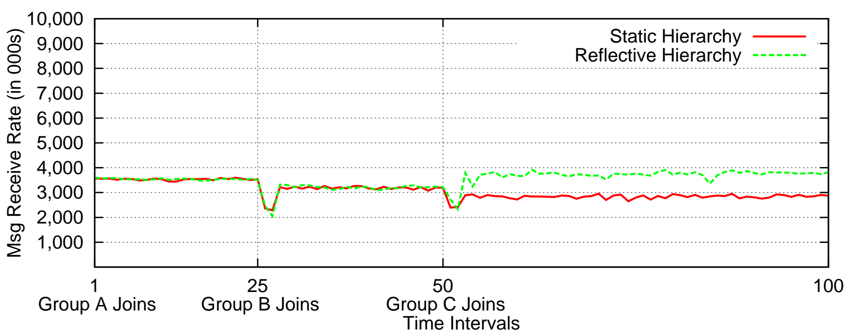


Figure 8.17 *Result of test case MtF-2 (150 publishers and 30 subscribers)*

The next test case, presented in Figure 8.17, produces a more positive result for the reflective hierarchy. Over the first 50 intervals performance between both hierarchies deviated by less that 1.08%. During intervals 50 to 100, the reflective hierarchy has an average improvement of 31.59%, the largest relevant improvement within this case study. The hierarchy quickly adjusts to the demands of group C, reaching optimal operation within three intervals.

8.5.3.3 300 Publishers / 60 Subscribers

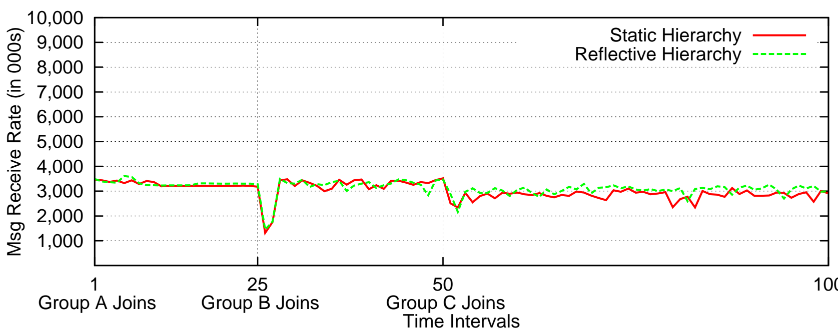


Figure 8.18 *Result of test case MtF-3 (300 publishers and 60 subscribers)*

The outcome of this test case, illustrated in Figure 8.18, is much closer than the previous test with only an average performance difference of 1.52% and 0.32% over the first two stages respectively. The smoothness of the level of service provided by both hierarchies is slightly erratic once group B joins the test case, once group C joins the reflective hierarchy outperforming the static hierarchy by an average of 7.12%.

8.5.3.4 450 Publishers / 90 Subscribers

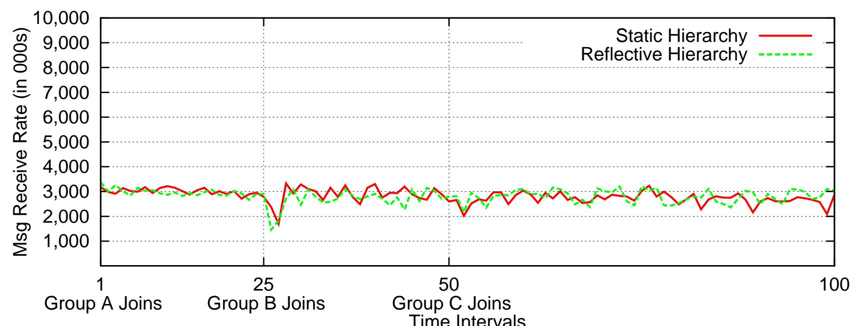


Figure 8.19 *Result of test case MtF-4 (450 publishers and 90 subscribers)*

The final test case in this case study, shown in Figure 8.19, is a near dead heat between both hierarchies. The static hierarchy is slightly superior over the first 50 intervals with an average performance improvement of 2.22% for intervals 0-25 and 5.13% for intervals 25-50. Over the remaining 50 intervals, the reflective hierarchy outperforms the static hierarchy by an average of 3.54%.

8.5.3.5 Summary of Many-to-Few Evaluation

When examining all the test cases used in this case study, the many-to-few scenarios are least successful for the reflective hierarchy. However, it still outperformed the static hierarchy twice and

Participants	Test Case	Intervals 0-25	Intervals 25-50	Intervals 50-100
15/3	**Static**	4,897,920	4,376,909	8,815,674
	Reflective	4,880,007	4,445,800	7,649,118
	Difference	-17,913 (-0.37%)	68,891 (1.57%)	-1,166,556 (-13.23%)
150/30	**Static**	85,939,886	78,170,427	142,083,289
	Reflective	85,919,309	79,012,941	186,965,072
	Difference	-20,577 (-0.02%)	842,514 (1.08%)	44,881,784 (31.59%)
300/60	**Static**	78,485,420	81,181,004	142,494,435
	Reflective	79,674,712	80,924,279	152,644,509
	Difference	1,189,292 (1.52%)	-256,725 (-0.32%)	10,150,074 (7.12%)
450/90	**Static**	73,193,679	72,683,700	136,281,747
	Reflective	71,568,068	68,954,010	141,108,670
	Difference	-1,625,612 (-2.22%)	-3,729,690 (-5.13%)	4,826,923 (3.54%)

Table 8.6 *Message receive throughput comparisons for the many-to-few set of test cases*

matched the performance of the static hierarchy to within 0.19% in the last test in the group. In only one test case the static hierarchy significantly outperformed the reflective hierarchy, while test case MtF-2 produced the biggest relative performance increase of any test case within the case study with a 31.59% improvement over the last 50 intervals. A full throughput comparison for the many-to-few test cases is available in Table 8.6.

8.5.4 Evaluation Summary and Discussion

A summary of the throughput figures for the test cases is provided in Table 8.7. The largest test case within the case study is FtM-3 with over 638,971,967 messages sent using the reflective hierarchy, an improvement of 42,278,602 messages, or 7.09%. The largest improvement was in MtF-2 with a throughput improvement of 45,703,721 message or 14.93% over the duration of the test case. Even though the many-to-few benchmarks experienced the largest improvement, these test cases also experienced the lowest relative throughput increase within the case study; this is due to the producer-centric nature of the test cases. In only two test cases, the static hierarchy significantly (more than 1% performance difference) outperformed the reflective hierarchy. In eight test cases, the reflective hierarchy had a performance improvement of greater that 5%.

The main factor that improves the performance of the reflective hierarchy is the removal of the need for filter evaluation. This act normalises the effort required to deliver messages within each group, improving the overall scalability of the deployment. This trend is apparent upon examination of the group timeline. Within the static hierarchy test cases, messages from each group are not delivered equally due to the filtering evaluation required. Examination of the reflective hierarchy test cases show an equal delivery of messages from each interest group, this is achieved by simplifying the delivery of messages from group C (removing the need for constraint evaluation)

Test Case ID	Publishers	Subscribers	Static Hierarchy Throughput	Reflective Hierarchy Throughput	Reflective /Static	Reflective /Static (percentage)
One-to-One						
OtO-1	3	3	21,489,436	17,684,468	-3,804,968	-17.71%
OtO-2	30	30	316,767,141	354,565,902	37,798,761	11.93%
OtO-3	90	90	298,709,576	329,966,184	31,256,608	10.46%
OtO-4	150	150	291,915,294	317,186,363	25,271,070	8.66%
OtO-5	300	300	581,456,548	618,279,975	36,823,426	6.33%
Few-to-Many						
FtM-1	3	15	196,909,479	195,158,031	-1,751,448	-0.89%
FtM-2	30	150	353,894,387	394,196,350	40,301,963	11.39%
FtM-3	60	300	596,693,365	638,971,967	42,278,602	7.09%
FtM-4	90	450	541,639,973	570,873,060	29,233,087	5.40%
Many-to-Few						
MtF-1	15	3	18,090,503	16,974,925	-1,115,578	-6.17%
MtF-2	150	30	306,193,601	351,897,322	45,703,721	14.93%
MtF-3	300	60	302,160,859	313,243,501	11,082,641	3.67%
MtF-4	450	90	282,159,126	281,630,748	-528,378	-0.19%

Table 8.7 *Summary of case study throughput analysis*

allowing the MOM to deliver more messages. Further results on the level of throughput from individual groups within the test cases are available in Appendix C.

The time it takes for the reflective solution to reach its optimum performance depends on a number of factors. These include the time for the adaptation to take place, time for consumers and producers to update their view of the hierarchy, and the time needed to clear messages currently within the system.

In three test cases, the static hierarchy outperformed the reflective hierarchy. One reason for the poor performance of the reflective hierarchy in these test cases may be attributed to the overhead of the self-management framework. At low levels of demand, self-management techniques may not be able to improve performance enough to cover associated overheads. In the three test cases in which the static hierarchy outperforms the reflective hierarchy, less that 18 message participants are involved:

- OtO-1, 6 participants, throughput comparison for RDH –17.71%
- FtM-1, 18 participants, throughput comparison for RDH –0.89%
- MtF-1, 18 participants, throughput comparison for RDH –6.17%

The MtF-1 test case indicates the difficulty in obtaining performance improvements within

producer-heavy environments. Given that fewer consumers exist within these deployments, there is less opportunity to increase performance by optimising filtering. This is backed up by the fact that the many-to-few scenarios are least successful for the RDH.

A major influence in determining the breakeven point for a RDH is the MOM provider used for the base-level. Given that each MOM implements filtering and routing services in a proprietary manner, the potential performance improvements of a RDH will be specific to each MOM. As such, the breakeven point for each MOM will vary accordingly.

Based on these results an estimated RDH breakeven point exists in the region of 20 message participants for the one-to-one and few-to-many deployments, with a slightly higher breakeven point within the many-to-few deployments.

8.6 Summary

The purpose of this case study was to examine the benefits of self-management capabilities for MOM and to support the case for the coordination of self-managed systems. The case study recreates a typical information dissemination service implemented with both a traditional and reflective destination hierarchy. The empirical evaluation of the benchmarking process reveals the reflective hierarchy to outperform the static hierarchy in 9 of the 14 test cases, with 2 test cases producing dead heats[1], demonstrating the benefits of both coordinated self-managed systems and the benefits of reflective techniques within the MOM domain.

Apart from the empirical benefits of reflective hierarchies, an additional benefit of reflective messaging solutions is their ability to adapt to current operating conditions. The case study demonstrated this capability with an existing hierarchy; however, it could also grow a hierarchy. Static destination hierarchies must predefine criteria for every potential deployment scenario in advance. A reflective destination hierarchy allows message participants to choose the criteria, allowing a hierarchy to evolve (or grow) and constantly adapt into one customised specifically for the deployment environment. Starting from a single destination (seed) a customised destination hierarchy (tree) can grow based on the expressed requirements of the environment; relieving the need to predefine the hierarchy.

[1] Less than 1% performance difference.

Chapter 9

Conclusions

If the vision of autonomic computing [6] is to be realised, the need for increased coordination between self-managed systems is a fundamental prerequisite. Middleware platforms may provide different levels of service depending on environmental conditions, resource availability, and costs. John Donne said 'No man is an island'. Likewise, no self-managed middleware platform, service or component is an island and each must be aware of both the individual consequences and group consequences of its actions [134]. Next-generation middleware systems must coordinate/cooperate with each other to maximise the available resources to meet the requirements encountered.

9.1 Thesis Summary

The focus of this research was to investigate the utility of coordinated behaviour within the domain of self-managed middleware. Message-Oriented Middleware (MOM) was chosen as the problem domain to investigate this hypothesis; MOM is interaction-centric making it an in ideal platform. Reflective self-managed techniques have yet to be utilised within the MOM domain, creating an additional research theme; investigating reflective self-management techniques within MOM.

The research commenced with an examination of current reflective self-managed systems to highlight their capabilities, limitations, and design with respect to coordination. The examination revealed that current state-of-the-art self-managed systems do not provide full coordination capabilities in an implementation agnostic, openly accessible manner.

The problem domain of MOM was examined to reveal its rudimentary theories and highlight its differences from traditional distribution mechanisms. A number of MOM implementations were scrutinised to highlight the diversity of the problem domain and to reveal any reflective self-managed capabilities. The analysis revealed limited self-management capabilities within the MOM domain.

With the motivation and background of the work in place, the next step was to outline the prerequisites needed for coordinated self-managed systems. Following these guidelines the Open Meta-level Interaction Protocol (OMIP) was developed to provide a mechanism to allow meta-level interaction.

The next stage in this work was to provide a platform to investigate coordinated meta-levels. To this end, the GenerIc Self-management for Message-Oriented middleware (GISMO) was defined for the study of meta-level coordination within self-managed systems. A key step in the defini-

tion of GISMO is the identification of common MOM characteristics (participants, behaviour, and state) for inclusion. Once the design of GISMO was in place, a prototype implementation was developed using the Chameleon framework. Chameleon provides non-invasive techniques to augment functionality onto a Java Message Service compliant MOM using interception capabilities.

With the development of the infrastructure complete, an extensive evaluation was required to assess the benefits of coordination. A comprehensive benchmarking evaluation process was run on a private network of 12 machines with the execution of 88 benchmark tests taking a combined total of more than 85 hours of benchmarking time. The objective of the evaluation process was broken down into two goals, evaluate coordination between self-managed systems, and evaluate reflective self-managed techniques within MOM.

The first case study evaluated the benefit of coordination between self-managed systems by examining the benefits of information exchange between interacting participants. The scenario used in the case study is similar to the motivational scenario presented in Chapter 1. Within this case study, a centralised routing solution is decentralised by exchanging management information (routing rules) between interacting message participants. The key mechanism needed in the decentralisation of the routing solution is the ability to exchange management information (meta-information) between systems in an open manner. With such ability in place, the benchmarks clearly show the advantages of coordination between self-managed systems and exemplify its potential to foster the development of innovative message solutions within the MOM domain.

The second case study used to evaluate this research examined the benefits of reflective self-management techniques for MOM within dynamic environments. Reflective techniques enable the MOM to alter its runtime configuration to match the current demands of its environment. In this case study, benchmarks of both reflective and non-reflective MOMs were run within a simulator that recreates dynamic messaging environments. The execution of these benchmarks revealed that self-management techniques could have a considerable affect on the performance of a MOM provider, increasing the level of service provided by the MOM within dynamic environments.

The evaluation and benchmarking process is a clear validation of the benefit of coordination between self-managed systems. *Within dynamic operating environments, coordinated interaction between self-managed systems can improve the ability of the individual and collective systems to fulfil performance and autonomy requirements of the environment.*

9.2 Contributions

This thesis addresses the lack of coordination between self-managed middleware systems. The principal contributions of this work can be broken down along the following lines.

9.2.1 A Self-Managed MOM

A suitable platform was required to evaluate the research hypothesis. Given that no fully coordinated self-managed platform currently existed, the decision was taken to develop one within the MOM domain. The choice of problem domain for this platform provided the opportunity to investigate self-management techniques within the MOM domain. To this end, the GenerIc Self-management for Message-Oriented middleware (GISMO) was defined. The design of this generic meta-level encompasses common MOM characteristics to make it applicable for multiple

MOM implementations. The implementation of GISMO is achieved with Chameleon, a lightweight framework to allow non-invasive augmentation of the meta-level onto multiple base-levels. Benchmarks of GISMO illustrate the performance enhancement self-management techniques can provide for MOM.

9.2.2 Coordination between Self-Managed Systems

The primary focus of this work was the investigation of the benefits of coordination between self-managed systems. This was achieved by defining a protocol to facilitate access to self-management services (state, adaptive capability, and analysis capacity) in a generic manner. A minimal set of generic prerequisites to facilitate open interaction were identified. These required any interaction protocol to maintain a participants independence and provide open accessibility and extensible interaction in an implementation agnostic manner.

The central contribution was the definition of the Open Meta-level Interaction Protocol (OMIP) that satisfies the requirements for meta-level interaction. This protocol defines a number of generic interaction commands containing an associated message, expressed in a domain specific language, used to describe the application/domain specific details of the request (i.e. security, hardware, multimedia, telecoms, flight control, education, UI, etc.).

With a self-managed MOM supporting OMIP in place, coordinated self-managed solutions could be developed to investigate the benefits of coordination. To this end, two solutions were created to examine the capabilities of coordinated interaction and the benefits of self-management techniques within the MOM domain. These solutions show how coordinated self-management can play an important role in the development of new messaging solutions and the enhancement of current messaging solutions. An extensive benchmarking was performed to verify and validate the benefits of the coordination techniques, with all benchmarks performed under conditions comparable to, or better than, standard industrial practices.

9.2.3 Additional Contributions

A number of additional contributions were also made in the design and benchmarking of self-managed systems.

9.2.3.1 M-SAR Design Pattern

One of the key contributions was in the area of self-managed system design with the identification of a design pattern for the development of portable meta-levels. The Meta-State-Analysis-Realisation (M-SAR) design pattern proposes a separation of concerns within a meta-level by insolating the three main roles of a meta-level, information, examination, and realisation into distinct encapsulated entities, promoting a clearer separation of concerns within a meta-level.

9.2.3.2 Dynamic Benchmarking

The empirical evaluation process also produced contributions within the area of benchmarking MOM platforms and recreating dynamic environments. These contributions were the identification of requirements to benchmark self-managed systems and the design of a dynamic test case to benchmark self-managed MOM systems within dynamic environments. With these in place,

a MOM benchmarking suite was developed to provide a unique simulator to recreate dynamic messaging environments.

9.3 Future Research Directions

There is a wide range of possible research avenues for this work. Future research directions are explored along two lines, technology transfer of current contributions and research opportunities in which coordinated self-managed systems may continue to be explored.

9.3.1 Technology Transfer

The success of transferring contributions from this research to common practice will be one of the metrics used to gauge the success of this work in years to come. The relevance and usefulness of these contributions within a real world deployment will be the true measurement of their value. This section highlights some possible efforts to initiate a technology transfer.

Generally speaking, one of the main obstacles when introducing a new technology into a production environment is the level of interaction the technology requires with current system assets. Non-functional concerns, such as under-the-hood performance optimisations, are simpler to introduce than a functional concern such as a user rating service that requires extensive participant interaction. Introducing an interaction-oriented technology requires the assets currently deployed within the environment to be updated to utilise the new technology: potentially a very expensive prospect. With this in mind, potential technology transfer is broken down along the functional and non-functional lines.

9.3.1.1 Functional

Functional contributions of this research include coordinated meta-levels and novel routing solutions; in order for these to be deployed within a production environment, they will require the redeployment of participating entities. However, the possibility for complete redeployment within large-scale deployments is limited. Within such environments, an incremental approach to the deployment of these services is required, with the safe co-existence of both technologies critical to the success of the process. The responsibility of ensuring co-existence falls on the self-managed systems. These systems must ensure that their adaptations do not interfere with the operation of other systems that rely on them or systems with which they interact.

To this end, a phased rollout of coordinated self-managed systems should be used to introduce the technology slowly into the production environment. Once a significant number of clients possess interaction capabilities, enhanced routing solutions may be introduced into the environment along side the current routing solutions. Clients may then be migrated from the old to the new routing solutions, thus introducing the technology in a controlled, phased manner. Given the need for legacy support within productions environments it may take some time to phase out the older technology, if indeed that is possible within the deployment.

9.3.1.2 Non-Functional

Fewer obstacles exist for the transfer of non-functional aspects of this research into production environments. Non-functional contributions range from the design pattern used in the construction

of self-managed systems to a meta-level for self-managed MOM. While there is great potential for the use of non-functional concerns within production environments, especially when one considers there is no need for a large-scale redeployment, obstacles still exist with their inclusion.

Deployment of self-managed systems into production mission critical environments will require these systems to reach a level of maturity where system administrators feel comfortable with such platforms in their environment. Of utmost importance to reaching this goal is the safe adaptation of the system with predictable results in the systems behaviour. The current practices used for software testing and quality assurance are inadequate for self-managed systems. In order to gain acceptance as a deployable technology, it is important for the research community to develop the necessary practices and procedures to test these systems to ensure they perform predictably. Such mechanisms will promote confidence in the technology.

9.3.2 Research Opportunities

Given the broad scope of this research, a number of interesting research opportunities have been identified to extend this work. Six key directions for future investigation are now examined:

9.3.2.1 Standards Development

In order to further develop the concept of the Open Meta-level Interaction Protocol (OMIP), a number of standards will need to be developed, including definitions for OMIP message encoding formats, transportation protocols, Domain Specific Language (DSL) message formats, and the definition of DSLs and relevant Interaction Commands (IC) for each domain

If any standardisation effort is to be successful, the self-managed (including adaptive and reflective) community needs to participate in the formation of an international group to develop such standards in an open collaborative environment. Any such group should consist of a mixed representation of international companies and academic institutions to reach a balanced common specification.

9.3.2.2 Marketplaces / Resource Trading

With coordinated interaction capabilities in place, a number of interesting research possibilities are exposed in the area of resource marketplaces. Within these marketplaces, resources may be traded in a number of ways from simple barter between two services to complex auctions with multiple participants, each with their own tradable resource budget, competing for the available resource.

In addition to the development of relevant negotiation protocols, trading participants also need to understand the commodities they are trading. This will require a method of defining a resource, its capabilities, and an assurance of the quality of service offered. Once a trade is finalised, enforceable contracts are needed to ensure compliance with the trade agreement. This concept of resource trading could be extended across organisational boundaries with the trading of unused or surplus resources in exchange for monetary reimbursement

9.3.2.3 Self-Protection

One of the main goals of this research is to open up self-managed systems to interact with other participants within their environment. The scope of this research has only considered friendly envi-

ronments where all systems strive for a mutually beneficial outcome for all participants. However, self-managed systems will face both friendly and hostile environments within real-world deployments. Hostile environments can come in many flavours, with participants that interact in a purely selfish manner to participants that perform Denial-of-Service (DoS) attacks. Any system that opens its self-management process will be vulnerable to such attacks. In a similar fashion to autonomic systems [6], coordinated self-managed system will need to have appropriate self-protection capabilities to cope with attacks and use appropriate countermeasures to defeat or at least nullify the attack within these environments.

9.3.2.4 Reinforcement Learning

The investigation of self-management MOM techniques within this work revealed their benefit within the MOM domain. The reflective computation used within this work utilised a straightforward algorithm to adapt the MOM to suit runtime requirements. Reflective computation of this nature could be enhanced with more intelligence to improve the accuracy of its outcome. One technique that merits investigation is the use of reinforcement learning within self-managed platforms. Reinforcement learning is the problem faced by an agent that learns behaviour through trial-and-error interactions within a dynamic environment. While some promising work [27, 38] has already been carried out with these techniques, many opportunities exist for the investigation of these techniques within the MOM domain.

9.3.2.5 Application-Level Messaging Semantics

An open area of MOM research is the exchange of messaging semantics between producers and consumers. Currently this exchange is performed at the developer-level with the exchange occurring between the developers of the message consumers and producers. The next step in messaging semantics is the autonomous exchange of information between participants. Much work is underway on the development of semantically enhanced message integration. To this end, GISMO could be extended to include additional messaging semantics.

This new sub-model would contain information on application-level messaging semantics. The model would allow a client unfamiliar with an environment to dynamically discover the messaging semantics of a destination. Information such as the format of the message payload, the properties used to tag messages, and the interaction model used by the destination such as a Request-Response or Inform (one-way | one-shot) protocol. Much work has been done on this form of semantic information exchange within the agent-oriented community [107, 109, 135, 136] and some effort has been made to introduce these techniques into the MOM domain by Cilia et al, using semantics to simplify integration issues [137].

9.3.2.6 Broker Meta-Model

Broker networks are in common use within large and wide-area MOM deployments. A number of the MOM implementations reviewed in Chapter 3 are deployable as a network of brokers. Broker networks are vital to the development of massively scaled wide-area notification services. Many research opportunities exist within these networks and provide ideal candidates for the investigation of self-management capabilities with the inclusion of a broker meta-level within GISMO.

The broker model would contain information on broker networks for managing both central and federated services. The model would contain information on the relationships and inter-connections between brokers such as master/slave, client/server or peer-to-peer. An event model could also be used to trigger events when certain conditions occur within the network, potential triggers include broker additions and changes in broker routing tables. The large number of MOM platforms that may be deployed as a network of brokers [62, 70, 74, 83], and the development of self-organising broker networks [105, 106], support the development of the broker meta-level.

Part IV

Appendices

Appendix A

An Extensible MOM Test Suite

This appendix describes the benchmarking process used to measure the performance of the messaging solutions within this research. The process builds on current approaches to benchmarking messaging systems with additional techniques to benchmarking dynamic environments.

A.1 Introduction

Planning and executing appropriate benchmarks for a distributed messaging infrastructure is a complex time-consuming, labour-intensive process. Pushing a messaging infrastructure to its limits is vital to understand how it will perform under a heavy workload with large numbers of message participants. Benchmarking for a sufficiently long duration with appropriate numbers of senders and receivers can establish the limits of the messaging infrastructures scalability within an environment.

The benchmark techniques within this work are forged by combining the strengths of a number of academic and industrial benchmark practices, reports, and test suites [92, 130, 131, 132, 133, 138, 139]. These techniques were then extended with a number of enhancements to increase the range and diversity of possible test environments. This appendix covers a number of aspects of the benchmark process including test case design and testbed configuration, the software test suite architecture, and the development of techniques to benchmark dynamic environments.

A.2 Test Case Design

Within any benchmarking process, the design of the test cases is a key factor affecting the endeavour's success. When designing a test case, a number of aspects must be considered to create a profile that accurately reflects a realistic deployment environment and recreates actual operating conditions. This section introduces benchmarking within the MOM domain and discusses the factors that affect the benchmarking process, including messaging models, producer/consumer ratios, configuration, reporting metrics, and the test case timeline.

A.2.1 Messaging Models

Two main message models are commonly available with MOM implementations, the Point-to-Point, and Publish/Subscribe models. From a benchmarking perspective, one must consider the message delivery ratios that exist within both models.

When benchmarking the point-to-point model it is important to remember that each message is delivered only once to only one receiver, known as *'once-and-once-only'* message delivery. The model allows multiple receivers to connect to the queue but only one of the receivers will consume the message.

The publish/subscribe messaging model is a very powerful one-to-many and many-to-many distribution mechanism, allowing a single producer to send a message to one user or potentially hundreds of thousands of consumers. Clients producing messages 'publish' to a specific topic, these topics are then 'subscribed' to by clients wishing to consume messages. The service routes the messages to consumers based on the topics to which they have subscribed. When benchmarking the publish/subscribe model, the number of subscribing clients has a significant impact on the overall performance of the MOM. Where multiple consumers have the same subscription, any message that matches the subscription will be distributed to each of the subscribers.

In summary:

- Point-to-Point - *One message is sent, one message is received.*
- Publish/Subscribe - *One message is sent, but multiple messages may be received*

Once the messaging model is chosen, the next step is to decide the number and ratio of message producers and consumers within the test case.

A.2.2 Producer/Consumer Ratio

It is vital that a realistic benchmark reflects an actual deployment environment. The number and ratio of message producers and consumers dictate the scalability tested within the test case. The more message participants used, the higher the level of scalability the test case will attempt to recreate. Another important factor to consider is the ratio between producers and consumers.

It is common for messages to be produced faster than they can be consumed. In such a scenario, message congestion can occur at the server and it is important to test the server's ability to handle the build-up of messages in its queues. As a means of replicating such events, it is possible to alter the ratio of message producers to message consumers, enabling the throttling of message production and consummation rates. A summary of possible ratios and their affect is available in Table A.1. These producer/consumer ratios create the same effect when used within both the point-to-point and publish/subscribe messaging models.

A.2.3 Configuration

When comparing MOM implementations the configuration of message producers and consumers will influence the outcome of the test. The configuration of message persistence, message filters, and message acknowledgement modes can drastically affect the results of a test. Each of these settings will test the MOM implementation in a different manner and care should be taken to ensure the correct settings are chosen to expose the goal of the test. One configuration option that heavily influences a test case is the delivery mode used for messages.

Producer/Consumer Ratio	Effect of Ratio
One-to-One	Tests the ability of a server to handle large numbers of connections.
Few-to-Many	Reduces the likelihood of queuing on the server.
Many-to-Few	Increases the likelihood of queuing on the server.
Many-to-Many	Same effect as one-to-one.

Table A.1 *The effects of message producer/consumer ratios within benchmarks*

A.2.3.1 Delivery Modes

Delivery modes dictate the reliability of a messaging solution in the event of a server crash. When a message is marked as persistent, the MOM is responsible for writing the message to a non-volatile storage medium, such as a hard disk, before acknowledging its receipt, enabling it to retrieve the message in the event of a crash. This task can drastically affect the throughput rate of a MOM. The performance of the underlying hardware (CPU, memory bus, and persistent store) will often be the bottleneck that limits the number of messages processed. Regardless of the efficiency of the persistence mechanism, the speed of the physical medium (i.e. hard disk) is critical in persistent message test cases [92, 131].

A.2.4 Reporting Metrics

When performing benchmarks, it is important to clearly define the objective of the test and measure the correct metric to quantify the objective. Common metrics within MOM benchmarking tests are described in Table A.2.

Metric	Description
Message Throughput Rate	The total number of messages received per time interval (i.e. second, minute, etc.) by all message consumers during the tests measurement window. Common time intervals for message rates is message throughput per second or msg/sec.
Size Throughput Rate	The total size of messages received per time interval (i.e. second, minute, etc.) by all message consumers during the tests measurement window. A common time interval for size rates is byte throughput per second or byte/sec.
Number of Producers/Consumers	The number of connected message producers/consumers in the test.

Table A.2 *Possible benchmark reporting metrics*

Message throughput rate is the primary measurement used in MOM benchmark tests. Unless explicitly stated, a message rate is always a received rate (number of messages delivered to and received by consumers) and not the send rate (the rate that producers are able to send messages to queues/topics).

Another metric used within MOM benchmarks is size throughput rate. This metric is appropriate for benchmarking the persistence mechanisms of a MOM and may also be used to test the ability of the MOM to handle large message sizes.

Within a test case, the numbers of message producers/consumers tests the scalability of the MOM, the more producers/consumers the greater the load. Given that the number of participants within the test case will affect the message rate, it is common to use a combination of these metrics to reveal different aspects of the MOMs performance. Some popular groupings include:

- Message Rate / Number of Consumers
- Message Rate / Number of Producers

Once an appropriate metric is chosen, the next task is to define how the test case will be run. Important factors include the duration of the test case, the sampling rate of the metric, and recreation of a realistic deployment scenario.

A.2.5 Timeline

In order to reach and evaluate the constant throughput level of the test case, it is necessary to take multiple measurements over the duration of the benchmark. The longer a test is run for the more likely it is to expose potential problems within the MOM including message congestion, memory leaks, server stability, and server stress. Given the long-running nature of MOM platforms, these problems are real and it is important to include them within the results of a benchmark.

Current benchmarks efforts have run tests cases for durations of 5 minutes [138] and 15 minutes [92]. It is important that any test case is performed for a period of time that allows the benchmark to reach this constant steady-state. If the test case is too short this steady-state will not be reached.

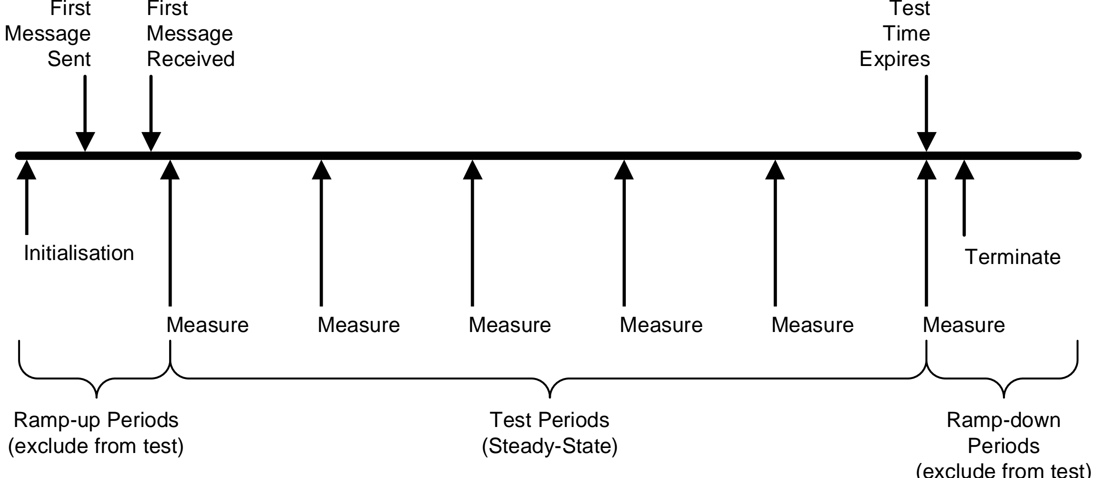


Figure A.1 *Test case timeline*

A sample test case timeline is illustrated in Figure A.1. In this timeline, ramp-up and ramp-down periods are used to ensure that the steady-state of the test is reached before test measurements are taken. Ramping periods are a common practice [92, 131, 138, 139] within MOM

benchmarking that ensure unpredictable behaviour (initialisation, object creation, message ramp-up) during setup and shutdown periods do not contaminate test results.

A.3 Testbed Configuration

The testbed is fundamental to any benchmark process. It is vital that a testbed accurately reflects a live deployment environment. At a minimum, message producers and message consumers, and the message server, should be located on different machines. Running these test participants on a single machine will not reveal any variations in the test results that occur due to the network. Distributing the test case over a number of machines is vital to expose potential bottlenecks such as network congestion, persistence overheads, and memory leaks that may not be apparent when a benchmark is run on a single machine. In addition, spanning the benchmark across a number of networked machines more accurately reflects an actual deployment environment.

The deployment infrastructure utilised for benchmarks within this work is illustrated in Figure A.2. Within this deployment, test case message producers and consumers are distributed over 10 client machines, with messages exchanged through a MOM located on a separate server machine. This configuration ensures that messages must travel over the network to be exchanged between message producers and consumers.

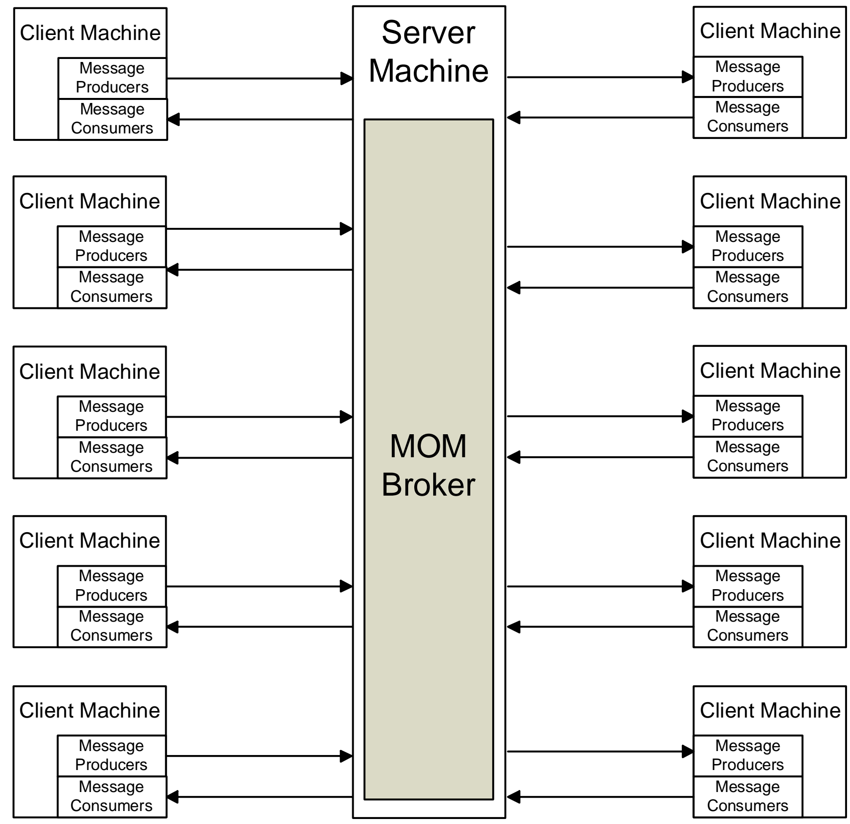


Figure A.2 *The testbed deployment*

A.3.1 Hardware

The hardware and software specifications of the machines within the testbed are detailed in Table A.3. Excluding benchmarks using only one message producer and one message consumer, all test cases were run with message producers and consumers evenly deployed across the 10 client machines. Producers and consumers deployed on the same client machine share the same JVM but are run under different threads and use independent connections.

	Server (x1)	**Clients (x10)**
CPU	Pentium 4 2.4 GHz	Pentium 4 2.4 GHz
RAM	512MB	512MB
Bus Type	EIDE	EIDE
Network	100Mbits/s Adapter running at 100mb in full-duplex mode	100Mbits/s Adapter running at 100mb in full-duplex mode
OS	Microsoft Windows 2000 Service Pack 4	Microsoft Windows 2000 Service Pack 4
JavaVM	Sun 1.4.2_06	Sun 1.4.2_06
JMS Providers	Chapter 7 case study: SonicMQ 6.1 Chapter 8 case study: ActiveMQ 1.3	Chapter 7 case study: SonicMQ 6.1 Chapter 8 case study: ActiveMQ 1.3

Table A.3 *Testbed hardware and software specifications*

A.3.2 Network

The network used in the testbed is a 100Base-T network. A network of this capacity reduces the likelihood of the network forming an artificial bottleneck, such as one caused by a slower 10Base-T network. The network is private and does not contain any additional traffic that may affect the execution of the test cases. The network HUB used was an Extreme Networks Summit 48[1].

A.4 Default Test Cases Setup

This section details the default setup for all the test cases carried out in this work.

A.4.1 Test Conditions

All benchmarks were carried out under the following test conditions:

- Each client was run in a separate JMS connection.
- Each test includes a 2-minute ramp-up and a 2-minute ramp-down period that are not included in the benchmark measurement period.
- All client connections are established and message consumer and producer (Senders/Receivers – Publishers/Subscribers) objects are created before the ramp-up period begins.

[1] http://www.extremenetworks.com/

- Unless explicitly stated, each measurement window was 30 minutes in duration, with measurements taken every 60 seconds.
- Performance was measured under maximum load by sending as many messages as possible using default settings on both the client and server.
  - All message consumers used auto-acknowledgement and asynchronous message receipt.
  - Messages were non-persistent to ensure the persistence mechanism is not a bottleneck for the benchmark.
- During the test no other applications were running. This reduces the 'noise' of a varying CPU.
- Before each test the following housekeeping chores were performed to ensure each test started with a clean slate and did not include any overhead or residue from a previous test.
  - The server was rebooted.
  - All client machines were rebooted.
  - The MOM message store was emptied.
  - Queues and topics were emptied, deleted, and recreated.
- Message producers produce messages as quickly as possible, there is no flow control on the client-side between sent messages.

A.4.2 Desired Metrics

This research focuses on the improvements to the routing capability of a MOM. Since the throughput of a MOM is often limited by its persistence mechanism, the message size throughput rate is an inappropriate metric.

The most appropriate metric for the benchmarking tests performed is the message throughput rate combined with the number of participants. This metric measures the throughput and scalability of the MOM.

A.5 Software – Extensible Mom Test Suite (EMiTS)

The Extensible Mom Test Suite (EMiTS) is designed to evaluate JMS compatible MOM implementations across a collection of networked machines. EMiTS has a number of advances over current benchmarking suites, including centralised configuration and reporting mechanisms, and streamlined management of test case execution across multi-machine testbeds. EMiTS also possesses capabilities to recreate dynamic test case conditions to simulate dynamic messaging environments, a capability vital to evaluate reflective self-managed MOM implementations.

The EMiTS test suite may be extended to test diverse messaging operations and capabilities with the use of extensible drivers for message producers and consumers. The remainder of this section provides a brief overview of the EMiTS architecture, highlighting its design and configuration options available for test cases.

A.5.1 Architecture

The architecture for EMiTS is broken down into a client and server. The server is responsible for the coordination and management tasks associated with test case setup, and the aggregation of client results once the test is complete. The client is responsible for the execution of message participants and for recording their activity over the duration of the test case. The architecture for EMiTS is illustrated in Figure A.3.

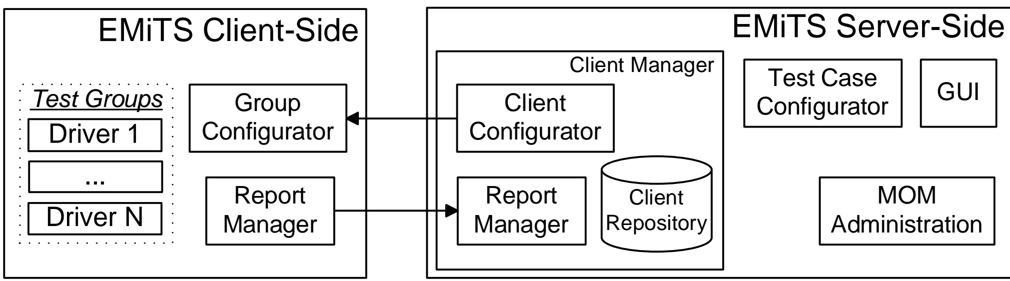


Figure A.3 *EMiTS architecture*

A.5.1.1 Server

The EMiTS server is primarily concerned with test case configuration and the collection and aggregation of results once the test case has finished execution. Configuration of the test case may be directed via the GUI and/or an XML configuration file. The server contains a client manager to track client machines within the test case. The client manager has responsibility for informing each client as to their role within the test case by transferring configuration information for their test groups. Once test case execution has completed, clients return results to the report manager for aggregation into a test case report. The last aspect of the server is the MOM administrator with responsibilities for configuring the MOM provider for the test case, including destinations and persistence requirements.

A.5.1.2 Client

The client of the test suite is responsible for the execution of test groups within the test case. Once the test case configuration has been received from the server, the client must setup the groups assigned. Each test group uses a driver to direct its messaging activity over the duration of the test case. These drivers allow the extension of the test suite to test a variety of messaging functionality. Once the test case has completed executed the results from the groups are sent to the server where they are aggregated into a test case report.

Three groups of configurations are used to direct the test case. The first set of options, discussed in Table A.4, details the type of connections used by message participants to connect to the MOM.

The second group, detailed in Table A.5, specifies general test case settings including the number of test intervals, ramp-up intervals, and the drivers used by producers and consumers.

The last collection of settings directs the configuration of the test groups within the test case. These settings are used to configure the client drivers.

Property	Description	Sample Value
Broker URL	The URL used by clients to connect to the broker within the test case.	tcp://localhost:61616
Provider Initial Context Factory	The initial context factory used by the client.	org.codehaus.activemq.jndi. ActiveMQInitialContextFactory
Connection Username (optional)	Username for the broker connection.	B.U.R.G.R.
Connection Password (optional)	Password for the broker connection.	CPE-1704-TKS

Table A.4 *Client connection configuration settings*

Property	Description	Sample Value
Test Case ID	The ID for the test case.	Test case
Client ID prefix	The prefix for client IDs within the test case.	Client-
Test Intervals	The number of intervals in the test case.	30
Ramping Intervals	The number of ramping intervals used in the test case (not included within test intervals).	5
Interval Time	The duration of each test interval (in seconds).	60
Rolling Intervals	The number of intervals used for rolling total averages.	5
Producer Class	The class used for the producers within the test case.	DefaultMsgProducer
Consumer Class	The class used for the consumers within the test case.	DefaultMsgConsumer

Table A.5 *General test case*

A.5.2 Test Drivers

Test group drivers are based on the Strategy [113] design pattern to allow pluggable functionality.

Both message consumers and producers use test drivers within the test suite. The interfaces for both of these test drivers, shown in Figure A.4, provide the ability to customise the messaging activity of a test group. Of particular interest are the *send()* and *onMessage(javax.jms.Message aMessage)* methods from the MsgProducer and MsgConsumer interfaces respectively. The tailoring of message send and receive actions allows the test suite to benchmark diverse messaging functionality. The drivers also possess the ability to define the metrics collected during the test case with the use of the *clearCount()* and *getMsgCount()* methods.

Test drivers may be configured for a test case by passing a collection of settings to the drivers upon creation. Configuration options are limited to types compatible with *java.util.Properties.* Eight mandatory configuration options are included, these are detailed in Table A.6.

A.5.2.1 Default Test Driver

The EMiTS default test driver allows the benchmarking of basic messaging functionality within both the point-to-point and publish/subscribe messaging models. In addition to benchmarking ba-

MsgProducer
MsgProducer(Properties props) +start(int currentInterval) : boolean +run() : void +send() : void +stopSender() : void +clearCount() : void +getMsgCount() : void

(a) Producer interface

MsgConsumer
MsgConsumer(Properties props) +start(int currentInterval) : boolean +onMessage(javax.jms.Message aMessage) : void +stop() : void +clearCount() : void +getMsgCount() : void

(b) Consumer interface

Figure A.4 *EMiTS test driver interfaces*

Property	**Description**	**Sample Value**
Group ID	Identification of test group.	GroupAA
Participant Type	Does the test group produce or consume messages?	Producer, Consumer
Message Model	What message model does the test group use?	Point-to-Point, Publish/Subscribe
Number of Clients	The quantity of client within the test group.	5
Number of Connections	The number of connections used by clients within the group.	5
Number of Destinations	The number of destinations used by the group.	5
Destination Name Prefix	The prefix of the destinations for the group.	TestDest-
Start Interval	The test case interval number in which the group will commence messaging activity. (This setting is used to stagger the introduction of test groups.)	0

Table A.6 *Mandatory test driver settings*

sic messaging exchange functionality, the driver can also benchmark transactions, filtering, delivery modes, and acknowledgement modes within both models. A full listing of configuration options for the default driver is provided in Table A.7.

A.5.3 Test Case Sequence

A full test case sequence within EMiTS is illustrated in Figure A.5. This sequence details the interactions within the test suite over the course of a test case, from the initialisation of the suite by the test case administrator to the return and aggregation of results from client machines.

A.6 Dynamic Environment Simulation

The traditional approach to benchmarking MOM solutions involves a test case with a fixed number of message producers, communicating through a MOM to a fixed number of messages consumers, using a fixed number (0-n) of filters.

Property	Description	Sample Value
Is Transacted	Are message interactions contained within a transaction?	False
Messages Per Transaction	The number of messages per transaction.	5
Filtered	Does the test group use filters?	True
Number of Filters	The number of filters used.	4
Filter Name	The prefix of the message attribute used for the filter.	Prop
Delivery Mode	Are messages persistent or non-persistent?	NON_PERSISTENT
Acknowledgement Mode	The acknowledgement mode used for the test group.	AUTO_ACKNOWLEDGE
Message Size	The size of messages used (in KB).	1

Table A.7 *Default test driver settings*

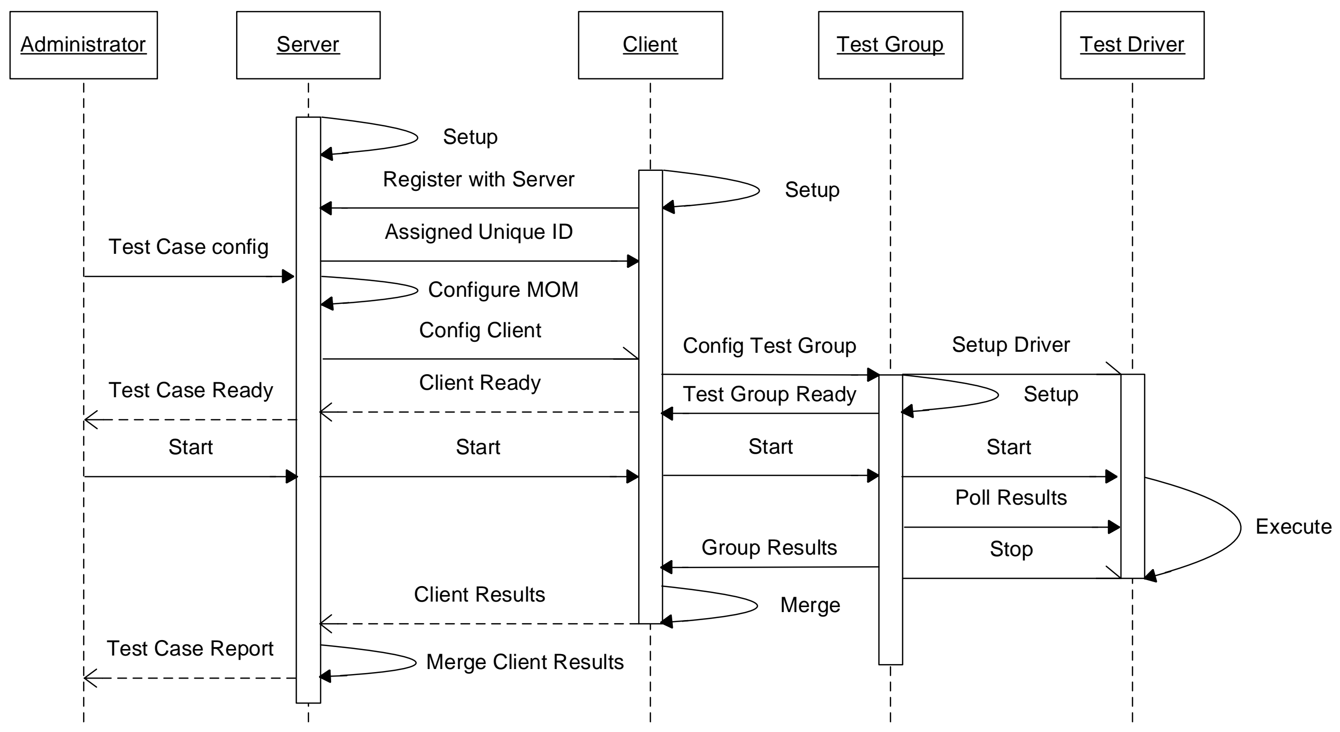


Figure A.5 *EMiTS test case execution sequence*

These conditions are static and remain constant for the duration of the test. This is the common form of test case and has been used by a number of industrial and academic benchmarking efforts [92, 130, 131, 132, 133]. This approach is referred to as static benchmarking.

Static benchmarking is of limited use for evaluating reflection-based self-managed systems. While it is possible to test the performance of a self-managed system under one set of conditions, static benchmarking has no capacity to test the system for its ability to adapt to new and changing conditions; the key objective of a self-managed system. With this deficiency in mind, the use of a new approach to test case design for reflection-based self-managed MOM platforms is proposed.

The key to successfully benchmarking a self-managed system is to gauge its reflective capability. To achieve this the test case must simulate the varying conditions experienced within dynamic environments. An effective method of achieving this is to stage environmental changes over the duration of the test case to replicate a non-static environment. With these environmental changes, it is possible to gauge the ability of the self-managed system to deal with dynamic requirements. The remainder of this section provides a systematic walkthrough of the dynamic MOM test case developed for this research and discusses reporting metrics within dynamic benchmarks.

A.6.1 Dynamic MOM Test Case Walkthrough

Simulating a dynamic MOM environment requires that the dynamic conditions within the environment are first identified. Within the MOM domain, the main dynamic conditions within deployment environments are the number of message producers, message consumers, and associated subscription constraints.

Creating a dynamic test case for a self-managed MOM can be achieved by altering these conditions at runtime and staggering the introduction of new message producers and consumers into the environment. Messaging conditions can also be altering by altering the subscriptions (interests) of message participants over the duration of the test case. In order to demonstrate this action an example test case is presented with the following three events simulated:

- Simulation Event A: 10 Producers and 10 Consumers with 5 filters join at interval 1, participating until interval 100
- Simulation Event B: 10 Consumers with 15 filters join at interval 33, participating until interval 100
- Simulation Event C: 5 Producers and 5 Consumers with 20 filters join at interval 66, participating until interval 100

The timeline for the test cases builds on the default timeline, discussed in Section A.2.5. This dynamic test case timeline is enhanced with the environmental simulation events and is illustrated in Figure A.6.

This test case has been specifically designed to replicate dynamic operational conditions with three specific events taking place within the environment over the duration of the test case. As each of these events take place, it is possible to observe how the MOM reacts to them and gauge its ability to cope with these dynamic conditions. In order to capture the MOMs reaction over the duration of a dynamic test case a different reporting metric is needed.

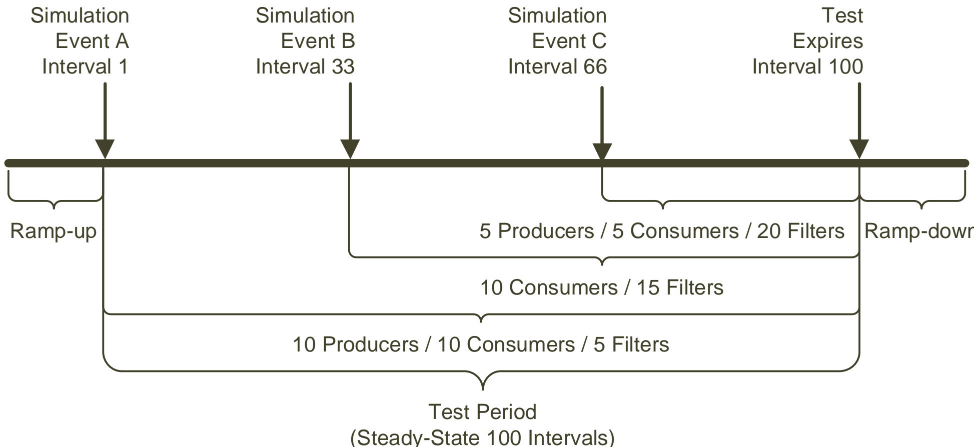


Figure A.6 *Dynamic test case timeline*

A.6.2 Dynamic Report Metric

The common metric used within static benchmarks is a message-received per interval rate. Typically, this is reported as an average for all intervals of the test case i.e. 15,543 msg/sec. However, within a dynamic benchmark, there is an important relationship between an intervals message rate and the activity of the dynamic environment during that interval. This relationship is not represented with a single throughput figure for the test case. It is vital to observe the reaction of a self-managed system to an environmental change: how it behaved before the change, how it reacted to and during the change, and how it performed after the change. To capture these characteristics a temporal performance trend is used to highlight environmental changes and the system's reaction to the changes. A sample trend result for the test case described in this section is shown in Figure A.7.

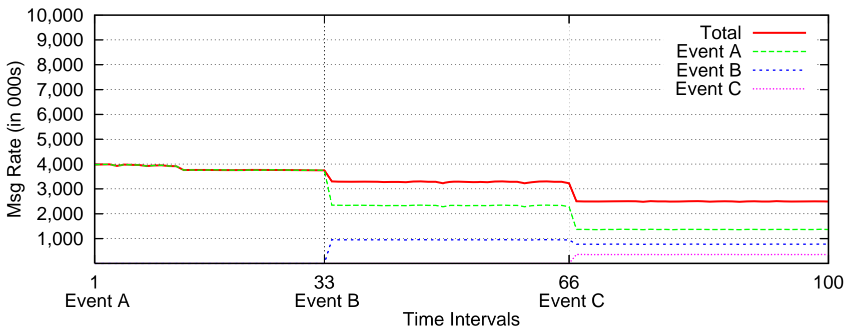


Figure A.7 *Sample trend result for dynamic test case*

A.7 Summary

Benchmarking messaging infrastructures is not a trivial task. This appendix describes the benchmarking process used to evaluate this research. The process was developed by combining the

strengths of a number of academic and industrial benchmarking practices, with additional techniques to benchmark dynamic environments.

The Extensible Mom Test Suite (EMiTS) was developed to evaluate JMS compatible MOM implementations across a collection of networked machines. In addition to a number of advantages over current test suites, EMiTS also possesses capabilities to recreate dynamic test case conditions to simulate dynamic messaging environments, a capability vital to evaluating reflective self-managed MOM implementations.

Appendix B

XML Schemas

B.1 Multimedia-DSL

B.1.1 Capability Request Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="multimedia_dsl_types.xsd"/>
  <xs:element name="Multimedia-Capability-Request">
    <xs:annotation>
      <xs:documentation>Request for multimedia capability.</xs:documentation>
    </xs:annotation>
  </xs:element>
</xs:schema>
```

B.1.2 Capability Request Example

```
<?xml version="1.0" encoding="UTF-8"?>
<Multimedia-Capability-Request/>
```

B.1.3 Capability Reply Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="multimedia_dsl_types.xsd"/>
  <xs:annotation>
    <xs:documentation>Capability types for the Multimedia DSL.</xs:documentation>
  </xs:annotation>
  <!-- Defination of root element -->
  <xs:element name="Multimedia-Capability-Reply">
    <xs:annotation>
      <xs:documentation>Structure used to store multimedia service capabilities. Contains
        a list of service descriptions.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ServiceDescription" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="ServiceDescription">
    <xs:annotation>
      <xs:documentation>Structure used to store service descriptions. Service description
        includes lists of Possible capabilities for available media types, encoding
        formats, bit rates, and delivery mechanisms.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="media" type="MediaTypeList" use="required"/>
      <xs:attribute name="encodingFormat" type="EncodingFormatTypeList" use="required"/>
      <xs:attribute name="bitrate" type="BitrateTypeList" use="required"/>
      <xs:attribute name="deliveryMechanism" type="DeliveryMechanismTypeList"
        use="required"/>
```

```
    </xs:complexType>
  </xs:element>
</xs:schema>
```

B.1.4 Capability Reply Example

```
<?xml version="1.0" encoding="UTF-8"?>
<Multimedia-Capability-Reply>
  <!-- Available multimedia services  -->
  <ServiceDescription media="audio_only" encodingFormat="mp3  ogg wav" bitrate="128kbps
    256kbps 512kbps" deliveryMechanism="download"/>
  <ServiceDescription media="video_only" encodingFormat="ram avi" bitrate="512kbps"
    deliveryMechanism="stream"/>
</Multimedia-Capability-Reply>
```

B.1.5 Service Request Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="multimedia_dsl_types.xsd"/>
  <xs:annotation>
    <xs:documentation>Request structures for specifying multimedia
    services.</xs:documentation>
  </xs:annotation>
  <!-- Defination of root element -->
  <xs:element name="Multimedia-Service-Request">
    <xs:annotation>
      <xs:documentation>Structure used to request specific multimedia services. Contains a
        list of service requests. Each request must contain an
      unique ID.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ServiceRequest" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <!-- RequestID must be unique -->
    <xs:key name="requestIDKey">
      <xs:selector xpath="ServiceRequest"/>
      <xs:field xpath="@requestID"/>
    </xs:key>
  </xs:element>
  <xs:element name="ServiceRequest">
    <xs:annotation>
      <xs:documentation>Structure used to specify multimedia service request. Contains a
        requestID and description of the multimedia service requested including its media
        type, encoding format, bit rate, and delivery mechanism.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="requestID" type="xs:string" use="required"/>
      <xs:attribute name="media" type="MediaType" use="required"/>
      <xs:attribute name="encodingFormat" type="EncodingFormatType" use="required"/>
      <xs:attribute name="bitrate" type="BitrateType" use="required"/>
      <xs:attribute name="deliveryMechanism" type="DeliveryMechanismType" use="required"/>
```

```
    </xs:complexType>
  </xs:element>
</xs:schema>
```

B.1.6 Service Request Example

```
<?xml version="1.0" encoding="UTF-8"?>
<Multimedia-Service-Request>
  <ServiceRequest requestID="1" media="audio_only" encodingFormat=" wma" bitrate="128kbps"
    deliveryMechanism="download"/>
</Multimedia-Service-Request>
```

B.1.7 Service Reply Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="multimedia_dsl_types.xsd"/>
  <xs:annotation>
    <xs:documentation>Defines the reply used for multimedia service
    requests.</xs:documentation>
  </xs:annotation>
  <!-- Defination of root element -->
  <xs:element name="Multimedia-Service-Reply">
    <xs:annotation>
      <xs:documentation>Structure used to store the reply to multimedia service requests.
        Contains a list of replies with a unique request ID.</xs:documentation>
    </xs:annotation>
    <!-- Defination of complex types -->
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ServiceReply" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <!-- RequestID must be unique -->
    <xs:key name="requestIDKey">
      <xs:selector xpath="ServiceReply"/>
      <xs:field xpath="@requestID"/>
    </xs:key>
  </xs:element>
  <xs:element name="ServiceReply">
    <xs:annotation>
      <xs:documentation>Reply to a service request. Contains the request ID, response, and
        URL of service (if applicable).</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="requestID" type="xs:string" use="required"/>
      <xs:attribute name="response" type="ServiceResposneType" use="required"/>
      <xs:attribute name="URL" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

B.1.8 Service Reply Example

```
<?xml version="1.0" encoding="UTF-8"?>
<Multimedia-Service-Reply>
  <ServiceReply requestID="1" response="accept" URL="http://url"/>
</Multimedia-Service-Reply>
```

B.1.9 Common Type Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:documentation>Common types used within the Multimedia DSL.</xs:documentation>
  </xs:annotation>
  <xs:simpleType name="ServiceResposneType">
    <xs:annotation>
      <xs:documentation>Possible responses to a service request.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="accept"/>
      <xs:enumeration value="refuse"/>
      <xs:enumeration value="bestEffort"/>
    </xs:restriction>
  </xs:simpleType>
  <!-- Types used within service descriptions -->
  <xs:simpleType name="MediaTypeList">
    <xs:annotation>
      <xs:documentation>A list of Possible media types.</xs:documentation>
    </xs:annotation>
    <xs:list itemType="MediaType"/>
  </xs:simpleType>
  <xs:simpleType name="EncodingFormatTypeList">
    <xs:annotation>
      <xs:documentation>A list of Possible encoding formats.</xs:documentation>
    </xs:annotation>
    <xs:list itemType="EncodingFormatType"/>
  </xs:simpleType>
  <xs:simpleType name="BitrateTypeList">
    <xs:annotation>
      <xs:documentation>A list of Possible bit rates.</xs:documentation>
    </xs:annotation>
    <xs:list itemType="BitrateType"/>
  </xs:simpleType>
  <xs:simpleType name="DeliveryMechanismTypeList">
    <xs:annotation>
      <xs:documentation>A list of Possible delivery mechanisms.</xs:documentation>
    </xs:annotation>
    <xs:list itemType="DeliveryMechanismType"/>
  </xs:simpleType>
  <xs:simpleType name="MediaType">
    <xs:annotation>
      <xs:documentation>Possible media formats.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="audio_only"/>
      <xs:enumeration value="video_only"/>
```

```
      <xs:enumeration value="audio_video"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="EncodingFormatType">
    <xs:annotation>
      <xs:documentation>Possible encoding formats.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="aac"/>
      <xs:enumeration value="avi"/>
      <xs:enumeration value="mp3"/>
      <xs:enumeration value="mpg"/>
      <xs:enumeration value="ram"/>
      <xs:enumeration value="ogg"/>
      <xs:enumeration value="wav"/>
      <xs:enumeration value=" wma"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="BitrateType">
    <xs:annotation>
      <xs:documentation>Possible bit rates.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="28kbps"/>
      <xs:enumeration value="56kbps"/>
      <xs:enumeration value="128kbps"/>
      <xs:enumeration value="256kbps"/>
      <xs:enumeration value="512kbps"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="DeliveryMechanismType">
    <xs:annotation>
      <xs:documentation>Possible delivery mechanisms.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="stream"/>
      <xs:enumeration value="download"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

B.2 MOM-DSL

B.2.1 Destination Type Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="mom_dsl_common_types.xsd"/>
  <xs:annotation>
    <xs:documentation>Destination meta-state structures.</xs:documentation>
  </xs:annotation>
  <!-- Destination root element -->
  <xs:complexType name="DestinationStateType">
    <xs:annotation>
      <xs:documentation> The Destination state type is used to store destination state. The
        top level of the structure contains two groupings to store destinations:
```

```
      - Single_Destinations (stores standalone destinations)
      - DestinationHierarchy (stores destination hierarchies)
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="Single_Destinations" type="DestinationListType" minOccurs="1"
      maxOccurs="1">
      <xs:annotation>
        <xs:documentation>A collection of standalone destinations. Each destination
          within the collection has a unique key.</xs:documentation>
      </xs:annotation>
      <xs:key name="destKey">
        <xs:selector xpath="Destination"/>
        <xs:field xpath="@id"/>
      </xs:key>
    </xs:element>
    <xs:element name="Hierarchys">
      <xs:annotation>
        <xs:documentation>A collection of destination hierarchies.</xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:sequence>
          <xs:element name="DestinationHierarchy" type="DestinationHierarchyType"
            minOccurs="0" maxOccurs="unbounded">
            <xs:annotation>
              <xs:documentation>A destination hierarchy. Each node within the hierarchy
                must contain a unique key. This key is then used in linking references
                within the hierarchy to maintain a consistent
              structure.</xs:documentation>
            </xs:annotation>
            <xs:key name="nodeKey">
              <xs:selector xpath="HierarchyNode"/>
              <xs:field xpath="@id"/>
            </xs:key>
            <xs:keyref name="rootRef" refer="nodeKey">
              <xs:selector xpath="DestinationHierarchy"/>
              <xs:field xpath="@root"/>
            </xs:keyref>
            <xs:keyref name="parentRef" refer="nodeKey">
              <xs:selector xpath="HierarchyNode"/>
              <xs:field xpath="@parentNode"/>
            </xs:keyref>
            <xs:keyref name="childRef" refer="nodeKey">
              <xs:selector xpath="HierarchyNode/ChildNode"/>
              <xs:field xpath="@childID"/>
            </xs:keyref>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<!-- Basic destination elements -->
<xs:complexType name="DestinationType">
  <xs:annotation>
    <xs:documentation>Structure used to track basic destination state. Contains the ID,
      name, and type of the destination. Optional routing conditions may also be tracked
      for the destination.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
```

```
    <xs:element name="condition" type="FilterType" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string" use="required"/>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="type" type="DestType"/>
</xs:complexType>
<xs:simpleType name="DestType">
  <xs:annotation>
    <xs:documentation>Specifies the destination type.</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="queue"/>
    <xs:enumeration value="topic"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="DestinationListType">
  <xs:annotation>
    <xs:documentation>A collection of destinations.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="Destination" type="DestinationType" minOccurs="0"
      maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<!-- Destination hierarchy element -->
<xs:complexType name="DestinationHierarchyType">
  <xs:annotation>
    <xs:documentation>Structure used to track hierarchy state. The hierarchy contains a
      ID and a reference to the ID of its root node.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="HierarchyNode" type="HierarchyNodeType" minOccurs="0"
      maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="hierarchyID" type="xs:string" use="required"/>
  <xs:attribute name="root" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="HierarchyNodeType">
  <xs:annotation>
    <xs:documentation>Structure used to track an individual node within a hierarchy. The
      node structure extends the basic destination structure with additional information
      to reference links to parent and child nodes.</xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="DestinationType">
      <xs:sequence>
        <xs:element name="ChildNode" type="ChildNodeType" minOccurs="0"
          maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="parentNode" type="xs:string"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="ChildNodeType">
  <xs:attribute name="childID" type="xs:string" use="required">
    <xs:annotation>
      <xs:documentation>Specify the unique ID for the child node.</xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:complexType>
```

```
  <!-- Destination Analysis -->
  <xs:complexType name="DestinationSearchRequestType">
    <xs:annotation>
      <xs:documentation>Structure to store a destination analysis request. Specify the
        destination name to search for, wildcards allowed.</xs:documentation>
    </xs:annotation>
    <xs:attribute name="destinationName" type="xs:string" use="required"/>
  </xs:complexType>
</xs:schema>
```

B.2.2 Subscription Type Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="mom_dsl_common_types.xsd"/>
  <xs:annotation>
    <xs:documentation>Subscription meta-state structures.</xs:documentation>
  </xs:annotation>
  <!-- Basic subscription type -->
  <xs:complexType name="SubscriptionType">
    <xs:annotation>
      <xs:documentation>Structure used to track subscription state. Contains a ID, a
        collection of filters and the target destination.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="filter" type="FilterType" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID"/>
    <xs:attribute name="destinationName" type="xs:string" use="required"/>
  </xs:complexType>
  <!-- Subscription analysis types -->
  <xs:complexType name="SubscriberCountRequestType">
    <xs:annotation>
      <xs:documentation>Structure used to express a subscription count analysis request.
        Contains the target destination to count subscribers for, wildcards
      allowed.</xs:documentation>
    </xs:annotation>
    <xs:attribute name="destinationName" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="SubscriberCountReplyType">
    <xs:annotation>
      <xs:documentation>Result of the subscriber count analysis.</xs:documentation>
    </xs:annotation>
    <xs:attribute name="subscriberCount" type="xs:int" use="required"/>
  </xs:complexType>
  <xs:complexType name="DestinationFilterCountRequestType">
    <xs:annotation>
      <xs:documentation>Structure to specify the destination count filter
      request.</xs:documentation>
    </xs:annotation>
    <xs:attribute name="destinationName" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="DestinationFilterCountReplyType">
    <xs:annotation>
      <xs:documentation>Contains the results from the filter count
      analysis.</xs:documentation>
    </xs:annotation>
```

```
    <xs:sequence>
      <xs:element name="filter" type="FilterCountType" minOccurs="0" maxOccurs="unbounded"
      />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="FilterCountType">
    <xs:annotation>
      <xs:documentation>Structure used to count filters. Extends the basic filter type
        with a count attribute.</xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="FilterType">
        <xs:attribute name="count" type="xs:int" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

B.2.3 Interception Type Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="mom_dsl_common_types.xsd"/>
  <xs:annotation>
    <xs:documentation>Interception meta-state structures.</xs:documentation>
  </xs:annotation>
  <!-- Basic Interceptior element -->
  <xs:complexType name="InterceptorType">
    <xs:annotation>
      <xs:documentation>Structure used to track interception state. Contains the name,
        interception point, classname, and scope of an interceptor. An optional collection
        of configuration options may also be tracked.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Option" type="OptionType" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="interceptionPont" type="InterceptionPointTypeList" use="required"/>
    <xs:attribute name="className" type="xs:string" use="required"/>
    <xs:attribute name="scope" type="xs:string" use="required">
      <xs:annotation>
        <xs:documentation>Specify the scope of the interceptor. If left blank a global
          scope is assumed. Otherwise specify local destination scope.</xs:documentation>
      </xs:annotation>
    </xs:attribute>
  </xs:complexType>
  <xs:complexType name="InterceptionType">
    <xs:annotation>
      <xs:documentation>Structure used to track a collection of
      interceptors.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Interceptor" type="InterceptorType" minOccurs="0"
        maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="InterceptionPointTypeList">
```

```
    <xs:annotation>
      <xs:documentation>A list of interception points.</xs:documentation>
    </xs:annotation>
    <xs:list itemType="InterceptionPointType"/>
  </xs:simpleType>
  <xs:simpleType name="InterceptionPointType">
    <xs:annotation>
      <xs:documentation>Possible locations for interception.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="CreateDestination"/>
      <xs:enumeration value="UpdateDestination"/>
      <xs:enumeration value="DeleteDestination"/>
      <xs:enumeration value="CreateSubscription"/>
      <xs:enumeration value="DeleteSubscription"/>
      <xs:enumeration value="SendMessage"/>
      <xs:enumeration value="ReceiveMessage"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

B.2.4 Reflective Type Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="mom_dsl_common_types.xsd"/>
  <xs:annotation>
    <xs:documentation>Reflective meta-state structures.</xs:documentation>
  </xs:annotation>
  <!-- Basic reflective policy element -->
  <xs:complexType name="ReflectivePolicyType">
    <xs:annotation>
      <xs:documentation>Structure used to track a reflective policy. Contains the name,
        reflective location, classname, scope, and synchronisation of a policy. An
        optional collection of configuration options may also be
      tracked.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Option" type="OptionType" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="location" type="ReflectiveLocationTypeList" use="required"/>
    <xs:attribute name="className" type="xs:string" use="required"/>
    <xs:attribute name="scope" type="xs:string" use="required">
      <xs:annotation>
        <xs:documentation>Specify the scope of the interceptor. If left blank a global
          scope is assumed. Otherwise specify local destination scope.</xs:documentation>
      </xs:annotation>
    </xs:attribute>
    <xs:attribute name="synchronisation" type="synchronisationType" use="required"/>
  </xs:complexType>
  <xs:complexType name="ReflectivePoliciesType">
    <xs:annotation>
      <xs:documentation>Structure used to track a collection of reflective
      policies.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
```

```
    <xs:element name="Policy" type="ReflectivePolicyType" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="ReflectiveLocationTypeList">
  <xs:annotation>
    <xs:documentation>A list of possible reflective locations.</xs:documentation>
  </xs:annotation>
  <xs:list itemType="ReflectiveLocationType"/>
</xs:simpleType>
<xs:simpleType name="ReflectiveLocationType">
  <xs:annotation>
    <xs:documentation>Possible locations for reflective policy
    attachment.</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="CreateDestination"/>
    <xs:enumeration value="UpdateDestination"/>
    <xs:enumeration value="DeleteDestination"/>
    <xs:enumeration value="CreateSubscription"/>
    <xs:enumeration value="DeleteSubscription"/>
    <xs:enumeration value="SendMessage"/>
    <xs:enumeration value="ReceiveMessage"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="synchronisationType">
  <xs:annotation>
    <xs:documentation>Synchronisation options available to policies.</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="sync"/>
    <xs:enumeration value="asyn"/>
  </xs:restriction>
</xs:simpleType>
</xs:schema>
```

B.2.5 Event Type Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:documentation>Event model structures.</xs:documentation>
  </xs:annotation>
  <!-- Basic event element -->
  <xs:complexType name="EventLocationType">
    <xs:annotation>
      <xs:documentation>Structure to store event location information. </xs:documentation>
    </xs:annotation>
    <xs:attribute name="event" type="EventSelectionType" use="required"/>
    <xs:attribute name="location" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:simpleType name="EventSelectionType">
    <xs:annotation>
      <xs:documentation>Available event types.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="CreateDestination"/>
      <xs:enumeration value="UpdateDestination"/>
```

```
      <xs:enumeration value="DeleteDestination"/>
      <xs:enumeration value="CreateSubscription"/>
      <xs:enumeration value="DeleteSubscription"/>
      <xs:enumeration value="SendMessage"/>
      <xs:enumeration value="ReceiveMessage"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

B.2.6 Capability Type Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:include schemaLocation="mom_dsl_event_types.xsd"/>
  <xs:annotation>
    <xs:documentation>Capability types for the MOM-DSL.</xs:documentation>
  </xs:annotation>
  <!--Basic capability structure -->
  <xs:complexType name="CapabilityType">
    <xs:annotation>
      <xs:documentation>Structure to store capability state. Includes access information
        for Destination, Subscription, Interception and Reflective meta-spaces. Also
        includes information on event model and meta-analyses.</xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Destination" type="MetaAccessCapabilityType" minOccurs="0"
        maxOccurs="1"/>
      <xs:element name="Subscription" type="MetaAccessCapabilityType" minOccurs="0"
        maxOccurs="1"/>
      <xs:element name="Interception" type="MetaAccessCapabilityType" minOccurs="0"
        maxOccurs="1"/>
      <xs:element name="Reflective" type="MetaAccessCapabilityType" minOccurs="0"
        maxOccurs="1"/>
      <xs:element name="AvailableEvents" minOccurs="0" maxOccurs="1">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="EventLocation" type="EventLocationType" minOccurs="0"
              maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="AvailableAnalysis" minOccurs="0" maxOccurs="1">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Analysis" type="AnalysisType" minOccurs="0"
              maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="MetaAccessCapabilityType">
    <xs:annotation>
      <xs:documentation>Generic structure containing access information for a meta-space.
        Specifies access to state, creation, update, and deletion capabilities.
      </xs:documentation>
    </xs:annotation>
```

```
    <xs:attribute name="state" type="xs:boolean" use="required"/>
    <xs:attribute name="create" type="xs:boolean" use="required"/>
    <xs:attribute name="update" type="xs:boolean" use="required"/>
    <xs:attribute name="delete" type="xs:boolean" use="required"/>
  </xs:complexType>
  <!-- Analysis types -->
  <xs:complexType name="AnalysisType">
    <xs:annotation>
      <xs:documentation>Structure to specific access to meta-analysis.</xs:documentation>
    </xs:annotation>
    <xs:attribute name="analysis" type="AnalysisOperationType" use="required"/>
    <xs:attribute name="access" type="xs:boolean"/>
  </xs:complexType>
  <xs:simpleType name="AnalysisOperationType">
    <xs:annotation>
      <xs:documentation/>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="DestinationSearch"/>
      <xs:enumeration value="SubscriptionCount"/>
      <xs:enumeration value="DestinationFilterCount"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

B.2.7 Common Type Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:documentation>Common types for the MOM-DSL.</xs:documentation>
  </xs:annotation>
  <!-- Common types -->
  <xs:complexType name="OptionType">
    <xs:annotation>
      <xs:documentation>Option element used for reflective policies and interceptors.
        Contain a name to identify the option and a value.</xs:documentation>
    </xs:annotation>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="value" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="FilterType">
    <xs:annotation>
      <xs:documentation>Structure used to track filter state. Contains a name to identify
        the attribute, the operator use in comparison and the valued
      compared.</xs:documentation>
    </xs:annotation>
    <xs:attribute name="attribute" type="xs:string" use="required"/>
    <xs:attribute name="operator" type="operatorType" use="required"/>
    <xs:attribute name="value" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:simpleType name="operatorType">
    <xs:annotation>
      <xs:documentation>Operators available within comparisons.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="NOT IN"/>
```

```
      <xs:enumeration value="="/>
      <xs:enumeration value="&gt;"/>
      <xs:enumeration value="&gt;="/>
      <xs:enumeration value="&lt;"/>
      <xs:enumeration value="&lt;="/>
      <xs:enumeration value="&lt;&gt;"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="AvailableStateType">
    <xs:annotation>
      <xs:documentation>Available meta-states.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Destination"/>
      <xs:enumeration value="Subscription"/>
      <xs:enumeration value="Interception"/>
      <xs:enumeration value="Reflective"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

B.2.8 Generic MOM-DSL Request/Reply Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!-- Import type schemas  -->
  <xs:include schemaLocation="mom_dsl_common_types.xsd"/>
  <xs:include schemaLocation="mom_dsl_capability_types.xsd"/>
  <xs:include schemaLocation="mom_dsl_destination_types.xsd"/>
  <xs:include schemaLocation="mom_dsl_subscription_types.xsd"/>
  <xs:include schemaLocation="mom_dsl_interception_types.xsd"/>
  <xs:include schemaLocation="mom_dsl_reflective_types.xsd"/>
  <xs:include schemaLocation="mom_dsl_event_types.xsd"/>
  <xs:annotation>
    <xs:documentation>Generic request/reply structure for MOM-DSL.</xs:documentation>
  </xs:annotation>
  <!-- Root element -->
  <xs:element name="MOM-DSL">
    <xs:annotation>
      <xs:documentation>Root element that contains a collection of Request and/or reply
        actions. Each request/reply must contain a unique request ID.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice>
        <xs:element name="Request" type="RequestType" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element name="Reply" type="ReplyType" minOccurs="1" maxOccurs="unbounded"/>
      </xs:choice>
    </xs:complexType>
    <xs:key name="requestIDKey">
      <xs:selector xpath="Request"/>
      <xs:field xpath="@requestID"/>
    </xs:key>
    <xs:key name="replyIDKey">
      <xs:selector xpath="Reply"/>
      <xs:field xpath="@replyID"/>
    </xs:key>
  </xs:element>
```

```
<xs:complexType name="RequestType">
  <xs:annotation>
    <xs:documentation>Structure used to specify a request. May be used to request
      capabilities, state, change to state (create, update, delete), or analysis. Each
      request must contain a unique request ID. </xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:element name="Capability-Request">
      <xs:annotation>
        <xs:documentation>Request capabilities. Optional username and
        password.</xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:attribute name="username" type="xs:string"/>
        <xs:attribute name="password" type="xs:string"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="State-Request">
      <xs:annotation>
        <xs:documentation>Request State. Specify state request (Destination,
          Subscription, Interception, Reflective).</xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:attribute name="state" type="AvailableStateType" use="required"/>
      </xs:complexType>
    </xs:element>
    <!-- State editing structures  -->
    <xs:element name="Create">
      <xs:annotation>
        <xs:documentation>Request the creation of state. Contains the new state
          structure (Destination, Subscription, Interception,
        Reflective).</xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:choice>
          <xs:element name="Destination" type="DestinationType" minOccurs="1"
            maxOccurs="1"/>
          <xs:element name="HierarchyNode" type="HierarchyNodeType" minOccurs="1"
            maxOccurs="1"/>
          <xs:element name="Subscription" type="SubscriptionType" minOccurs="1"
            maxOccurs="1"/>
          <xs:element name="Interceptor" type="InterceptorType" minOccurs="1"
            maxOccurs="1"/>
          <xs:element name="ReflectivePolicy" type="ReflectivePolicyType" minOccurs="1"
            maxOccurs="1"/>
        </xs:choice>
      </xs:complexType>
    </xs:element>
    <xs:element name="Update">
      <xs:annotation>
        <xs:documentation>Request state to be updated. Contains the old and new state
          structures (Destination, Subscription, Interception,
        Reflective).</xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:choice>
          <xs:element name="UpdateDestination">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="CurrentDestination" type="DestinationType"
```

```
          minOccurs="1" maxOccurs="1"/>
        <xs:element name="NewDestination" type="DestinationType" minOccurs="1"
          maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="UpdateHierarchy">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="CurrentHierarchyNode" type="HierarchyNodeType"
          minOccurs="1" maxOccurs="1"/>
        <xs:element name="NewHierarchyNode" type="HierarchyNodeType"
          minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="UpdateSubscription">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="CurrentSubscription" type="SubscriptionType"
          minOccurs="1" maxOccurs="1"/>
        <xs:element name="NewSubscription" type="SubscriptionType" minOccurs="1"
          maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="UpdateInterceptor">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="CurrentInterceptor" type="InterceptorType"
          minOccurs="1" maxOccurs="1"/>
        <xs:element name="NewInterceptor" type="InterceptorType" minOccurs="1"
          maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="UpdateReflectivePolicy">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="CurrentReflectivePolicy" type="ReflectivePolicyType"
          minOccurs="1" maxOccurs="1"/>
        <xs:element name="NewReflectivePolicy" type="ReflectivePolicyType"
          minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:choice>
</xs:complexType>
</xs:element>
<xs:element name="Delete">
  <xs:annotation>
    <xs:documentation>Request the deletion of state. Contains the state structure
      (Destination, Subscription, Interception, Reflective) to be
    removed.</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:choice>
      <xs:element name="Destination" type="DestinationType" minOccurs="1"
        maxOccurs="1"/>
      <xs:element name="HierarchyNode" type="HierarchyNodeType" minOccurs="1"
```

```
          maxOccurs="1"/>
        <xs:element name="Subscription" type="SubscriptionType" minOccurs="1"
          maxOccurs="1"/>
        <xs:element name="Interceptor" type="InterceptorType" minOccurs="1"
          maxOccurs="1"/>
        <xs:element name="ReflectivePolicy" type="ReflectivePolicyType" minOccurs="1"
          maxOccurs="1"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <!-- Analysis structures  -->
  <xs:element name="Analysis">
    <xs:annotation>
      <xs:documentation>Request Analysis. Specify the analysis type with associated
        parameters.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice>
        <xs:element name="DestinationSearch" type="DestinationSearchRequestType"/>
        <xs:element name="SubscriberCount" type="SubscriberCountRequestType"/>
        <xs:element name="DestinationFilterCount"
          type="DestinationFilterCountRequestType"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:choice>
<xs:attribute name="requestID" type="xs:string" use="required">
  <xs:annotation>
    <xs:documentation>Specify a required unique ID for this request.</xs:documentation>
  </xs:annotation>
</xs:attribute>
</xs:complexType>
<xs:complexType name="ReplyType">
  <xs:annotation>
    <xs:documentation>Structure used to specify a reply. May be used to reply to
      requests for capabilities, state, change to state (create, update, delete), or
      analysis. Each reply must contain a unique ID and a response type.
    </xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:element name="Capability" type="CapabilityType" minOccurs="0" maxOccurs="1"/>
    <xs:element name="DestinationState" type="DestinationStateType" minOccurs="0"
      maxOccurs="1"/>
    <xs:element name="ReflectivePolicies" type="ReflectivePoliciesType" minOccurs="0"
      maxOccurs="1"/>
    <xs:element name="Interception" type="InterceptionType" minOccurs="0" maxOccurs="1"/>
    <xs:element name="AnalysisResult" type="AnalysisResultType" minOccurs="0"
      maxOccurs="1"/>
  </xs:choice>
  <xs:attribute name="replyID" type="xs:string" use="required"/>
  <xs:attribute name="response" type="ResposneType" use="required"/>
</xs:complexType>
<xs:simpleType name="ResposneType">
  <xs:annotation>
    <xs:documentation>Possible responses to a service request.</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="accept"/>
    <xs:enumeration value="refuse"/>
  </xs:restriction>
```

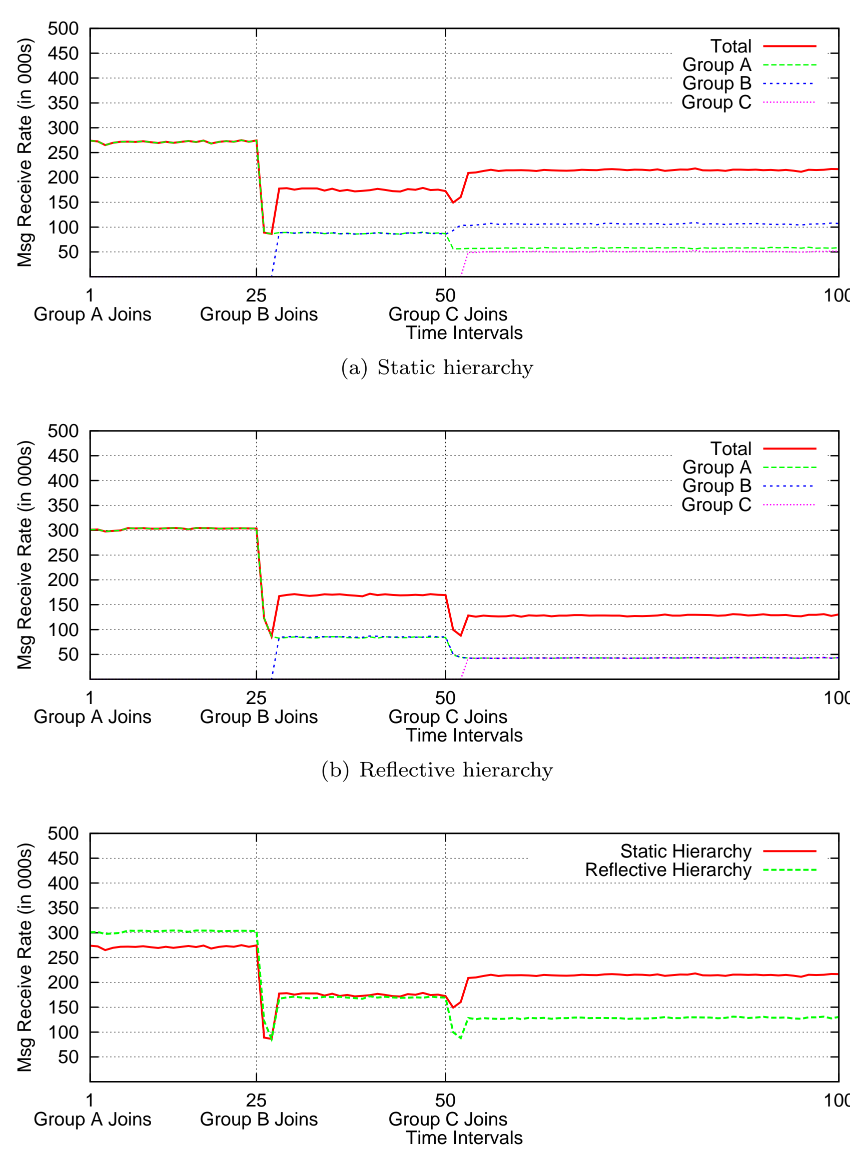
```
  </xs:simpleType>
  <xs:complexType name="AnalysisResultType">
    <xs:annotation>
      <xs:documentation>Possible responses to a analysis request.</xs:documentation>
    </xs:annotation>
    <xs:choice>
      <xs:element name="DestinationSearchResults" type="DestinationListType"/>
      <xs:element name="SubscriberCountResults" type="SubscriberCountReplyType"/>
      <xs:element name="DestinationFilterCountResults"
        type="DestinationFilterCountReplyType"/>
    </xs:choice>
  </xs:complexType>
</xs:schema>
```

Appendix C

Additional Results from Chapter 8

C.1 One-to-One Results

C.1.1 3 Producers / 3 Consumers



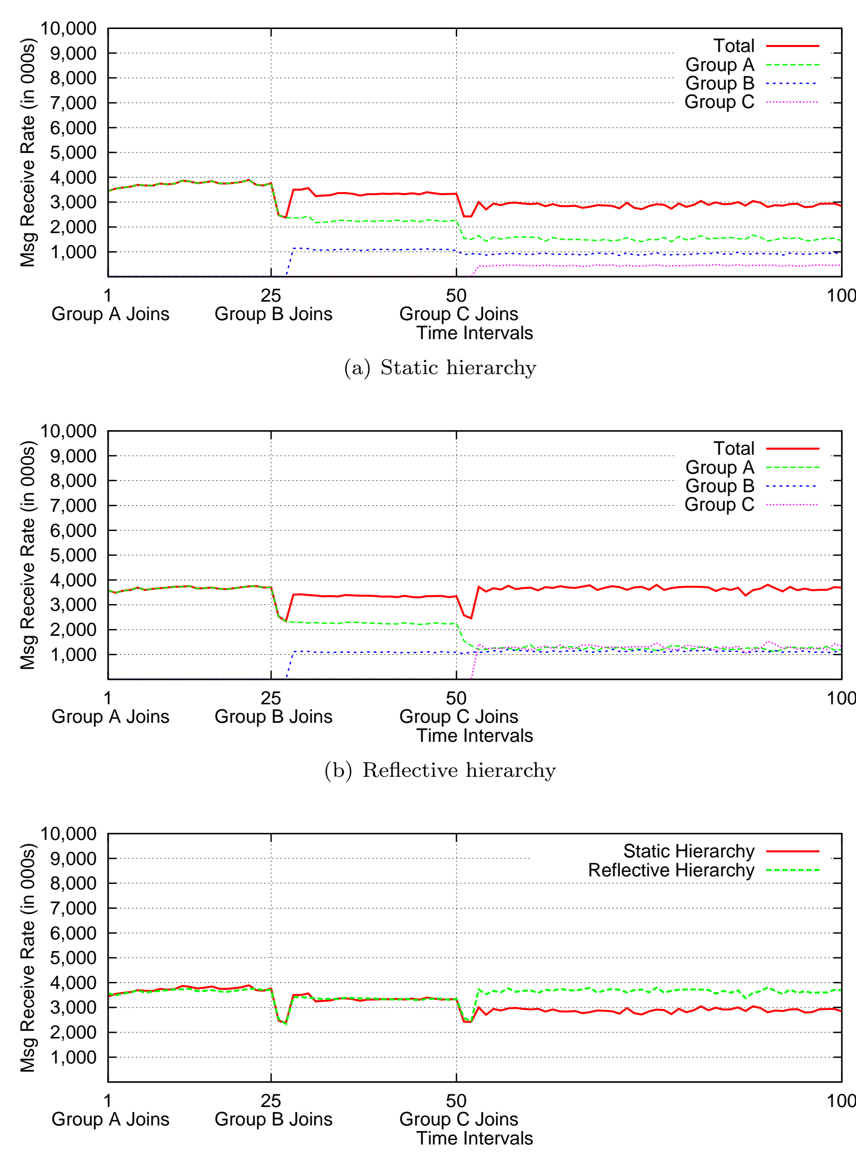
(a) Static hierarchy

(b) Reflective hierarchy

(c) Static hierarchy vs. reflective hierarchy

Figure C.1 *One-to-one 3 / 3*

C.1.2 30 Producers / 30 Consumers



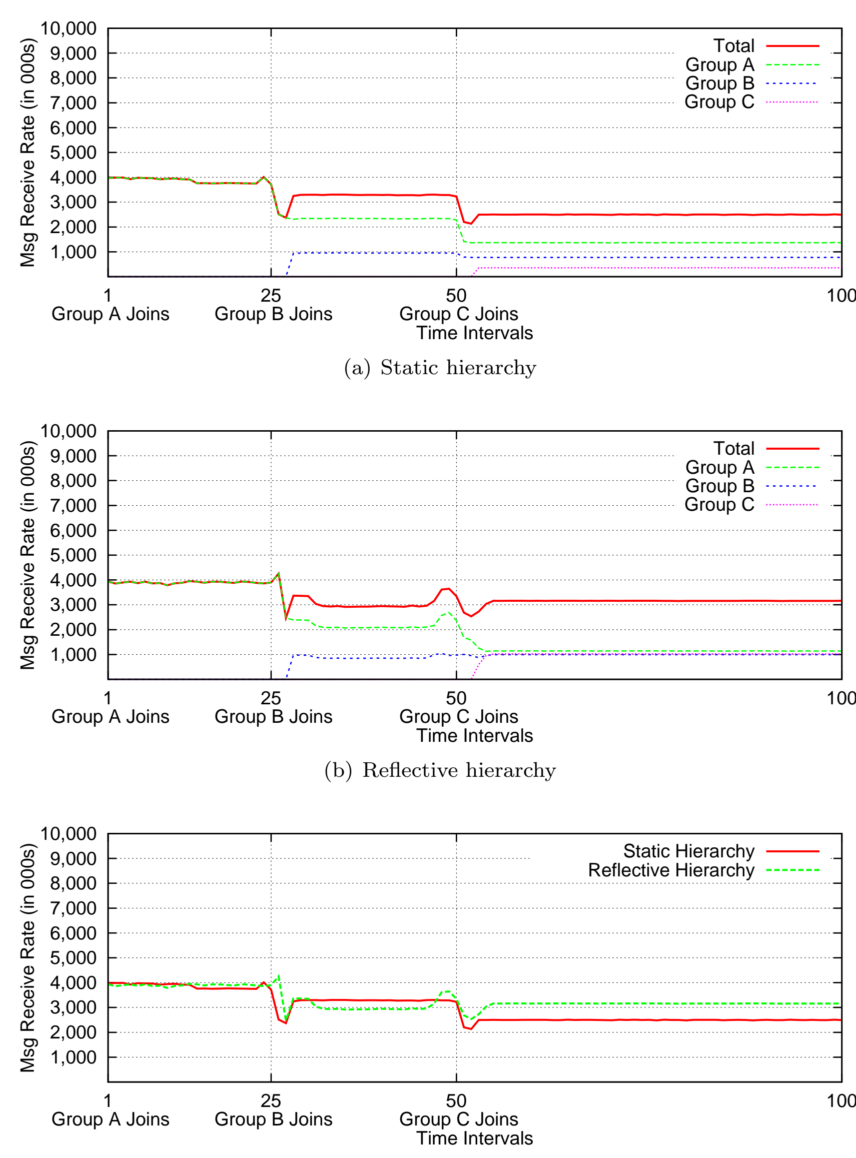
(a) Static hierarchy

(b) Reflective hierarchy

(c) Static hierarchy vs. reflective hierarchy

Figure C.2 *One-to-one 30 / 30*

C.1.3 90 Producers / 90 Consumers



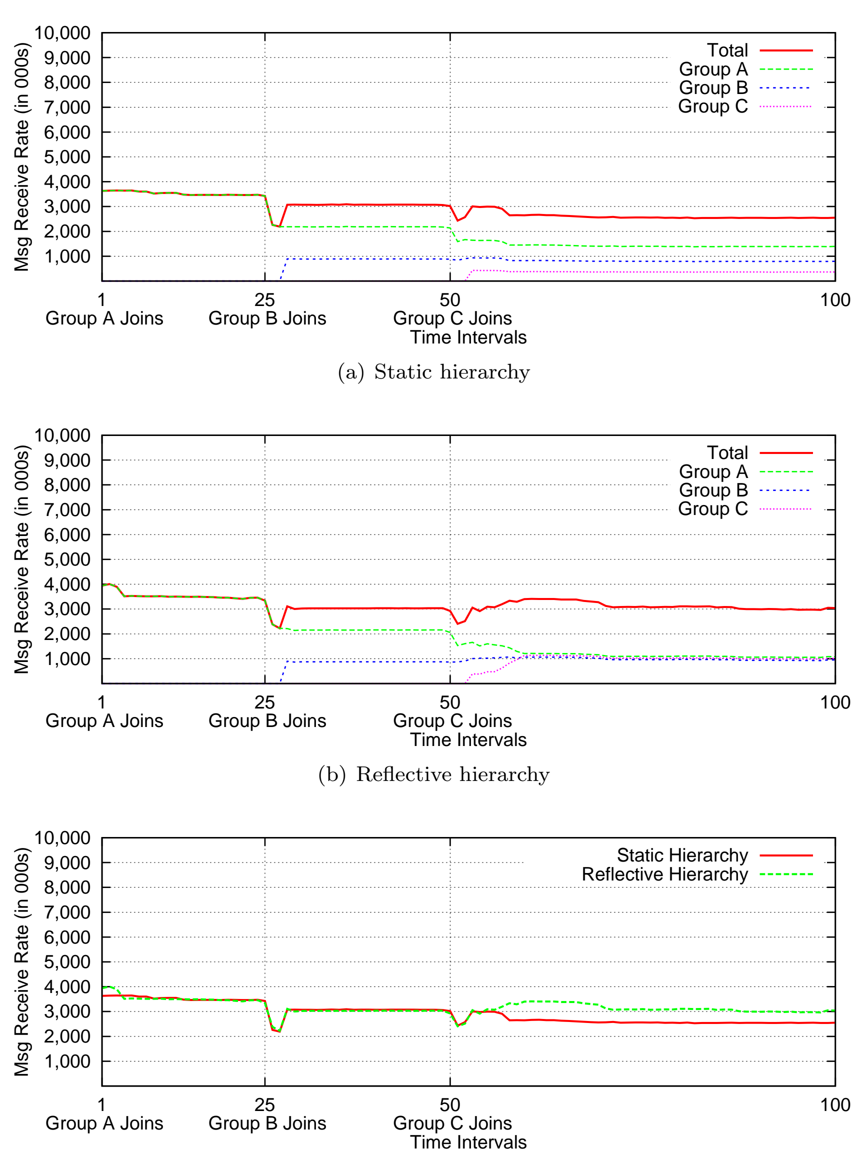
(a) Static hierarchy

(b) Reflective hierarchy

(c) Static hierarchy vs. reflective hierarchy

Figure C.3 *One-to-one 90 / 90*

C.1.4 150 Producers / 150 Consumers



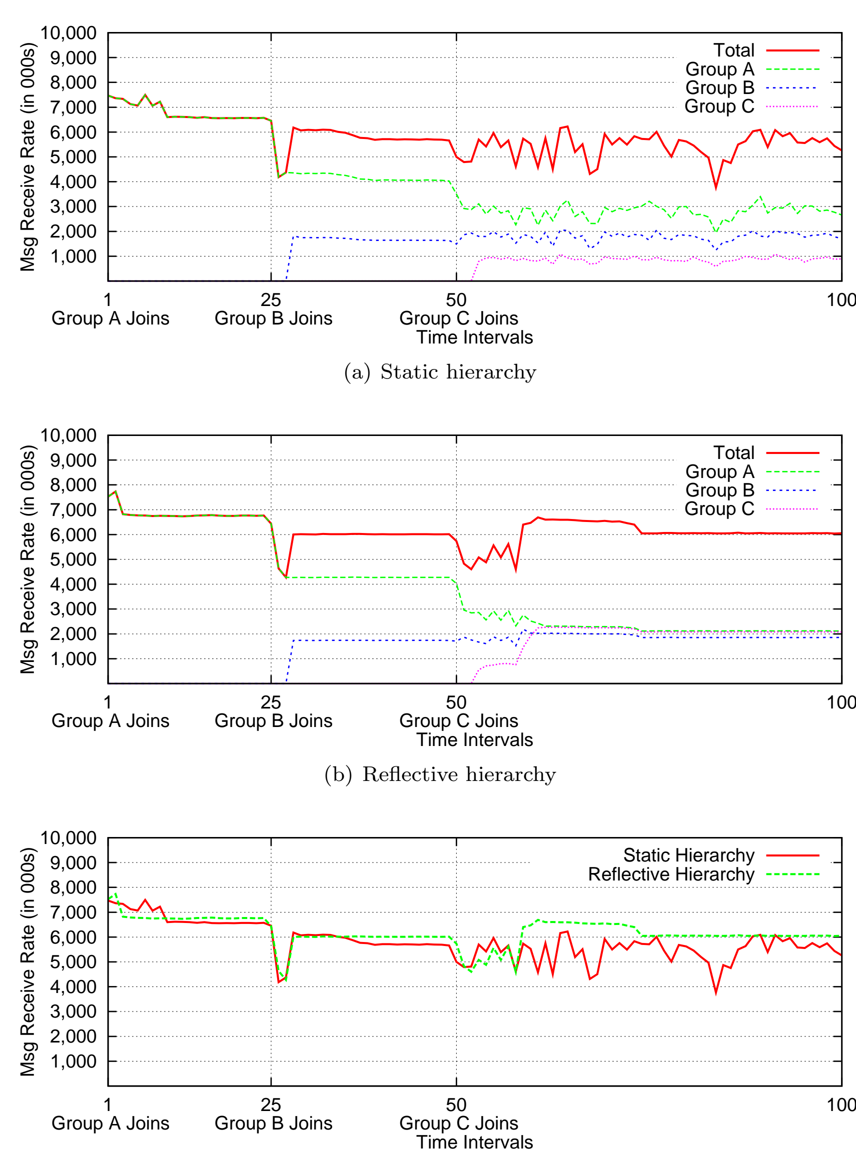
(a) Static hierarchy

(b) Reflective hierarchy

(c) Static hierarchy vs. reflective hierarchy

Figure C.4 *One-to-one 150 / 150*

C.1.5 300 Producers / 300 Consumers

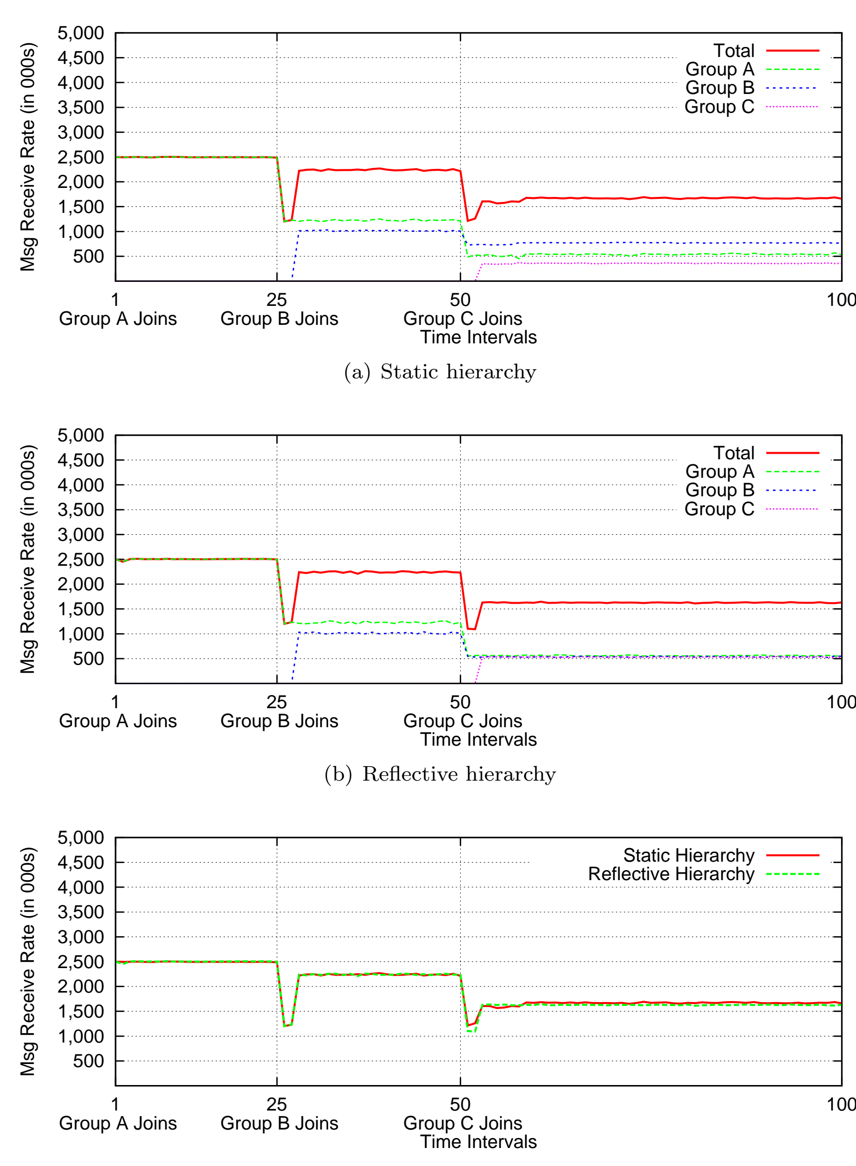


(c) Static hierarchy vs. reflective hierarchy

Figure C.5 *One-to-one 300 / 300*

C.2 Few-to-Many Results

C.2.1 3 Producers / 15 Consumers



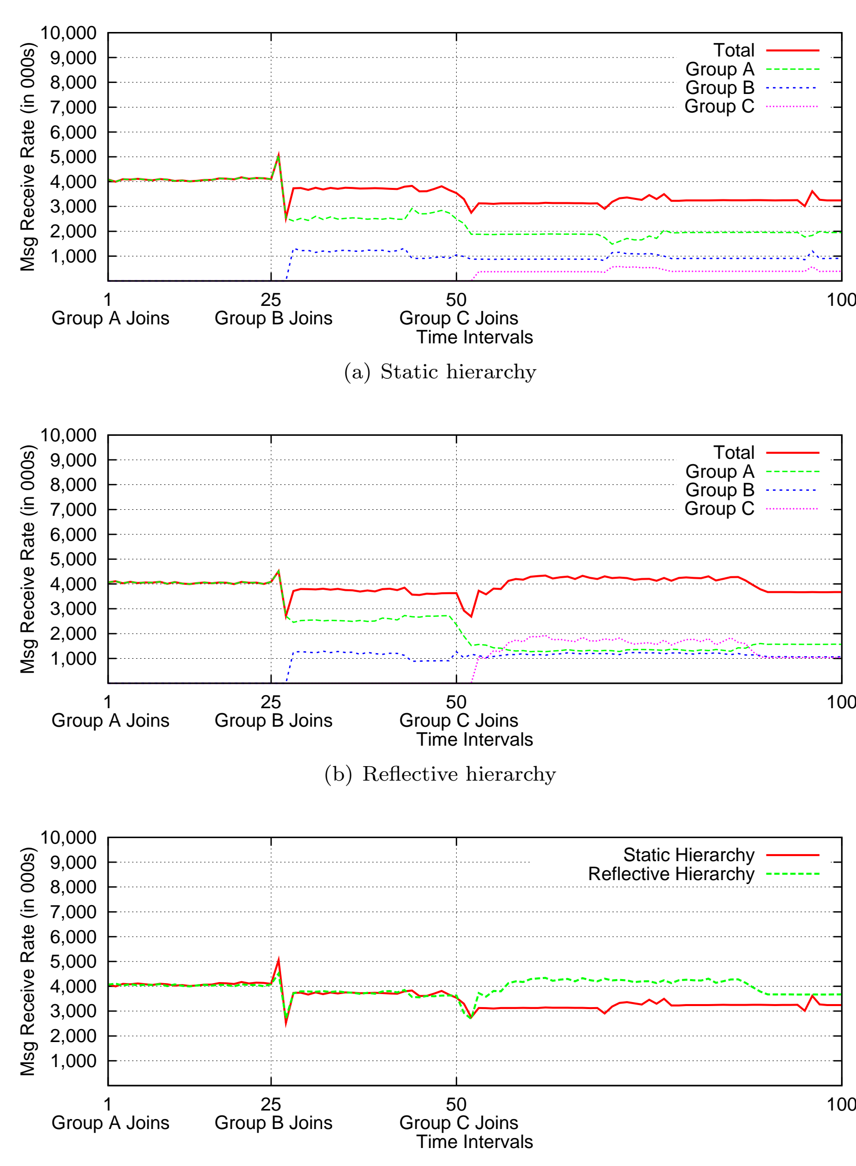
(a) Static hierarchy

(b) Reflective hierarchy

(c) Static hierarchy vs. reflective hierarchy

Figure C.6 *Few-to-many 3 / 15*

C.2.2 30 Producers / 150 Consumers



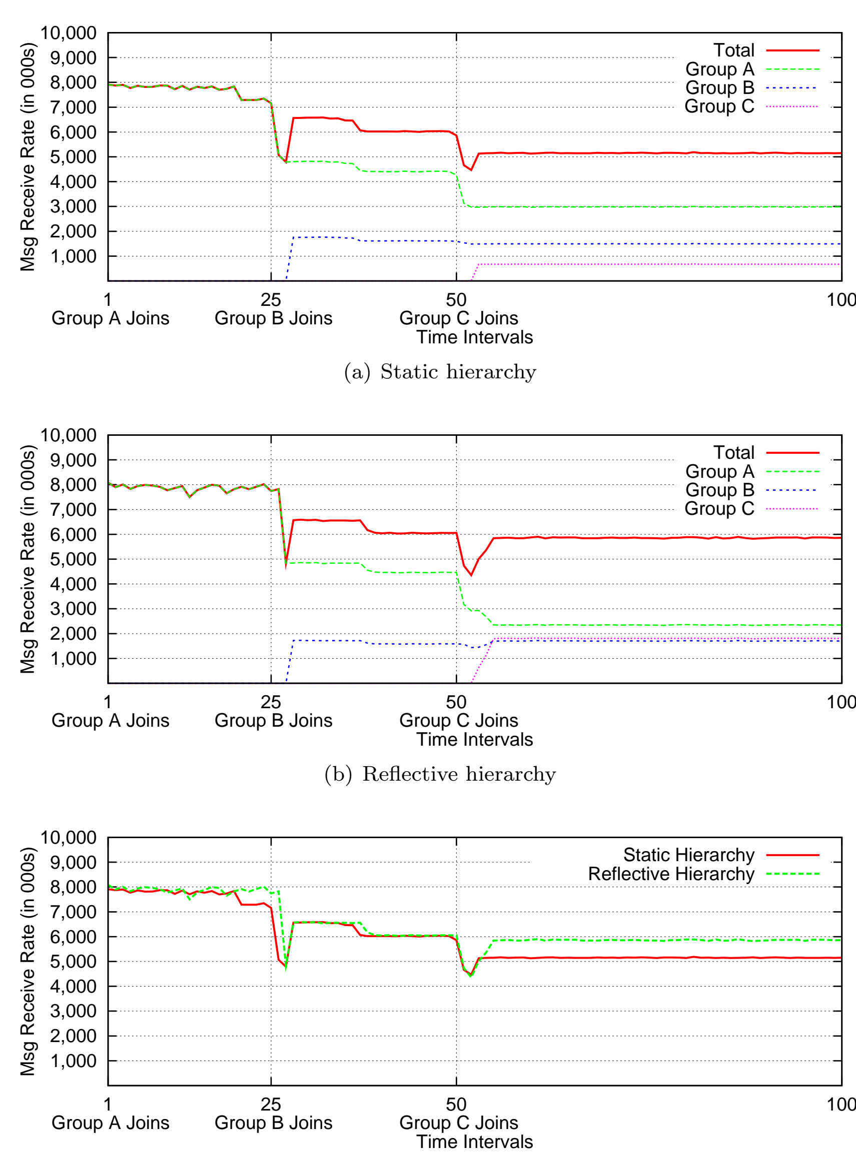
(a) Static hierarchy

(b) Reflective hierarchy

(c) Static hierarchy vs. reflective hierarchy

Figure C.7 *Few-to-many 30 / 150*

C.2.3 60 Producers / 300 Consumers



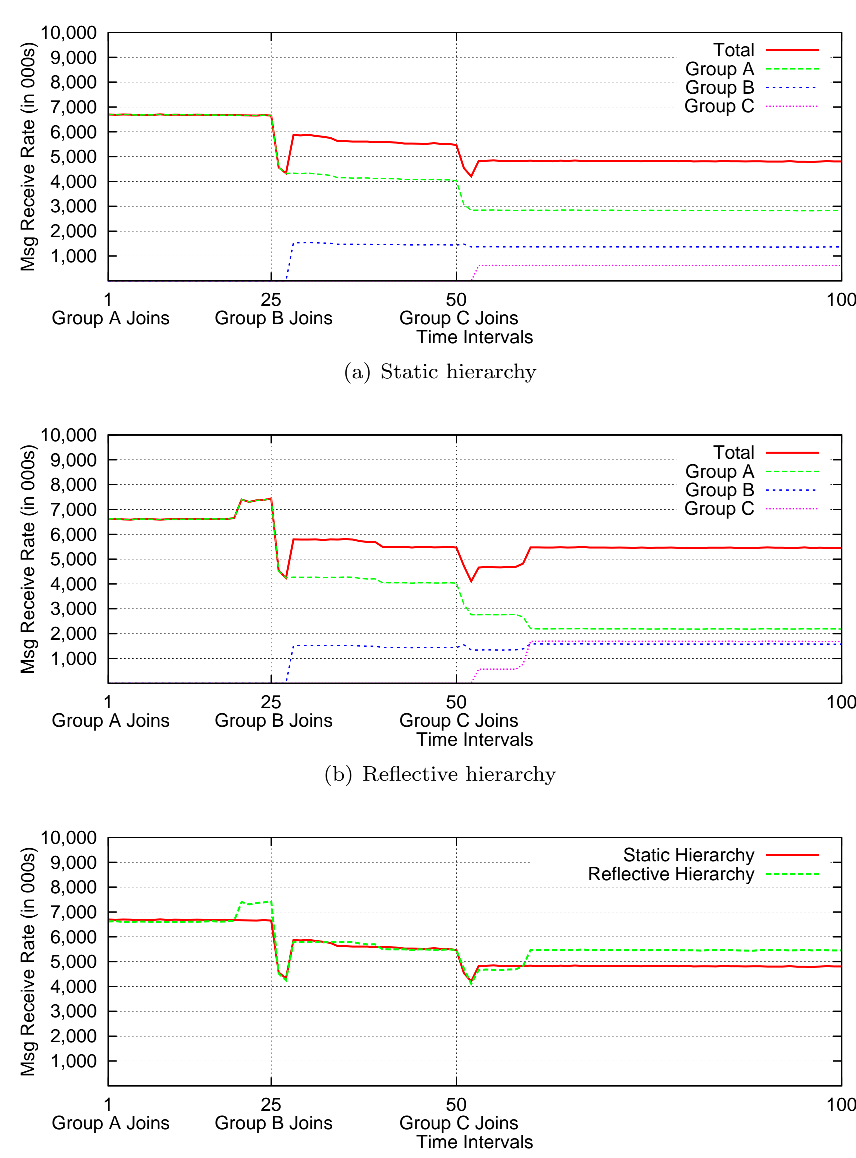
(a) Static hierarchy

(b) Reflective hierarchy

(c) Static hierarchy vs. reflective hierarchy

Figure C.8 *Few-to-many 60 / 300*

C.2.4 90 Producers / 450 Consumers



(a) Static hierarchy

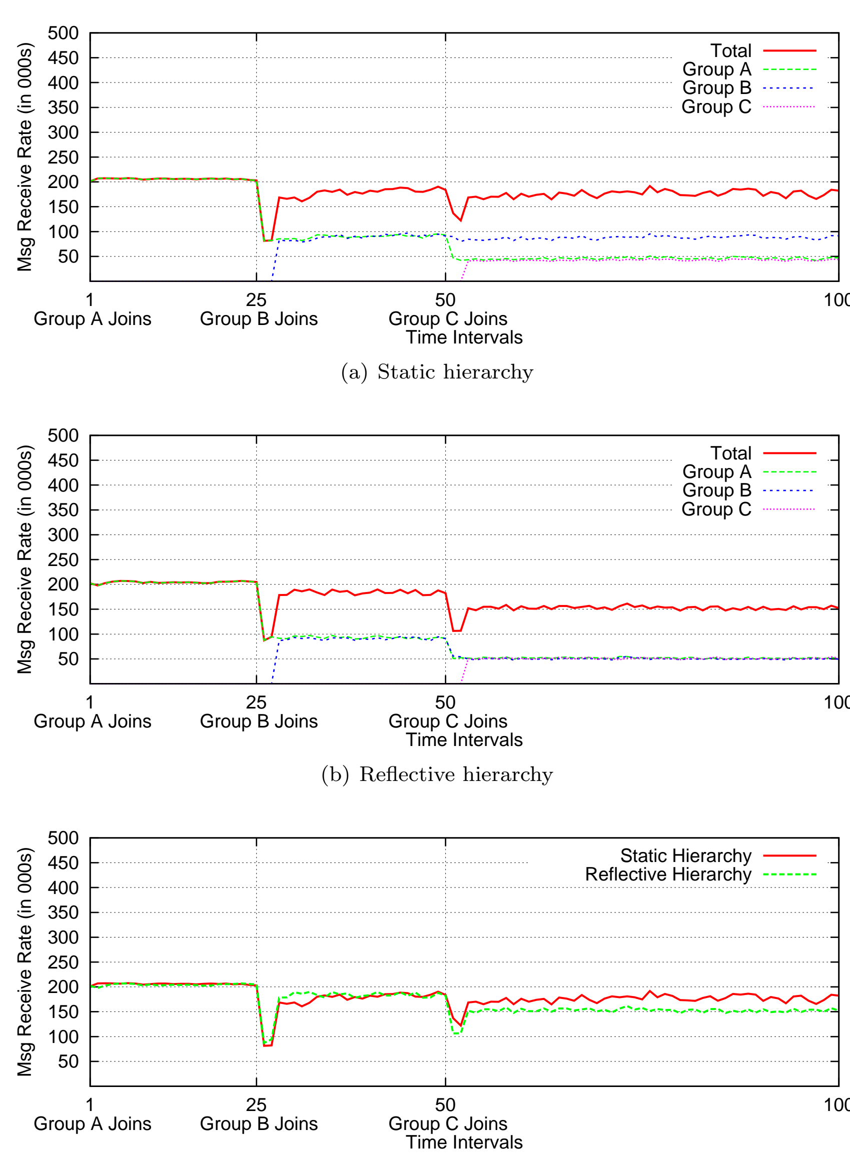
(b) Reflective hierarchy

(c) Static hierarchy vs. reflective hierarchy

Figure C.9 *Few-to-many 90 / 450*

C.3 Many-to-Few Results

C.3.1 15 Producers / 3 Consumers



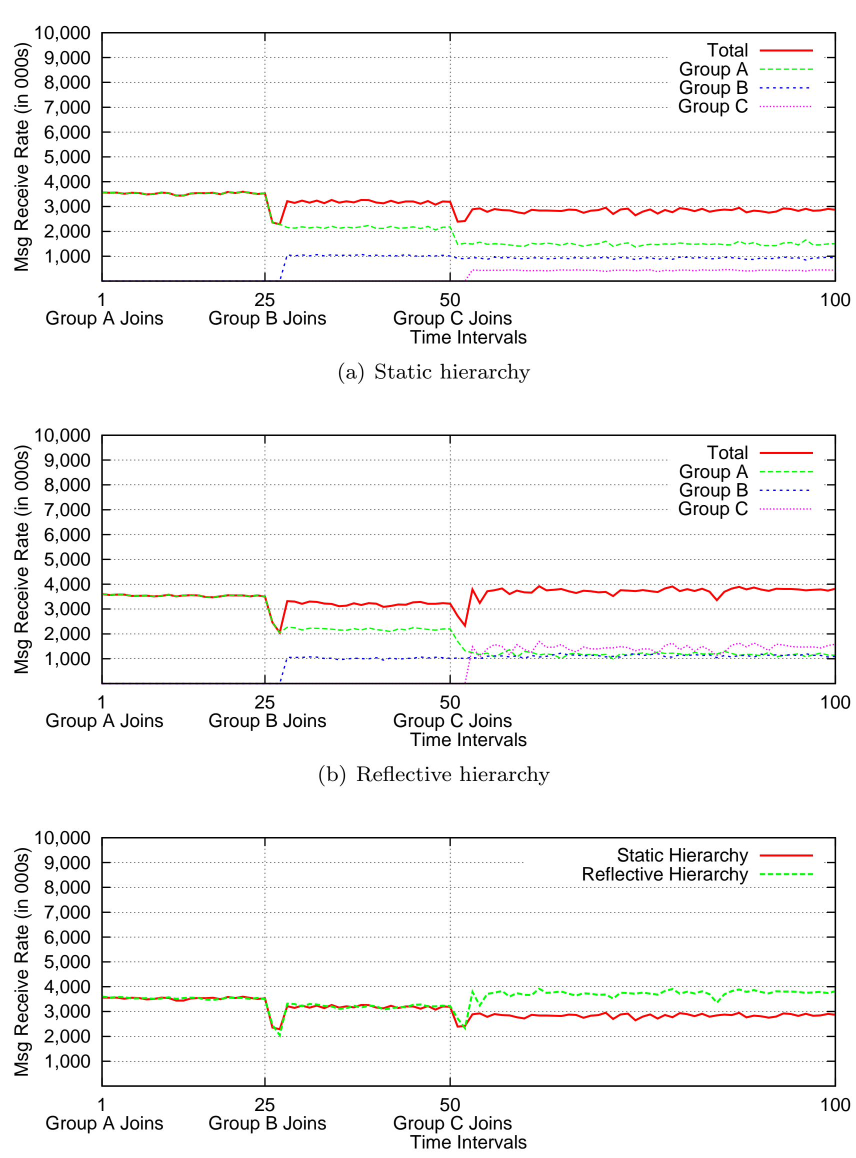
(a) Static hierarchy

(b) Reflective hierarchy

(c) Static hierarchy vs. reflective hierarchy

Figure C.10 *Many-to-few 15 / 3*

C.3.2 150 Producers / 30 Consumers



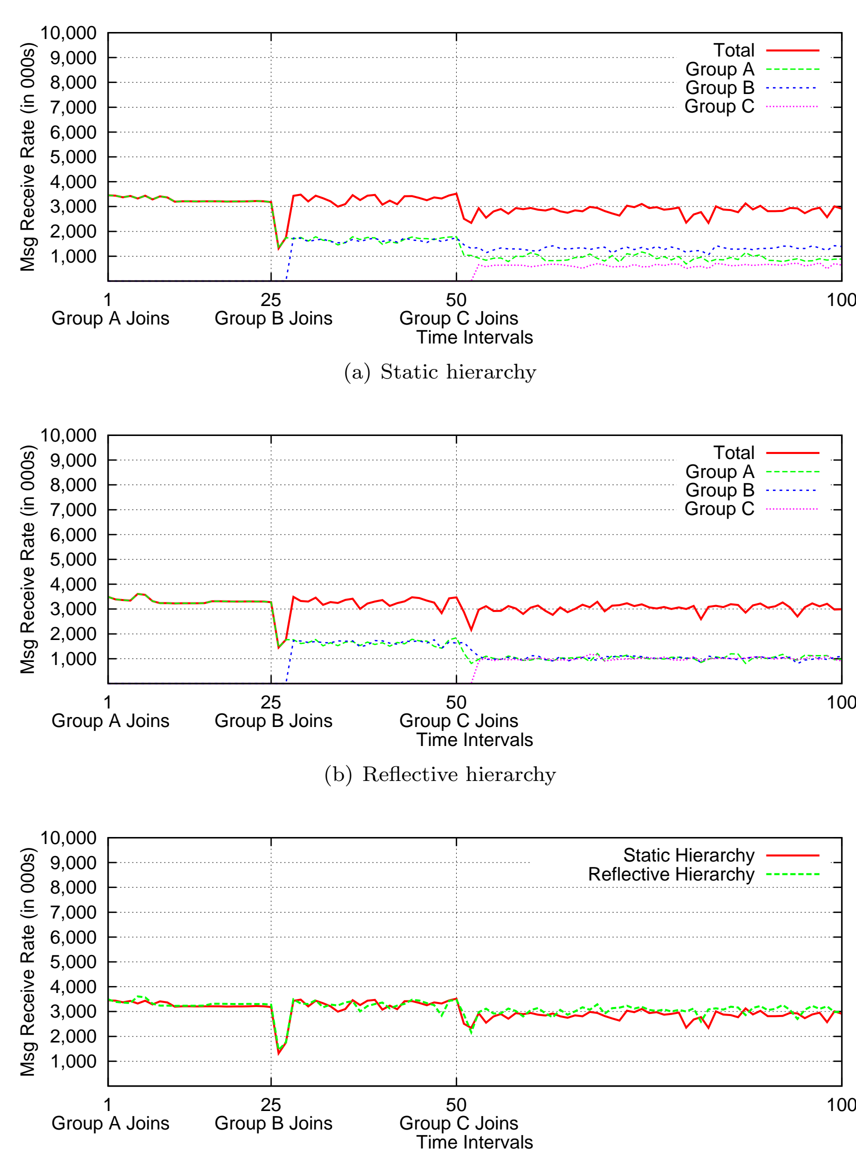
(a) Static hierarchy

(b) Reflective hierarchy

(c) Static hierarchy vs. reflective hierarchy

Figure C.11 *Many-to-few 150 / 30*

C.3.3 300 Producers / 60 Consumers



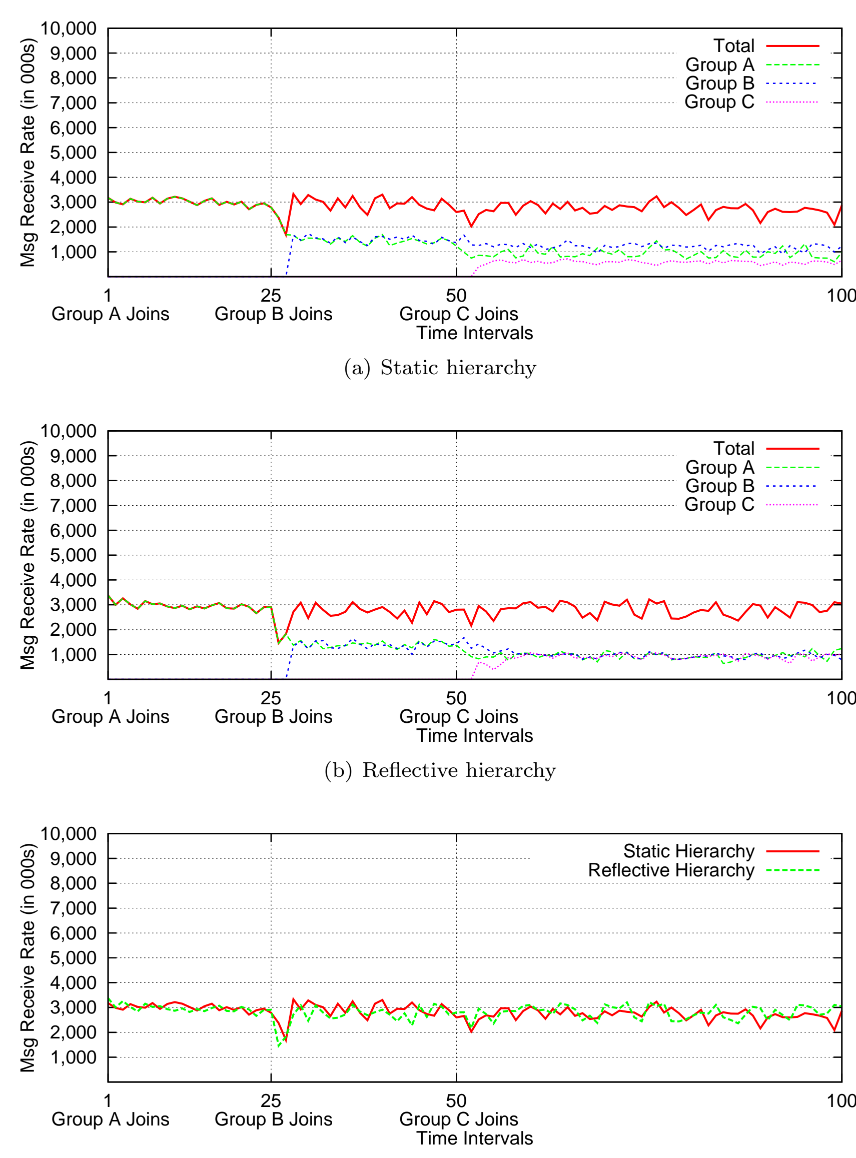
(a) Static hierarchy

(b) Reflective hierarchy

(c) Static hierarchy vs. reflective hierarchy

Figure C.12 *Many-to-few 300 / 60*

C.3.4 450 Producers / 90 Consumers



(a) Static hierarchy

(b) Reflective hierarchy

(c) Static hierarchy vs. reflective hierarchy

Figure C.13 *Many-to-few 450 / 90*

References

[1] K. Geihs, "Middleware Challenges Ahead," *IEEE Computer*, vol. 34, no. 6, pp. 24–31, 2001.

[2] M. Weiser, "Some Computer Science Issues in Ubiquitous Computing," *Communications of the ACM*, vol. 36, no. 7, pp. 74–84, 1993.

[3] L. Kleinrock, "Nomadic Computing," in *Information Network and Data Communication, IFIP/ICCC International Conference on Information Network and Data Communication*, F. A. Aagesen, H. Botnevik, and D. Khakhar, Eds. Trondheim, Norway: Kluwer, 1996, pp. 223–233.

[4] F. P. Brooks, *The Mythical Man-month: Essays on Software Engineering, 20th Anniversary Edition*. Reading, MA: Addison-Wesley, 1995.

[5] R. N. Charette, "Why Software Fails," *IEEE Spectrum*, September 2005.

[6] A. Ganek and T. Corbi, "The Dawning of the Autonomic Computing Era," *IBM Systems Journal*, vol. 42, no. 1, pp. 5–18, 2003.

[7] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski, "The Design and Implementation of Open ORB 2," *IEEE Distributed Systems Online*, vol. 2, no. 6, 2001.

[8] F. Kon, M. Romn, P. Liu, J. Mao, T. Yamane, L. C. Magalhes, and R. H. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*, ser. Lecture Notes in Computer Science, J. S. Sventek and G. Coulson, Eds., vol. 1795. New York, USA: Springer, 2000, pp. 121–143.

[9] A. Andersen, G. S. Blair, T. Stabell-Kulo, P. H. Myrvang, and T.-A. N. Rost, "Reflective Middleware and Security: OOPP meets Obol," in *2nd Workshop on Reflective and Adaptive Middleware at Middleware 2003*. Rio de Janeiro, Brazil: PUC-Rio, 2003, pp. 100–104.

[10] R. Karlsen and A.-B. A. Jakobsen, "Transaction Service Management: An Approach Towards a Reflective Transaction Service," in *2nd Workshop on Reflective and Adaptive Middleware at Middleware 2003*. Rio de Janeiro, Brazil: PUC-Rio, 2003, pp. 135–138.

[11] F. Favarim, F. Siqueira, and J. Fraga, "Adaptive Fault-Tolerant CORBA Components," in *2nd Workshop on Reflective and Adaptive Middleware at Middleware 2003*. Rio de Janeiro, Brazil: PUC-Rio, 2003, pp. 144–148.

[12] R. E. Schantz and D. C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering*. New York: John Wiley and Sons, 2001, pp. 801–813.

[13] J. Donne, *Devotions upon Emergent Occasions*. Oxford University Press, 1987.

[14] G. Hohpe and B. Woolf, "Message Router," in *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003, pp. 78–84.

[15] G. Hohpe and B. Woolf, "Dynamic Router," in *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003, pp. 243–248.

[16] G. Hohpe and B. Woolf, "Content-Based Router," in *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003, pp. 230–236.

[17] G. S. Blair, F. M. Costa, G. Coulson, H. A. Duran, N. Parlavantzas, F. Delpiano, B. Dumant, F. Horn, and J.-B. Stefani, "The Design of a Resource-Aware Reflective Middleware Architecture," in *Second International Conference on Meta-Level Architectures and Reflection (Reflection'99)*, ser. Lecture Notes in Computer Science, P. Cointe, Ed., vol. 1616. St. Malo, France: Springer, 1998, pp. 115–134.

[18] J. Loyall, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, D. Karr, J. M. Gossett, and C. D. Gill, "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications," in *21st International Conference on Distributed Computing Systems*. Mesa, AZ: IEEE Computer Society, 2001, pp. 625–635.

[19] B. C. Smith, "Procedural Reflection in Programming Languages," Ph.D. Thesis, Laboratory of Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 1982.

[20] G. Coulson, "What is Reflective Middleware?" 2002. [Online]. Available: http://dsonline.computer.org

[21] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," *Communications of the ACM*, vol. 45, no. 6, pp. 33–38, 2002.

[22] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato, "Architectural Reflection. Realising Software Architectures via Reflective Activities," in *2nd International Workshop on Engineering Distributed Objects (EDO 2000)*, ser. Lecture Notes in Computer Science, W. Emmerich and S. Tai, Eds., vol. 1999. Davis, USA: Springer, 2000, pp. 102–115.

[23] J. Dowling and V. Cahill, "The K-Component Architecture Meta-model for Self-Adaptive Software," in *Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, ser. Lecture Notes in Computer Science, A. Yonezawa and S. Matsuoka, Eds., vol. 2192. Kyoto, Japan: Springer, 2001, pp. 81–88.

[24] S. Chiba, "A Metaobject Protocol for C++," in *10th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, ser. SIGPLAN. Austin, Texas: ACM Press, 1995, pp. 285–299.

[25] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano, "OpenJava: A Class-Based Macro System for Java," in *Reflection and Software Engineering*, ser. Lecture Notes in Computer Science, W. Cazzola, R. J. Stroud, and F. Tisato, Eds., vol. 1826. Springer, 2000, pp. 117–133.

[26] J. McAffer, "Meta-level programming with CodA," in *9th European Conference on Object-Oriented Programming (ECOOP'95)*, ser. Lecture Notes in Computer Science, W. G. Olthoff, Ed., vol. 952. Aarhus, Denmark: Springer, 1995, pp. 190–214.

[27] J. Dowling, "The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems," Ph.D. thesis, Department of Computer Science, University of Dublin, Trinity College, Dublin, Ireland, 2004.

[28] G. Kiczales, J. d. Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*. Cambridge, Massachusetts: The MIT Press, 1991.

[29] D. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[30] H. Okamura, Y. Ishikawa, and M. Tokoro, "AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework," in *International Workshop on Reflection and Meta-Level Architecture*, A. Yonezawa and B. C. Smith, Eds. Japan: ACM Press, 1992, pp. 36–47.

[31] R. Vanegas, J. A. Zinky, J. P. Loyall, D. Karr, R. E. Schantz, and D. E. Bakken, "QuO's Runtime Support for Quality of Service in Distributed Objects," in *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, N. Davies, K. Raymond, and J. Seitz, Eds. The Lake District, England: Springer, 1998.

[32] W. Cazzola and M. Ancona, "mChaRM: a Reflective Middleware for Communication-Based Reflection," *IEEE Distributed System On-Line*, vol. 3, no. 2, 2002.

[33] E. Curry, D. Chambers, and G. Lyons, "Reflective Channel Hierarchies," in *2nd Workshop on Reflective and Adaptive Middleware at Middleware 2003*. Rio de Janeiro, Brazil: PUC-Rio, 2003, pp. 105–109.

[34] J. Dowling, T. Schaefer, and V. Cahill, "Using Reflection to Support Dynamic Adaptation of System Software: A Case Study Driven Evaluation," in *Software Engineering and Reflection 2000*, ser. Lecture Notes in Computer Science, W. Cazzola, R. J. Stroud, and F. Tisato, Eds., vol. 1826. Denver, Colorado: Springer, 2000, pp. 169–188.

[35] G. Coulson, G. Blair, and P. Grace, "On the Performance of Reflective Systems Software," in *International Workshop on Middleware Performance (MP 2004): Satellite workshop of the IEEE International Performance, Computing and Communications Conference (IPCCC 2004)*. Phoenix, Arizona: IEEE Press, 2004, pp. 763–71.

[36] B. Garbinato, R. Guerraoui, and K. R. Mazouni, "Distributed Programming in GARF," in *ECOOP Workshop on Object-Based Distributed Programming*, R. Guerraoui, O. Nierstrasz, and M. Riveill, Eds., vol. Lecture Notes In Computer Science. Kaiserslautern, Germany: Springer, 1993, pp. 225–239.

[37] B. Garbinato, R. Guerraoui, and K. R. Mazouni, "Implementation of the GARF Replicated Object Plateform," *Distributed Systems Engineering Journal*, vol. 2, no. 1, pp. 14–27, 1995.

[38] J. Dowling and V. Cahill, "Self-Managed Decentralised Systems using K-Components and Collaborative Reinforcement Learning," in *1st ACM SIGSOFT Workshop on Self-managed Systems (WOSS'04)*. Newport Beach, CA, USA: ACM Press, 2004, pp. 39–43.

[39] J. Zinky, R. Schantz, J. Loyall, K. Anderson, and J. Megquier, "The Quality Objects (QuO) Middleware Framework," in *IFIP/ACM (Middleware2000) Workshop on Reflective Middleware*. New York, USA: Springer, 2000.

[40] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier, "Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration," in *Third IEEE International Symposium on Object-Oriented Real Time Distributed Computing*. Newport Beach, California, USA: IEEE Computer Society, 2000, pp. 310–319.

[41] N. Wang, C. Gill, D. Schmidt, A. Gokhale, B. Natarajan, J. Loyall, R. Schantz, and C. Rodrigues, "QoS-enabled Middleware," in *Middleware for Communications*, Q. H. Mahmoud, Ed. Chichester, England: John Wiley and Sons, 2004, pp. 131–188.

[42] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky, "Building Adaptive Distributed Applications with Middleware and Aspects," in *3rd International Conference on Aspect-Oriented Software Development (AOSD 04)*. Lancaster, UK: ACM Press, 2004, pp. 66–73.

[43] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *11th European Conference on Object-Oriented Programming (ECOOP'97)*, ser. Lecture Notes in Computer Science, M. Aksit and S. Matsuoka, Eds., vol. 1241. Jyväskylä, Finland: Springer, 1997, pp. 220–242.

[44] L. Bergmans and M. Aksit, "Aspects and crosscutting in layered middleware systems," in *IFIP/ACM (Middleware2000) Workshop on Reflective Middleware*. New York, USA: Springer, 2000.

[45] S. Williams and C. Kindel, "The Component Object Model: Technical Overview," *Dr. Dobbs Journal*, December 1994.

[46] N. Parlavantzas, G. Blair, and G. Coulson, "A Resource Adaptation Framework for Reflective Middleware," in *2nd Workshop on Reflective and Adaptive Middleware at Middleware 2003*. Rio de Janeiro, Brazil: PUC-Rio, 2003, pp. 163–168.

[47] N. Parlavantzas, G. Coulson, and G. Blair, "An Extensible Binding Framework for Component-based Middleware," in *7th IEEE International Enterprise Distributed Object Computing Conference (EDOC'03)*. Brisbane, Australia: IEEE Computer Society, 2003, pp. 252–263.

[48] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Special Issue on Design Patterns*, vol. 37, no. 4, pp. 54–63, 1999.

[49] F. Kon, B. Gill, M. Anand, R. H. Campbell, and M. D. Mickunas, "Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents," in *IEEE Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA'2000)*, ser. Lecture Notes in Computer Science, D. Kotz and F. Mattern, Eds., vol. 1882. Zurich, Switzerland: Springer, 2000, pp. 86–98.

[50] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. thesis, Deprtment of Information and Computer Science, University of California, Irvine, California, 2000.

[51] R. Glassey, G. Stevenson, M. Richmond, P. Nixon, S. Terzis, F. Wang, and I. Ferguson, "Towards a Middleware for Generalised Context Management," in *1st International Workshop on Middleware for Pervasive and Ad-Hoc Computing at Middleware 2003*. Rio de Janeiro, Brazil: PUC-Rio, 2003, pp. 45–52.

[52] A. R. Portillo, S. Walker, G. Kirby, and A. Dearle, "A Reflective Approach to Providing Flexibility in Application Distribution," in *2nd Workshop on Reflective and Adaptive Middleware at Middleware 2003*. Rio de Janeiro, Brazil: PUC-Rio, 2003, pp. 95–99.

[53] A. Maurino, S. Modaeri, and B. Pernici, "Reflective Architectures for Adaptive Information Systems," in *International Conference on Service-Oriented Computing*, ser. Lecture Notes in Computer Science, M. E. Orlowska, S. Weerawarana, M. P. Papazoglou, and J. Yang, Eds., vol. 2910. Trento, Italy: Springer, 2003, pp. 115–131.

[54] J. E. White, "A High-Level Framework for Network-Based Resource Sharing," in *AFIPS National Computer Conference*, 1976, pp. 561–570.

[55] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39–54, 1984.

[56] G. Banavar, T. Chandra, R. E. Strom, and D. C. Sturman, "A Case for Message Oriented Middleware," in *13th International Symposium on Distributed Computing*, ser. Lecture Notes In Computer Science, P. Jayanti, Ed., vol. 1693. Bratislava, Slovak Republic: Springer, 1999, pp. 1–18.

[57] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms*, 1st ed. Prentice Hall, January 15, 2002.

[58] B. Naughton, S. Cranton, C. King, N. Nagarajaya, T. Schmal, D. Vleugels, and D. Winstone, "Deployment Strategies focusing on Massive Scalability," Massive Scalability Focus Group - Operations Support Systems (OSS) through Java Initiative, Tech. Rep., 25 April 2003. [Online]. Available: http://java.sun.com/products/oss/pdf/MassiveScalability-1.0.pdf

[59] E. Curry, D. Chambers, and G. Lyons, "A JMS Message Transport Protocol for the JADE Platform," in *2003 IEEE/WIC International Conference on Intelligent Agent Technology (IAT'03)*. Halifax, Canada: IEEE Computer Society, 2003, pp. 596–600.

[60] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, 1991.

[61] A. Hinze and S. Bittner, "Efficient Distribution-Based Event Filtering," in *1st International Workshop on Distributed Event-Based Systems (DEBS'02)*. Vienna, Austria: IEEE Computer Society, 2002, pp. 525–532.

[62] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, 2001.

[63] E. Curry, D. Chambers, and G. Lyons, "Enterprise Service Facilitation within Agent Environments," in *8th IASTED International Conference on Software Engineering and Applications (SEA 2004)*, M. Hamza, Ed. MIT Cambridge, MA, USA: IASTED Press, 2004.

[64] P. R. Pietzuch and J. M. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture," in *1st International Workshop on Distributed Event-Based Systems (DEBS'02)*. Vienna, Austria: IEEE Computer Society, 2002, pp. 611–618.

[65] G. Mühl and L. Fiege, "Supporting Covering and Merging in Content-Based Publish/Subscribe Systems: Beyond Name/Value Pairs," *IEEE Distributed Systems Online*, vol. 2, no. 7, 2001.

[66] P. R. Pietzuch, B. Shand, and J. Bacon, "A Framework for Event Composition in Distributed Systems," in *ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, ser. Lecture Notes in Computer Science, M. Endler and D. C. Schmidt, Eds., vol. 2672. Rio de Janeiro, Brazil: Springer, 2003, pp. 62–82.

[67] A. Carzaniga, "Architectures for an Event Notification Service Scalable to Wide-area Networks," Ph.D. thesis, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italy, 1998.

[68] G. Mühl, "Generic constraints for content-based publish/subscribe systems," in *6th International Conference on Cooperative Information Systems (CoopIS 2001)*, ser. Lecture Notes In Computer Science, C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, Eds., vol. 2172. Trento, Italy: Springer, 2001, pp. 211–225.

[69] G. Mühl, L. Fiege, and A. P. Buchmann, "Filter Similarities in Content-Based Publish/Subscribe Systems," in *International Conference on Architecture of Computing Systems (ARCS'02): Trends in Network and Pervasive Computing*, ser. Lecture Notes In Computer Science, vol. 2299. Karlsruhe, Germany: Springer, 2002, pp. 224–240.

[70] L. Gilman and R. Schreiber, *Distributed Computing with IBM MQSeries*. New York: John Wiley, 1996.

[71] D. Skeen, "An Information Bus Architecture for Large-Scale, Decision-Support Environments," in *Proceedings of the Winter 1992 USENIX Conference*. San Francisco, California: USENIX Association, 1992, pp. 183–195.

[72] "Sonic MQ," 2005. [Online]. Available: http://www.sonicmq.com

[73] L. F. Cabrera, M. B. Jones, and M. Theimer, "Herald: Achieving a Global Event Notification Service," in *8th Workshop on Hot Topics in Operating Systems*. Elmau, Germany: IEEE Computer Society, 2001, pp. 87–94.

[74] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward, "Gryphon: An Information Flow Based Approach to Message Brokering," in *International Symposium on Software Reliability Engineering*. Paderborn, Germany: IEEE Press, 1998.

[75] G. Cugola, E. D. Nitto, and A. Fuggetta, "The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS," *IEEE Transactions on Software Engineering*, vol. 27, no. 9, pp. 827–850, 2001.

[76] L. Fiege and G. Mühl, "REBECA," 2005. [Online]. Available: http://www.gkec.informatik.tu-darmstadt.de/rebeca/

[77] "OpenJMS," 2005. [Online]. Available: http://openjms.sourceforge.net/

[78] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, "Java Message Service Specification," Sun Microsystems, Inc., std. 1.1, 2002.

[79] K. Haase, "The Java Message Service (JMS) Tutorial," 2002. [Online]. Available: http://java.sun.com/products/jms/tutorial/

[80] R. Monson-Haefel and D. A. Chappell, *Java Message Service*. Sebastopol, CA: O'Reilly and Associates, 2001.

[81] E. Curry, "Message-Oriented Middleware," in *Middleware for Communications*, Q. H. Mahmoud, Ed. Chichester, England: John Wiley and Sons, 2004, pp. 1–28.

[82] S. Tai, T. A. Mikalsen, and I. M. Rouvellou, "Transaction Middleware," in *Middleware for Communications*, Q. H. Mahmoud, Ed. Chichester, England: John Wiley and Sons, 2004, pp. 53–80.

[83] P. R. Pietzuch, "Hermes: A Scalable Event-Based Middleware," Ph.D. thesis, Computer Laboratory, Queens' College, University of Cambridge, Cambridge, England, 2004.

[84] *SonicMQ V6.1 Administrative Programming Guide*. Sonic Software, 2004.

[85] "ActiveMQ," 2005. [Online]. Available: http://activemq.codehaus.org/

[86] "CORBA: Event Service Specification," Object Management Group, std. 1.2, 2004. [Online]. Available: http://www.omg.org/technology/documents/formal/event_service.htm

[87] "CORBA: Notification Service Specification," Object Management Group, std. 1.1, 2004. [Online]. Available: http://www.omg.org/technology/documents/formal/notification_service.htm

[88] "TIBCO Rendezvous: Concepts 7.3," TIBCO Software, Tech. Rep., 2004. [Online]. Available: http://developer.tibco.com/

[89] B. Ban, "JGroups," 2005. [Online]. Available: http://www.jgroups.org/

[90] "Java Reliable Multicast Service," 2005. [Online]. Available: http://www.experimentalstuff.com/Technologies/JRMS/

[91] H. Hrasna, "Java 2 Platform, Enterprise Edition Management Specification JSR-77," Sun Microsystems, std. 1.0, 2002. [Online]. Available: http://jcp.org/jsr/detail/77.jsp

[92] "High Performance Messaging with JMS: A Benchmark Comparison of SonicMQ. 4.0 vs. IBM MQSeries. 5.2," Jahming Technologies, Tech. Rep. 1.0, 2001. [Online]. Available: http://www.sonicsoftware.com/products/whitepapers/docs/sonic40_vs_mqseries52.pdf

[93] "Clustering and DRA in SonicMQ," Sonic Software, Tech. Rep. 1.0, 2004. [Online]. Available: http://www.sonicsoftware.com/solutions/learning_center/whitepapers

[94] G. Mühl, "Large-Scale Content-Based Publish/Subscribe Systems," Ph.D. thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, 2002.

[95] L. Fiege, G. Mühl, and F. C. Grtner, "A modular approach to build structured event-based systems," in *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02).* Madrid, Spain: ACM Press, 2002, pp. 385–392.

[96] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann, "Engineering Event-Based Systems with Scopes," in *16th European Conference on Object-Oriented Programming (ECOOP'02)*, ser. Lecture Notes in Computer Science, B. Magnusson, Ed., vol. 2374. Malaga, Spain: Springer, 2002, pp. 309–333.

[97] L. Fiege, F. C. Grtner, O. Kasten, and A. Zeidler, "Supporting Mobility in Content-Based Publish/Subscribe Middleware," in *ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, ser. Lecture Notes in Computer Science, M. Endler and D. C. Schmidt, Eds., vol. 2672. Rio de Janeiro, Brazil: Springer, 2003, pp. 103–122.

[98] E. McManus, "Java Management Extensions Instrumentation and Agent Specification," Sun Microsystems, Inc., std. 1.2, 2002.

[99] L. Fiege, "REBECA Management Infrastructure," 2005, personal correspondence.

[100] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman, "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems," in *9th International Conference on Distributed Computing Systems (ICDS'99).* Austin, Texas, USA: IEEE Computer Society, 1999, pp. 262–272.

[101] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, "Matching Events in a Content-based Subscription System," in *18th ACM Symposium on Principles of Distributed Computing.* Atlanta, Georgia, USA: ACM Press, 1999, pp. 53–61.

[102] P. R. Pietzuch and S. Bhola, "Congestion Control in a Reliable Scalable Message-Oriented Middleware," in *ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, ser. Lecture Notes in Computer Science, M. Endler and D. C. Schmidt, Eds., vol. 2672. Rio de Janeiro, Brazil: Springer, 2003, pp. 202–221.

[103] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiter, "Generic Support for Distributed Applications," *IEEE Computer*, vol. 33, no. 3, pp. 68–76, 2000.

[104] M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul, "Filtering and Scalability in the ECO Distributed Event Model," in *International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE2000)*. Limerick, Ireland: IEEE CS Press, 2000, pp. 83–92.

[105] M. A. Jaeger, "Self-Organizing Publish/Subscribe," *IEEE Distributed Systems Online*, vol. 7, no. 2, 2006.

[106] M. A. Jaeger, "Self-Organizing Publish/Subscribe," in *2nd International Doctoral Symposium on Middleware*, ser. ACM International Conference Proceeding, vol. 114. Grenoble, France: ACM Press, 2005, pp. 1–5.

[107] FIPA, "FIPA - Foundation for Intelligent Physical Agents." [Online]. Available: www.fipa.org

[108] "FIPA Agent Message Transport Service Specification," Foundation for Intelligent Physical Agents, std. SC00067F, 2002. [Online]. Available: http://www.fipa.org/specs/fipa00067/SC00067F.pdf

[109] "FIPA Request Interaction Protocol Specification," Foundation for Intelligent Physical Agents, std. SC00026H, 2002. [Online]. Available: http://www.fipa.org/specs/fipa00026/SC00026H.pdf

[110] "FIPA RDF Content Language Specification," Foundation for Intelligent Physical Agents, std. XC00011B, 2001. [Online]. Available: http://www.fipa.org/specs/fipa00011/XC00011B.pdf

[111] E. Curry and E. Ridge, "The Collective: A Common Information Service for Self-Managed Middleware," in *4th Workshop on Adaptive and Reflective Middleware at Middleware 2005*, ser. ACM International Conference Proceeding Series, vol. 116. Grenoble, France: ACM Press, 2005, pp. 1–6.

[112] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley and Sons, 2000, vol. 2.

[113] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[114] M. Fleury and F. Reverbel, "The JBoss Extensible Server," in *ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, ser. Lecture Notes in Computer Science, M. Endler and D. C. Schmidt, Eds., vol. 2672. Rio de Janeiro, Brazil: Springer, 2003, pp. 344–373.

[115] G. Booch, "Through the Looking Glass," *Software Development*, July 2001.

[116] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. r. Palm, and W. G. Griswold, "An Overview of AspectJ," in *The 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, ser. Lecture Notes In Computer Science, J. L. Knudsen, Ed., vol. 2072. Budapest, Hungary: Springer, 2001, pp. 327–353.

[117] H. Ossher and P. Tarr, "Using multidimensional separation of concerns to (re)shape evolving software," *Communications of the ACM*, vol. 44, no. 10, pp. 43–50, 2001.

[118] S. Vinoski, "A Time for Reflection," *IEEE Internet Computing*, vol. 9, no. 1, pp. 86–89, 2005.

[119] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications.* New York: Addison-Wesley, 2000.

[120] C. Cleaveland, *Program Generators with XML and Java.* Upper Saddle River, New Jersy: Prentice Hall, 2001.

[121] D. M. Chess, “Security issues in mobile code systems,” in *Mobile Agents and Security*, ser. Lecture Notes in Computer Science, G. Vigna, Ed. Springer, 1998, vol. 1419, pp. 1–14.

[122] “BEA Systems,” 2005. [Online]. Available: http://www.bea.com/

[123] D. A. Chappell, *Enterprise Service Bus.* Sebastopol, CA: O’Reilly and Associates, 2004.

[124] P. T. Eugster, P. Felber, R. Guerraoui, and S. B. Handurukande, “Event Systems: How to Have Your Cake and Eat It Too,” in *International Workshop on Distributed Event-Based Systems (DEBS’02).* Vienna, Austria: IEEE Computer Society, 2002, pp. 625–632.

[125] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps, “Content Based Routing with Elvin4,” in *The Australian UNIX and Open Systems Users Group (AUUG2K).* Canberra, Australia: ACM Press, 2000.

[126] E. Curry, D. Chambers, and G. Lyons, “Extending Message-Oriented Middleware using Interception,” in *Third International Workshop on Distributed Event-Based Systems (DEBS ’04) at ICSE ’04*, A. Carzaniga and P. Fenkam, Eds. Edinburgh, Scotland, UK: IEEE Computer Society, 2004, pp. 32–37.

[127] E. Curry, D. Chambers, and G. Lyons, “ARMAdA: Creating a Reflective Fellowship (Options for Interoperability),” in *3rd Workshop on Adaptive and Reflective Middleware at Middleware 2004.* Toronto, Canada: Springer, 2004, pp. 226–231.

[128] E. Curry, D. Chambers, and G. Lyons, “Could Message Hierarchies Contemplate?” in *17th European Conference on Object-Oriented Programming (ECOOP 2003)*, Darmstadt, Germany, 2003, (poster).

[129] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* Cambridge, MA: The MIT Press, 1998.

[130] A. Carzaniga and A. L. Wolf, “A Benchmark Suite for Distributed Publish/Subscribe Systems,” Department of Computer Science, University of Colorado, Tech. Rep. CU-CS-927-02, 2002.

[131] “Benchmarking E-Business Messaging Providers,” Sonic Software, Tech. Rep. 1.0, 2004. [Online]. Available: http://www.sonicsoftware.com/solutions/learning_center/whitepapers

[132] M. Pang and P. Maheshwari, “Benchmarking Message-Oriented Middleware – TIB/RV vs. SonicMQ,” in *Workshop on Foundations of Middleware Technologies*, Irvine, CA, 2002.

[133] P. Maheshwari and M. Pang, “Benchmarking Message-Oriented Middleware – TIB/RV vs. SonicMQ,” *Concurrency and Computation: Practice and Experience*, vol. 17, no. 12, pp. 1507–1526, 2005.

[134] E. Curry, “Adaptive and Reflective Middleware,” in *Middleware for Communications*, Q. H. Mahmoud, Ed. Chichester, England: John Wiley and Sons, 2004, pp. 29–52.

[135] F. Bellifemine, A. Poggi, and G. Rimassa, “JADE - A FIPA-Compliant Agent Framework,” in *4th International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-99).* London, England: The Practical Application Company Ltd., 1999, pp. 97–108.

[136] "FIPA Subscribe Interaction Protocol Specification," Foundation for Intelligent Physical Agents, std. SC00035, 2002. [Online]. Available: http://www.fipa.org/specs/fipa00035/SC00035H.pdf

[137] M. Cilia, M. Antolliniy, C. Bornhovdz, and A. Buchmann, "Dealing with Heterogeneous Data in Pub/Sub Systems:The Concept-Based Approach," in *Third International Workshop on Distributed Event-Based Systems (DEBS '04) at ICSE '04*, A. Carzaniga and P. Fenkam, Eds. Edinburgh, Scotland, UK: IEEE Computer Society, 2004, pp. 26–31.

[138] "FioranoMQ 7.1 vs. SonicMQ 5.0.2: A comprehensive technical evaluation of SonicMQ 5.0.2," Fiorano Software, Tech. Rep. 1.0, 2003. [Online]. Available: http://www.fiorano.com/whitepapers/performance_comparision_sonic.pdf

[139] "SonicMQ vs. TIBCO Enterprise for JMS: Performance Comparison: Publish/Subscribe Messaging," Sonic Software, Tech. Rep. 1.0, 2003. [Online]. Available: http://www.sonicsoftware.com/products/whitepapers/docs/jms_comparison_tibco.pdf