

Runtime Support for Quality of Information Requirements in Event-based Systems

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.) vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte Dissertation von Diplom-Wirtschaftsinformatiker Sebastian Frischbier aus Darmstadt (Hessen) März 2016 — Darmstadt — D 17



TECHNISCHE
UNIVERSITÄT
DARMSTADT



DVS

Runtime Support for Quality of Information Requirements in Event-based Systems

Vom Fachbereich Informatik
der Technischen Universität Darmstadt

zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation

von Diplom-Wirtschaftsinformatiker Sebastian Frischbier aus Darmstadt (Hessen)

1. Gutachten: Prof. Alejandro Buchmann, Ph.D.
2. Gutachten: Prof. Peter Pietzuch, Ph.D.

Tag der Einreichung: 03.02.2016

Tag der Prüfung: 17.03.2016

Darmstadt — D 17

Bitte zitieren Sie dieses Dokument als:

URN: urn:nbn:de:tuda-tuprints-53663

URL: <http://tuprints.ulb.tu-darmstadt.de/5366>

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
<http://tuprints.ulb.tu-darmstadt.de>
tuprints@ulb.tu-darmstadt.de



Die Veröffentlichung steht unter folgender Creative Commons Lizenz:

Namensnennung – Keine kommerzielle Nutzung – Keine Bearbeitung 3.0 Deutschland
<http://creativecommons.org/licenses/by-nc-nd/3.0/de/>

To Mareike



Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 03.02.2016

(Sebastian Frischbier)

Curriculum Vitae

Sebastian Frischbier

Education

- 2009–2016 PhD Candidate
Technischen Universität Darmstadt
Databases and Distributed Systems Group (DVS)
- 2002–2009 Joint Masters Degree in Management and Computing
(Diplom-Wirtschaftsinformatiker)
Technische Universität Darmstadt
- 2001 Abitur
Eleonorenschule Darmstadt

Academic Positions

- 2009–2015 Research and Teaching Assistant
Technische Universität Darmstadt
Databases and Distributed Systems Group (DVS)
- 2013 Visiting Researcher
Imperial College London, UK
Large-Scale Distributed Systems Group (LSDS)

Fellowships

- 2013–2015 Software Campus
- 2002–2008 Studienstiftung des deutschen Volkes

Contents

1. Introduction	1
1.1. Problem Statement	2
1.2. Research Questions	6
1.3. Proposed Solution, Scope and Contributions	6
1.4. Related Activities and Publications	10
1.5. Structure	12
2. Background	13
2.1. Event-based Systems	13
2.1.1. Publish/Subscribe for Dispatching Event Notifications	15
2.1.2. Complex Event Processing for Reasoning and Deciding on Events	17
2.2. Related Concepts and the Role of EBS	17
2.2.1. Wireless Sensor Networks and Cyber-Physical Systems	17
2.2.2. Data Stream Management Systems	18
2.2.3. Service-Oriented Architectures	19
2.2.4. Cloud Computing	20
2.2.5. Systems-of-Systems and the Vision of Emergent Software Systems	22
2.3. Summary	23
3. A Generic Model to Express Quality of Information Requirements in EBS	25
3.1. Quality of Information and Related Concepts	26
3.2. Properties: the Basic Building Blocks	28
3.2.1. Categorization of Properties	32
3.2.2. Deriving a Generic Property Representation	40
3.3. Expectations: Requirements About Quality of Information (QoI) Properties	46
3.3.1. Expectation Definition	48
3.3.2. Lifecycle of an Expectation	52
3.3.3. Fidelity: Quantifying the Satisfaction of a Subscriber	53
3.4. Capabilities: Support for Properties	60
3.4.1. Capability Definition: Spectrum of Support and Costs	61
3.4.2. Capability Profiles: Characterizing Publishers	64
3.4.3. Lifecycle of Capabilities and Capability Profiles	66
3.5. Feedback: Enabler of Self-Adaptation and Renegotiation	68
3.5.1. Individual Feedback	68
3.5.2. Aggregated Feedback	71
3.6. Summary	77
4. Runtime Negotiation and Enforcement	79
4.1. Matching Expectations to Capabilities	80
4.1.1. Set-Matching to Find Suitable Capability Profiles	81
4.1.2. Range-Matching for Each Generic Property	82
4.1.3. Determining the Preliminary State of an Expectation	85

4.2. Deciding on Satisfiable Expectations	87
4.2.1. Decision Strategies Encapsulating the Decision Process	88
4.2.2. Example Strategy: First-Come, First-Served while Minimizing Costs	89
4.3. Safeguarding the Decision	90
4.4. Runtime Adaptation	93
4.4.1. Middleware Self-Adaptation	93
4.4.2. Client Self-Adaptation Using Feedback	94
4.4.3. Coordinated Adaptation	95
4.5. Monitoring the System State at Runtime	98
4.5.1. Detect and React to State Changes with Monitorlets and Watchdogs	98
4.5.2. Monitoring the Global State of a Distributed Event-Based System	100
4.6. Resolving Possible Conflicts at Runtime	107
4.7. Summary	109
5. Implementation	111
5.1. Architecture and Design	112
5.1.1. Extending the Message-Oriented Middleware: ExpectationController	112
5.1.2. Decentralized Monitoring with ASIA	118
5.1.3. Libraries, Handlers and Editors Provided to Clients	119
5.1.4. Classes for Expectations, Capabilities and Generic Properties	123
5.2. Platform-Specific Prototypes	125
5.2.1. Centralized Implementation: Apache ActiveMQ	125
5.2.2. Distributed Implementation: REDS	131
5.3. Example Applications	135
5.4. Summary	142
6. Evaluation	143
6.1. Expressivity of Expectations and Capabilities	144
6.1.1. Related Approaches and their Expressiveness	144
6.1.2. Summarizing the Limitations of Related Approaches	149
6.1.3. Expressiveness of Expectations and Capabilities	150
6.2. Benefits Regarding Data Quality and Resource Savings	151
6.2.1. Heterogeneity Scenario: Dealing with Unsuitable Data	151
6.2.2. Comparing Expectations with Features of Related Approaches	157
6.2.3. Benefits: Higher Data Quality	159
6.2.4. Benefits: Resource Savings	165
6.3. Scalability and Execution Costs for Brokers Applying our Approach	169
6.3.1. Used Scenario and Characterization of Workload	169
6.3.2. Tailoring jms2009-PS to Gauge Execution Costs of Runtime Negotiation . .	171
6.3.3. Testplan, Scaling Parameters and Measured KPIs	172
6.3.4. Discussion of Measured Results	174
6.4. Effectiveness of Using ASIA to Monitor a DEBS	188
6.4.1. Benefits: Adjustable Precision	189
6.4.2. Experiment Setup for Gauging Traffic, Throughput and Latency	190
6.4.3. Impact on Network Traffic	190
6.4.4. Execution Costs: Throughput and Latency	193
6.5. Summary	197

7. Related work	199
7.1. Standards and Protocols for Asynchronous Communication	200
7.2. Related Approaches with Explicit Support for Qo*	200
7.3. Related Approaches Regarding Monitoring, Self-Adaptation, Negotiation	202
7.4. Summary	205
8. Conclusion	207
9. Outlook and Future Work	209
9.1. Centralized Event-Based Systems: Immediate Challenges	209
9.2. Decentralized Event-Based Systems: Synchronizing State	210
9.3. Applying our Approach to Pull-Based Systems	212
9.4. Economic Perspective: Incentives and Negotiation on Two-Sided Markets	218
A. Appendix	219
A.1. Literature Survey Details	220
A.2. Runtime Negotiation: Pseudocode	228
A.3. Reference Architecture: APIs and Code Examples	231
A.4. FINCoS: Extensions and Experimental Setup	237
A.4.1. Test Harness for Automated Testing	238
A.4.2. Test Setup	239
A.4.3. Anatomy of a Single Test Run	240
A.5. Drill Down Data for Heterogeneity Scenario with Surplus Publishers	242
A.6. jms2009-PS Extensions and Experimental Setup	244
A.6.1. Test Harness for Automated Testing	244
A.6.2. Anatomy of a Single Run	245
A.7. Regression Tables jms2009-PS Benchmark Results	247
A.8. Experimental Setup Application-Specific Integrated Aggregation (ASIA)	249
B. Bibliography	251

List of Definitions

1.	Definition (Generic property)	43
2.	Definition (Action)	44
3.	Definition (Expectation)	48
4.	Definition (Capability)	61
5.	Definition (Capability Profile)	65
6.	Definition (Reason)	69
7.	Definition (Adaptation Advice)	70
8.	Definition (Aggregated Feedback)	74
9.	Definition (Relationships capability and generic property)	83

List of Tables

3.1.	Scales of measurement used to represent different properties about runtime quality	42
3.2.	Example of two expectations defining the preferences of a subscriber	51
3.3.	Values for sets S_1 and S_2 of notifications for the fidelity shown in Figure 3.17d	57
3.4.	Fidelity of notifications in S_1 and S_2 when using X_3^e as shown in Figure 3.17.	59
3.5.	Fidelity of notifications in S_1, S_2 when using X_4^e as shown in Figure 3.17.	59
3.6.	Atomic, broker-side, complex and interdependent properties	63
3.7.	Example of three capability profiles	65
6.1.	Properties supported by related approaches	149
6.2.	Comparison features related approaches	149
6.3.	Expectation X_5^{pos} and its reduced form X_6^{pos}	151
6.4.	Capability profiles used by suitable and unsuitable publishers.	152
6.5.	Encoded types used for the evaluation	158
6.6.	Baseline: linear regression analysis results for a bare ActiveMQ	174
A.1.	Literature survey QoD	220
A.1.	Literature survey QoD	221
A.2.	Literature survey QoS	222
A.2.	Literature survey QoS	223
A.3.	Literature survey QoI	224
A.3.	Literature survey QoI	225
A.4.	Literature survey VoI	226
A.4.	Literature survey VoI	227
A.5.	Regression analysis result: scaling population and throughput	247
A.6.	Regression analysis result: scaling number of properties	247
A.7.	Regression analysis result: moderate and aggressive dynamics	247
A.8.	Regression analysis result: scaling properties and dynamics	248
A.9.	Regression analysis result: scaling population/throughput for different dynamics .	248
A.10.	Regression analysis result: scaling population/throughput and properties	249
A.11.	Parameters for the ASIA deployment used in the reference scenario.	249

List of Figures

1.1.	Expectations, capabilities and feedback in a nutshell	7
2.1.	Chapter overview	13
2.2.	Minimal set of components of an Event-based System (EBS)	14
2.3.	Interaction models	14
2.4.	Components of a Publish/Subscribe (PS) system	15
2.5.	Distributed network of brokers (B) forming a Message-oriented Middleware (MOM).	16
2.6.	Service consumer and service provider in a Service-oriented Architecture (SOA).	19
2.7.	Resource provisioning models for Cloud computing.	21
2.8.	Key enablers for Cloud computing as described in [174].	22
3.1.	Chapter overview	25
3.2.	Concepts related to runtime quality	27
3.3.	General-purpose vs. domain-specific solutions	29
3.4.	Generic property: definition, relationships, action	30
3.5.	Steps and filters to derive a generic property format	31
3.6.	Mapping quality-related properties to QoD, QoS, QoI, VoI	33
3.7.	Generic property components	43
3.8.	Types of relationships between generic properties in an EBS.	45
3.9.	Preferences of a subscriber map to requirements and utility values.	46
3.10.	Indifference curves representing malleable requirements.	48
3.11.	Requirement about property p_k defined as part of an expectation.	49
3.12.	Utility for a subscriber in case of open and closed intervals.	50
3.13.	Star plots for X_1^e and X_2^e as defined in Table 6.3.	52
3.14.	Lifecycle of an expectation.	53
3.15.	Fidelity metric intention	54
3.16.	Star plots of expectations X_3^e and X_4^e	56
3.17.	Generic properties for S_1 and S_2 using eqs. (3.8) to (3.10).	58
3.18.	Support for generic properties: design time and runtime constraints	60
3.19.	Capability: actual and potential support for a generic property.	62
3.20.	Freshness to illustrate a broker-side and complex generic property.	64
3.21.	Star plots for capability profiles mapped to expectations	67
3.22.	Lifecycle of a capability/ a capability profile	68
3.23.	Feedback: types and recipients	68
3.24.	Individual feedback: states of an expectation	69
3.25.	Relationship adaptation advice, capability, generic property and actions.	70
3.26.	ASIA imprecision trade-off	74
3.27.	Example: sampling rate represented in ASIA	75
3.28.	Example: same metric represented with different imprecisions	76
3.29.	ASIA dashboard example	77
4.1.	Chapter overview	79
4.2.	Runtime negotiation: matching	81

4.3. Runtime negotiation: types of range-matching (maximizing)	82
4.4. Runtime negotiation: types of range-matching (minimizing)	83
4.5. Runtime negotiation: range-matching algorithm	84
4.6. Runtime negotiation: capability vs. requirement	85
4.7. Runtime negotiation: decision strategy	89
4.8. Runtime negotiation: safeguarding	91
4.9. Runtime adaptation: coordinated adaptation problem	95
4.10. Runtime adaptation: coordinated adaptation solution 1	96
4.11. Runtime adaptation: coordinated adaptation solution 2	97
4.12. Runtime monitoring using watchdogs and monitorlets.	98
4.13. Example for watchdogs detecting satisfied or unsatisfied conditions.	99
4.14. Distributed decentralized EBS	100
4.15. Information flow in a Distributed Event-based System (DEBS)	101
4.16. ASIA example DEBS	106
4.17. Conflicting definitions used by subscribers	107
4.18. Conflicting definitions used by publishers	108
4.19. Runtime support summary	109
5.1. Chapter overview	111
5.2. Overview architecture	112
5.3. Legend of symbols	112
5.4. Design the ExpectationController extension	113
5.5. Design of the ResourceMonitor	114
5.6. Design of the Registry	115
5.7. Design of the Balancer	115
5.8. Design of the ReactionCoordinator	116
5.9. Design of the RateController	117
5.10. Design of the ASIAController	118
5.11. Design of the ExpectationHandlerClient	119
5.12. Design of the ReactionManager	120
5.13. Prototype of a graphical editor for expectations.	121
5.14. Prototype of a graphical editor for capability profiles.	122
5.15. Classes for core components	124
5.16. Class for generic value entity	125
5.17. Plugin-support on Apache ActiveMQ.	126
5.18. ActiveMQ: forwarding in broker networks	126
5.19. ActiveMQ: ExpectationController plugin	127
5.20. ActiveMQ: ASIAController plugin	128
5.21. REDS general architecture: routing and overlay layers.	131
5.22. REDS: differences edge brokers and inner brokers	131
5.23. REDS: joinpoints used	132
5.24. McCAT approach	135
5.25. McCAT: interceptors	136
5.26. Jms2009-PS architecture and extensions	140
5.27. FINCoS architecture and extensions	141
6.1. Chapter overview	143
6.2. Heterogeneity scenario	152

6.3.	Star plots for capability profiles	154
6.4.	Measured results WOE	155
6.5.	Requirements in baseline workload	156
6.6.	Cumulative fidelity compared	159
6.7.	Legend heat maps	160
6.8.	Comparing scenarios regarding conformance with requirements	161
6.9.	Comparing scenarios regarding conformance with requirements	162
6.10.	Data in different scenarios	163
6.11.	Surplus publishers: cumulative fidelity compared	164
6.12.	Subscribers: savings and overhead without encoded types	165
6.13.	Resource savings vs. fidelity.	166
6.14.	Subscribers: savings and overhead with encoded types	166
6.15.	MOM: savings and overhead regarding resource utilization	167
6.16.	Publishers: savings and overhead regarding resource utilization	168
6.17.	Execution costs: exploring the system limits	172
6.18.	Trade-offs between cost drivers for execution costs	175
6.19.	Benchmark results: impact of population size	177
6.20.	Benchmark results: impact of throughput	178
6.21.	Benchmark results: impact on CPU varying when one parameter	179
6.22.	Benchmark results: impact on memory when varying one parameter	180
6.23.	Benchmark results: impact on traffic when varying one parameter	181
6.24.	Benchmark results: impact on latency when varying one parameter	182
6.25.	Benchmark results: impact on CPU when varying two parameters	183
6.26.	Benchmark results: impact on memory when varying two parameters	184
6.27.	Benchmark results: impact on traffic when varying two parameters	185
6.28.	Benchmark results: impact on latency when varying two parameters	186
6.29.	Benchmark results: impact on KPIs when varying all parameters	187
6.30.	ASIA: precision results	191
6.31.	ASIA: network traffic	195
6.32.	ASIA: throughput and latency	196
7.1.	Chapter overview	199
9.1.	Challenges to be addressed in future work	209
9.2.	Steps of the WS-Agreement protocol (bold) using expectations (<i>italic</i>).	212
A.1.	Broker interfaces for controllers.	231
A.2.	Action interface	231
A.3.	Callback interfaces for subscribers and publishers.	232
A.4.	ExpectationHandlerClient and CapabilityHandlerClient interfaces	232
A.5.	ClientHandler for publishers	233
A.6.	ActiveMQ: API provided to plugins	233
A.7.	REDS: joinpoints provided by broker class	234
A.8.	Example generic property	236
A.9.	Test harness to automate executing and analyzing series of single test runs.	238
A.10.	Test environment FINCoS + ActiveMQ	239
A.11.	Anatomy of a single FINCoS run	241
A.12.	Conformance with requirements for surplus publishers.	242
A.13.	Data for scenario with surplus publishers.	243

A.14.Test environment jms2009-PS + ActiveMQ	245
A.15.Anatomy of a single jms2009-PS run	246

List of Abbreviations

Aml	Ambient Intelligence
AMQP	Advanced Message Queuing Protocol
AOP	Aspect-oriented Programming
API	Application Programming Interface
ASIA	Application-specific Integrated Aggregation
BC	Best-Case workload
BNF	Backus-Naur Form
BPEL	Business Process Execution Language
BPM	Business Process Management
CEP	Complex Event Processing
CPS	Cyber-physical System
CPU	Central Processing Unit
CUSP	Channel-based Unidirectional Stream Protocol
DDS	Data Distribution Service
DEBS	Distributed Event-based System
DQML	Distributed QoS Modeling Language
DSMS	Data Stream Management System
EBS	Event-based System
ECA	Event Condition Action rules
EDA	Event-driven Architecture
EFO	Expectations, Filtering Only
ERP	Enterprise Resource Planning
ESA	Expectations + Self-Adaptation
ET	Encoded Types
FCFS	First-come, First-served
FIFO	First In – First Out
FIT	Function, Integration, Traffic
GPS	Global Positioning System
HCC	Human-centered Computing

HCI	Human Computer Interaction
IaaS	Infrastructure-as-a-Service
IoT	Internet of Things
ITSM	IT Service Management
JMS	Java Message Service
JVM	Java Virtual Machine
KPI	Key Performance Indicator
LOI	Locations of Interest
McCAT	Multi-cloud Cost-Aware Transport
MCDM	Multi-Criteria Decision Making
MOM	Message-oriented Middleware
MOOP	Multi-Objective Optimization Problem
MQTT	MQ Telemetry Transport
MSH	Managed Service Hosting
NIIRS	National Image Interpretability Rating Scales
NTP	Network Time Protocol
OASIS	Organization for the Advancement of Structured Information Standards
OGF	Open Grid Forum
OMG	Object Management Group
OODA	Observe-Orient-Decide-Act
OWL	Web Ontology Language
PaaS	Platform-as-a-Service
PoD	Power of Demand
PoS	Power of Supply
PP	Percentage Point
PS	Publish/Subscribe
QoC	Quality of Context
QoD	Quality of Device
QoE	Quality of Experience
QoI	Quality of Information
QoS	Quality of Service
RDF	Resource Description Framework
REDS	REconfigurable Dispatching System

RFID	Radio-Frequency IDentification
RIA	Rich Internet Applications
RMI	Remote Method Invocation
SaaS	Software-as-a-Service
SCM	Supply Chain Management
SLA	Service Level Agreement
SLO	Service Level Objective
SOA	Service-oriented Architecture
SoS	System-of-Systems
SPEC	Standard Performance Evaluation Corporation
SSN	Semantic Sensor Network
STOMP	Simple Text Oriented Messaging Protocol
VoI	Value of Information
VM	Virtual Machine
WOE	Without Expectations
WSDL	Web Services Description Language
WSN	Wireless Sensor Network
XML	Extensible Markup Language

Abstract

Modern reactive software systems turn fine-granular real-time notifications about processes in the physical world into information and knowledge to react in time. Push-based Event-based Systems (EBSs) complement pull-based architectures, such as Service-oriented Architectures (SOAs), and enable enterprises to react to meaningful events in a timely manner. Applications for algorithmic trading, energy-aware reactive data center management, or smart supply chain management are just three examples of reactive systems where information provided by heterogeneous data sources has to be interpreted and where false alarms, missed events or otherwise data of inadequate Quality of Information (QoI) carries a cost.

Whether the QoI of notifications is adequate depends on the purpose, the information is intended to be used for by each receiver. This purpose is application-specific and changes at runtime. Thus, the notion of QoI combines objectively measurable *properties* of a notification and their application-specific *assessment* that determines the Value of Information (VoI) for a receiver.

Receiving only data that conforms to their QoI requirements is crucial for reactive applications. Current support for QoI, however, is limited in terms of expressiveness and effectiveness.

In this dissertation, we introduce the *concept of expectations, capabilities and feedback* as a holistic concept to express, negotiate and enforce QoI requirements at runtime in push-based systems. Participants express requirements and define individual trade-offs between them as expectations; the ability of the system to support properties by adapting itself is captured by capabilities that include the individual costs of participants. Feedback to participants is a central component of our approach and is used to coordinate the adaptation of participants at runtime. We show that our approach is more expressive and supports a wider range of properties than current approaches; our approach actively enforces complex requirements about QoI in an effective way without deteriorating the system's performance.

The work presented in this dissertation contributes to the challenge of runtime QoI support in push-based architectures on a *conceptual* and *practical* level.

On the conceptual level, we contribute a *generic and extensible model* to express and manage requirements about arbitrary QoI properties, *algorithms for negotiation and enforcing* these requirements at runtime as well as a *concept for effective runtime monitoring* in a distributed and decentralized EBS. The conceptual part of this dissertation synthesizes and expands approaches devised in pull- and push-based systems as well as in economics into a novel concept to support QoI at runtime in reactive software systems.

On the practical level, we contribute a *reference architecture* for runtime support of QoI requirements, *two prototypes* built on a centralized and a decentralized Message-oriented Middleware (MOM), *examples for existing applications enhanced with our approach* as well as an *extensive evaluation* of our prototypes built on the industry-strength Apache ActiveMQ platform and the academic REconfigurable Dispatching System (REDS). The evaluation uses industry-strength benchmarks and systems to quantify the benefits and execution costs for participants. The practical part of this dissertation shows the practicability of our approach and quantifies the benefits of actively enforcing QoI requirements using feedback.

Zusammenfassung

Reaktive Softwaresysteme verdichten und interpretieren feingranulare Echtzeitinformationen über Realweltprozesse zu Informationen, die zur Analyse und automatisierten Reaktion genutzt werden. Push-basierte Ereignis-basierte Systeme (EBS) ergänzen sich mit pull-basierten Ansätzen, wie beispielsweise Service-orientierten Architekturen (SOA), und ermöglichen es so Unternehmen, auf relevante Ereignisse zeitnah reagieren zu können. Anwendungen aus den Bereichen Algorithmic Trading, energieeffizientes Rechenzentrumsmanagement oder Smart Supply-Chain-Management sind nur drei Beispiele für reaktive Softwaresysteme, in denen Informationen aus heterogenen Datenquellen in Form von Notifikationen zeitnah interpretiert werden müssen, denn Fehlalarme, nicht detektierte Ereignisse oder sonstige Daten mit ungenügender Informationsqualität (QoI) verursachen massive Kosten.

Ob die Informationsqualität von Daten ausreichend ist, hängt vom jeweiligen Verwendungszweck ab, welcher anwendungsspezifisch und für jeden einzelnen Empfänger verschieden ist; zudem verändert er sich zur Laufzeit. Der Begriff der Informationsqualität umfasst daher sowohl objektiv messbare Eigenschaften einer Notifikation wie auch subjektive Bewertungen derselben, um den Informationswert (VoI) für einen Empfänger zu quantifizieren. Für reaktive Softwaresysteme ist es essentiell, nur Daten zu empfangen, die ihren Qualitätsanforderungen entsprechen. In derzeitigen Systemen werden solche Anforderungen jedoch nur unzureichend unterstützt.

In dieser Dissertation wird das *Konzept von Expectations, Capabilities und Feedback* vorgestellt. Es handelt sich um ein neuartiges und ganzheitliches Konzept, welches es ermöglicht, Anforderungen hinsichtlich Datenqualität zur Laufzeit formulieren, automatisiert verhandeln und durchsetzen zu können. Teilsysteme und Anwendungen beschreiben ihre Anforderungen und mögliche Trade-offs als Expectations. Die Fähigkeit des Systems, bestimmte Qualitätsanforderungen mittels Selbstadaptation erfüllen zu können, wird mittels Capabilities beschrieben. Diese beinhalten auch die individuellen Kosten für die Bereitstellung von Daten mit bestimmten Eigenschaften. Feedback ist eine zentrale Komponente dieses Ansatzes und wird genutzt, um die Adaption von Teilsystemen zur Laufzeit zu koordinieren. Der hier beschriebene Ansatz hat eine höhere Ausdrucksfähigkeit und unterstützt ein breiteres Spektrum an Qualitätseigenschaften als bisherige Ansätze. Anforderungen hinsichtlich Informationsqualität werden effektiv erfüllt, ohne die Performanz des EBS nachhaltig zu mindern.

Diese Dissertation leistet konzeptuelle wie praktische Beiträge zur Unterstützung von Anforderungen über Informationsqualität in reaktiven Softwaresystemen. Auf der konzeptuellen Ebene sind dies: ein generisches und erweiterbares Modell zur Formulierung und Verwaltung von komplexen Qualitätsanforderungen, Algorithmen zur automatisierten Verhandlung und Durchsetzung dieser Anforderungen zur Laufzeit sowie ein Konzept zur effizienten Laufzeitüberwachung von verteilten dezentralen EBS. Der konzeptuelle Teil dieser Dissertation synthetisiert und erweitert Ansätze aus pull- wie push-basierten Systemen sowie den Wirtschaftswissenschaften. Auf der praktischen Ebene sind die Beiträge: eine Referenzarchitektur, Prototypen auf einer zentralisierten sowie einer verteilten Middleware Plattform, Beispiele für die Umsetzung des Konzepts in existierenden Anwendungen sowie eine umfassende Evaluation der Prototypen auf Basis von Apache ActiveMQ und REDS mittels industrie-erprobter Benchmarks und Werkzeuge. Der praktische Teil dieser Dissertation zeigt die Verwendbarkeit des vorgestellten Ansatzes.

Acknowledgements

Without the collaboration, feedback and support of so many people over the last couple of years, I would not have been able to pursue this work in the form presented here.

I want to thank my advisor, Prof. Alejandro Buchmann, Ph.D., for his guidance and support during my time with his Database and Distributed Systems Group (DVS) at TU Darmstadt. Without the freedom he gave me, I would not have been able to pursue my various research activities as I did. I want to thank Prof. Peter Pietzuch, Ph.D. (Imperial College London), for being second referee, hosting me as a Visiting Researcher at Imperial and helping me to shape my ideas.

Special thanks go to John Wilkes, Ph.D. (Google, Inc.), and Kimberly Keeton, Ph.D. (HP Labs), for their feedback and the encouragement to tackle the challenge of information quality; Prof. Wolfram Wiesemann, Ph.D. (Imperial College London), for the discussions on Multi-Objective Optimization; Prof. Dr. Patrick Eugster, Prof. Dr. Johannes Fürnkranz, Prof. Dr. Mira Mezini, and Prof. Dr. Ralf Steinmetz for being members of my dissertation committee.

From 2010 till this very day, I had a great and intensive research collaboration with Prof. Dr. Patrick Eugster (then Purdue University, US), David Evers, Ph.D. (University of Otago, NZ), Alessandro Margara, Ph.D. (University of Lugano, CH), Prof. Dr. Peter Pietzuch, Ph.D., and Emilio Coppa (Sapienza University of Rome, I). Working with them has not only resulted in ASIA and McCAT but also fostered a lot of other ideas described in this dissertation. I am very thankful for all the discussions and in particular to Patrick Eugster for inviting me to the group.

I had the opportunity to collaborate with many industrial partners in various projects. I want to thank Dr. Michael Gesmann, Dirk Mayer, Dr. Harald Schöning, and Ralf Vatter from Software AG, Dr. Christian Webel and Dr. Sebastian Adam from Fraunhofer IESE, Dr. Martin Verlage and Carsten Dänschel from vwd group, as well as Dr. Stefan Roth from SAP; during my time with Deutsche Post MAIL, I had the privilege to work with Jacqueline Burkhard, Helga Schüll, and Dr. Dieter Pütz from Deutsche Post, and Irene Buchmann from Archimetrica. It was a pleasure to work with them all and I am in particular thankful for all the things I have learned from them.

I want to thank my former colleagues from DVS for their collegiality, in particular Dr. Stefan Appel, Max Lehn, Robert Rehner, Dr. Pablo Guerrero, Dr. Christoph Leng, Dr. Kai Sachs, Alex Frömmgen, and Daniel Bausch as well as Dr. Ilia Petrov and Astrid Endres. Special thanks go to our secretary and invisible engine Maria Tiedemann for her support and assistance.

Thanks also to my students Matthias Eichholz, Erman Turan, Pascal Kleber, Karim Abou Sedera, Christoph Schott, Arne Stühlmeyer, Julian Dean, Henriette Röger, Jan Matuschek, Vikrant Lawangare, and Michael Chromik. My thanks also go to the Coffea bar in Darmstadt and its team of dedicated baristas: many ideas presented here flourished over a well-made espresso.

Finally, thanking one's wife and family for carrying all the burdens along the way with you might seem perfunctory and stereotypical to some. I am certain, however, that those who have embarked on such a long and not always sunny journey themselves can imagine, how deeply indebted I feel to my wife Mareike and my family for their continuous support.

In memoriam Hans-Dieter Ebert, Marion Braun, Walter Waterfeld and those of friends and family who had seen the start of this project but not the end of it.



1 Introduction

I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of Science, whatever the matter may be.

Popular lectures and addresses [408]

LORD KELVIN

The gap between the physical world and its state represented and reasoned about in software systems is about to close. This trend is manifesting itself most prominently in the recent notion of the Internet of Things (IoT). The IoT refers to the integration of heterogeneous digital devices, software architectures and Cloud-based infrastructures to provide and process fine-granular data about real-world events without requiring human intervention [12, 39, 192, 247, 273, 341, 413]. This enables modern reactive software systems to turn real-time data about processes in the physical world into information and knowledge to react in time: smart Supply Chain Management (SCM) systems can automatically redirect delivery routes or trigger processes to replenish goods if they detect delays based on incoming notifications [233]; earthquake early-warning systems can initiate counter-measures if they infer indications for imminent earthquakes by combining data from various sensors and backend systems [93]; financial trading applications decide to buy or sell at stock exchanges based on real-time news feeds; fraud detection systems cancel suspicious transactions in financial services if they are confident to witness an attempted fraud by reasoning about data of the ongoing transaction [78, 208]; and data center management systems scale and reassign resources based on detected usage patterns and energy consumption of resources to balance resource usage with cooling costs [81, 106, 250].

On the software architecture level, push-based Event-based Systems (EBSs) complement pull-based Service-oriented Architectures (SOAs) to leverage information about real-world events in software systems [76, 78].

Within the push-based communication model of EBS, software components act with different roles: as *subscribers*, they get notified whenever data sources provide information about events they are interested in; as *publishers*, they publish notifications about meaningful events that they have detected with a certain confidence. Subscribers and publishers are fully decoupled by a *Message-oriented Middleware (MOM)* that *pushes* notifications from publishers to interested subscribers as soon as they are available [147]. Dependencies between participants in an EBS are formed dynamically at runtime, based on the type, quality-related properties (e.g., sampling rate, confidence of detection, precision, trustworthiness) or content of notifications, enabling many-to-many communication [208]. Publishers and subscribers are unaware of each other and anonymous but to the middleware, allowing them to join and leave the system without causing interruptions. Thus, a push-based EBS is a suitable architecture for reactive data-centric systems with an inherent need for flexibility, extensibility and scalability at runtime.

1.1 Problem Statement

Having information of adequate quality available at the right time in the right place is vital for software systems to react to situations or support decisions. The applications in the area of IoT and EBS mentioned above are just a few examples of modern reactive systems where heterogeneous information provided by various publishers has to be interpreted and where false alarms, missed events or otherwise information of inadequate Quality of Information (QoI) carries a cost [35, 50, 55, 125, 182, 250].

In general, information in an EBS can be considered to be of inadequate quality if it is not precise, accurate, or fresh enough; if notifications about events arrive out of order, causelessly (false positives) or not at all (false negatives); if data is not reliable enough because the publisher is not trustworthy, or not confident enough. The degree to which any of these properties have to be met for QoI to be considered adequate, however, depends on the purpose the information is intended to be used for by each subscriber [35, 182]. This *Value of Information* (*VoI*) is application-specific and can change dynamically at runtime as it depends on the individual *utility function* of each subscriber, itself subject to the subscriber's context and state [56, 225, 421, 424].

Supporting requirements about QoI depends on the EBS satisfying individual requirements about quality-related properties at runtime as most of these properties cannot be determined at design time [35, 42, 85, 213]. In that regard, the notion of QoI in EBS encompasses aspects of Quality of Service (QoS) and other concepts addressing runtime quality but is not limited to them.

We use three motivational examples inspired by research cooperations: financial data vendors, energy-efficient data center and smart supply chains. The examples highlight different aspects of QoI and VoI in reactive software systems and the relevance of QoI.

Financial Data Vendors

Trading on today's financial markets is based on software systems for analyzing, planning and executing transactions. Financial data vendors provide banks, traders, and end users with information to base investment decisions upon. The spectrum of the provided information ranges from raw data feeds about trades at stock exchanges delivered at high sampling rates and low latency to aggregated analytics that include background reports about markets or trends.

A financial data vendor is subscribed to fine-granular notifications about events at the stock market, published by the different stock exchanges. This data is directly forwarded to some consumers; for others, it is aggregated and fused with historic data or general news first, using approaches such as Complex Event Processing (CEP) or manual analysis. The sampling rate of incoming notifications from a single stock exchange can range from one notification per day to more than 100.000 notifications per second. All participants of such an EBS are usually located in data center connected by high-speed networks.

Customers of a financial data vendor subscribe to a combination of product (e.g., tick, aggregate, report), content attributes (e.g., stock exchange, ticker symbol) and QoI properties (e.g., latency, sampling rate). At an extreme end of the subscriber spectrum, high-frequency or low-latency trading applications exploit the speed of algorithmic decision making in software systems and minute information asymmetries for arbitrage revenues. They have restrictive requirements about latency and sampling rate but are willing to pay a premium; some applications require a

minimum sampling rate while others need to define a maximum sampling rate as they are unable to process information properly if the sampling rate is too high. End users managing their own portfolio manually, on the other hand, are usually more interested in aggregated updates on the development of a stock symbol or trend forecasts, rendering latency and sampling rate insignificant compared to the requirements of algorithmic traders [73].

For financial data vendors and their subscribers, QoI directly translates into products, prices and penalties. The VoI of each subscriber determines the products it consumes and holds the financial data vendor responsible for. Providing data with insufficient QoI results in penalties and revenue loss for the financial data vendor. For consumers, receiving data with insufficient QoI results in suboptimal strategies to buy or sell – in the worst case even leading to stock market crashes such as the May 6, 2010 Flash Crash [255].

Energy-aware Reactive Data Center Management

Virtualized resources in data centers accessible via broadband networks and the Internet provide scalable infrastructures for applications, CEP engines and MOMs at different levels of abstraction. In this dissertation, they are subsumed under the term Cloud computing and allow applications to adjust resources automatically to meet fluctuations in demand. Resources are rented out to tenants by providers on a pay-per-use basis. The physical servers managed by the provider are not accessible to tenants but hosted Virtual Machines (VMs) and applications are. Network traffic in and out of the data centers is charged for by the provider.

From the perspective of a Cloud provider, this pay-per-use business model requires fine-granular monitoring of resources and applications for billing and availability management as violations of Service Level Agreements (SLAs) result in penalties and reduce revenue. Energy consumption of servers and cooling facilities are the main cost drivers in data center operations [44, 49, 257, 313]. Thus, providers try to optimize the utilization of their resources by balancing the level of utilization of each server with the energy it consumes, the heat that it produces, and the costs necessary for cooling. For this, sensory data about energy consumption and ambient temperature is incorporated into load-balancing algorithms together with metrics about applications, their SLAs, server utilization and network traffic. Resource management in this setting is done in a push-based fashion: certain events trigger a reassignment of resources for a given application, e.g., scale-in or scale-out, runtime migration to other hosts etc. Triggering events can be: changes to the workload or SLAs of the hosted application, critical resource utilization caused by other tenants hosted on the same resource, or outages [48, 81, 106, 128, 257, 280, 313].

At runtime, the whole technology stack has to be monitored: network traffic, racks, single servers, VMs hosted on each server, applications such as Apache Hadoop running on each VM, or single jobs executed by an application [250, 315]. Thus, runtime monitoring requires multiple publishers to provide runtime information about different entities. For example, monitoring systems like Ganglia or Nagios report on the state of a VM, application-specific agents like Hadoop Task-Trackers [16] or Borglets [414] monitor job execution, and components of the hosted application provide application-specific metrics. The data provided by these publishers is sometimes redundant in its content but differs in its QoI properties such as sampling rate, granularity, precision, or latency. The same data is consumed by many different subscribers such as applications for billing and metering, data warehouses, resource managers such as BorgManagers or Hadoop JobTrackers, dashboards, the applications themselves, load balancers, or cooling systems.

Requirements of subscribers regarding different quality-related properties of notifications are individual and can change dynamically over time. Some examples: monitoring data about a VM delivered at a given sampling rate and confidence of detection might be sufficient for the purpose of one subscriber while another subscriber might need the same type of data at a higher sampling rate but would tolerate less confidence of detection or precision – a third subscriber might not care about precision at all but requires measurements about the same entity from three different publishers for cross-validation; monitoring data in its current form might be sufficient for a subscriber as long as there is no indication of malfunction at the monitored entity – in case of anomalies, the same data is required at a high sampling rate for root cause analysis by this subscriber while other subscribers still require a lower sampling rate as they are resource-restricted.

In the context of reactive data center management systems, providing, processing or consuming data with insufficient QoI has a severe impact. Resource managers and load-balancers in data centers rely on precise data about the ambient temperature and energy consumption of server racks. They are bound to misjudge the actual utilization and power consumption of resources if the data they receive is imprecise or precise data is drowned out by too much imprecise data. Consequently, resources might be overloaded and overheat, resulting in outages and violated SLAs for jobs running on these resources. Alternatively, underutilization of resources or overprovisioning of cooling facilities results in skyrocketing costs [81, 106, 313].

Smart Supply Chain Management and Industry 4.0

Advances along the whole technology stack have accelerated the emergence of the term *Industry 4.0* [203, 282]. The term denotes the vision of tightly integrated production and delivery processes that rely on machine-to-machine communication and the IoT to monitor, execute and optimize the manufacturing value chain. On the device level, increasing miniaturization and decreasing production costs enable a myriad of sensors to be used in monitoring real-world conditions while actuators can manipulate objects and processes in the real-world. Sensors and actuators are omnipresent in modern supply chains and production processes to feed EBSs and SOAs. For example, as part of Wireless Sensor Networks (WSNs), or as active and passive Radio-Frequency IDentification (RFID) tags in Cyber-physical Systems (CPSs) [76, 78, 176, 177, 208].

The resulting applications are distributed and form federated systems with a high degree of heterogeneity and dynamics; often they are a mix of energy- or otherwise resource-constrained participants and Cloud-based backend systems with no such constraints. As for smart buildings, multiple publishers for the same type of information are bound to become available over time as new devices are added that provide a bundle of capabilities previously provided by dedicated devices. Subscribers in those systems can range from Cloud-based Enterprise Resource Planning (ERP) applications and data warehouses to resource-constrained mobile devices such as smart glasses or handhelds. The data provided, processed and required is also very heterogeneous in terms of type, granularity and quality-related properties. For example, position information is provided or required about single items, or about the container or even the truck the items are contained in; some sensors provide precise information about the position or status of an entity while others have a certain drift; others again cannot provide data as frequently as required due to energy-constraints.

In terms of QoI, applications in the domain of smart business processes in manufacturing and logistics usually do not require subsecond latencies. Rather, they require complete, precise and

trustworthy information about the state of a process. Incomplete or imprecise data can lead to miscalculations of supplies, lot sizes or due dates. At the same time, energy-constrained devices have to avoid draining their batteries and rendering them useless. Thus, energy-constrained publishers have to be aware of interested subscribers and their required sampling rates [62, 99, 261, 325, 415, 418].

Limitations of a Typical EBS Regarding QoI

The three examples from different domains of modern reactive systems illustrate the relevance of QoI for applications in an EBS. In a typical EBS, however, runtime support for QoI is insufficient as requirements about QoI can only be supported implicitly by encoded types, or by additional metadata in notifications; some MOMs offer explicit support only for domain-specific and fixed sets of properties. Both approaches have limitations on the conceptual and technological level.

Implicit support for requirements about QoI-related properties can be provided by publishers in an EBS by advertising types that encode quality-related properties in their name (e.g., `Cpu-Usage_rate50_confidence70`), or by adding metadata to the content of each published notification (e.g., `rate=50, confidence=70`). Subscribers can express their requirements by subscribing to the encoded type that fits their requirements best, assuming that the semantics are known. This approach, however, is limited in terms of expressiveness and efficiency. First, using encoded types restricts the set of available properties to those determinable by publishers at design-time, excluding important runtime properties like latency and reliability that are determinable only by the MOM before dispatching notifications to subscribers. Crucially, publishers cannot coordinate their supply with the demand of subscribers, as there is no feedback from subscribers to publishers in a typical EBS. Above all, interdependent properties that require the participation of multiple publishers cannot be supported in a typical EBS using encoded types, as there is no coordination between publishers. For example, a typical EBS cannot support the requirement of a subscriber about a number of *alternatives*, i.e., the same type of notification has to be provided by a certain number of different publishers. Second, using encoded types for different combinations of quality-properties would result in an unmanageable growth of available types and traffic overhead as the same information has to be processed for multiple encoded types [23, 85, 171]. Such overhead, however, is not suitable for environments where processing data is expensive.

Explicit support for a few quality-related properties is provided by MOMs like IndiQoS [85], Adamant [213] with the underlying Data Distribution Service (DDS) [197, 269, 334, 345], or Harmony [253, 428]. These solutions are also limited on the conceptual as well as on the technological level. On the conceptual level, they focus on a fixed set of MOM-related QoS properties at a low level of abstraction, which they try to satisfy by adapting the MOM on the transport protocol level only. They do not consider requirements about QoI properties that would require publishers to adapt at runtime. On the technological level, they have specific requirements about the infrastructure and require tight vertical integration across the technology stack to switch between custom transport protocols. The applicability of these approaches in heterogeneous IoT deployments that involve Cloud-based services, however, is limited. For example, direct access to specific hardware features on the host machines for performance tuning is no longer provided and transport mechanisms like multicast are not available in Cloud environments [173].

1.2 Research Questions

Based on the discussed challenges regarding runtime support of QoI requirements in push-based systems, we derive four research questions to be addressed in this dissertation:

- Q1 What constitutes Quality of Information (QoI) and Value of Information (VoI)?
 - Q2 What are suitable abstractions to express individual requirements and capabilities about QoI properties in EBS?
 - Q3 What are strategies to enforce requirements and resolve conflicts?
 - Q4 What is the influence of resource- or cost-constrained environments?
-

1.3 Proposed Solution, Scope and Contributions

In this dissertation, we introduce the concept of **expectations, capabilities and feedback** as a holistic approach to support QoI in EBS as a first-class citizen and enable participants to adapt at runtime based on feedback about their actions and the global system state. We consider the fact that participants operate on resource-constrained devices and Cloud environments. Hence, resource- and data-efficiency are important criteria in our approach.

We enable subscribers to express preferences for QoI properties based on their individual VoI while publishers can associate their capabilities with costs for providing certain QoI properties. As part of our approach, we model QoI-related properties like sampling rate, confidence of detection, or latency in a generic way. Actions are associated with a generic property and define how it can be adjusted at runtime: by advising a publisher to adapt, adapting the MOM, or by adapting both publisher and MOM in a coordinated fashion.

Subscribers can express preferences and individual trade-offs between requirements as *expectations* in a consistent and information-centric way. Publishers expose their general abilities to support a generic property, their costs, and the state they are currently operating at to the MOM as *capabilities*. In a capability, we do not only capture the current state of the publisher but, more importantly, we model the extent to which it could adapt. Thus, a capability describes first and foremost the spectrum of adaptation that would be possible together with the costs arising from this adaptation; the current state is included as well to analyze the still exploitable spectrum.

Malleable requirements (expectations) and the ability of the system to adapt (capabilities), defined as weighed ranges over generic properties, enable us to automatically negotiate QoI requirements at runtime. The algorithms used in this process identify the extent to which the system would have to adapt to satisfy requirements. The system assesses the feasibility by balancing the costs for performing the identified adaptation with the benefits to be derived from it using custom decision- and load-balancing strategies; costs and benefits depend on the current system state, the preference expressed by subscribers and the costs associated with publishers. Based on this assessment, requirements are declined or satisfied by adapting the system. Subscribers receive feedback about the state of their expectations while publishers get feedback about the usage of their capabilities and explicit advice to adapt if necessary.

The proposed concept complementary extends the push-based paradigm of an EBS without compromising the model of indirect many-to-many communication, making it backward compatible. As shown in Figure 1.1, expectations and capabilities are defined independently of advertisements, notifications or subscriptions. They are matched only at the MOM, preserving the

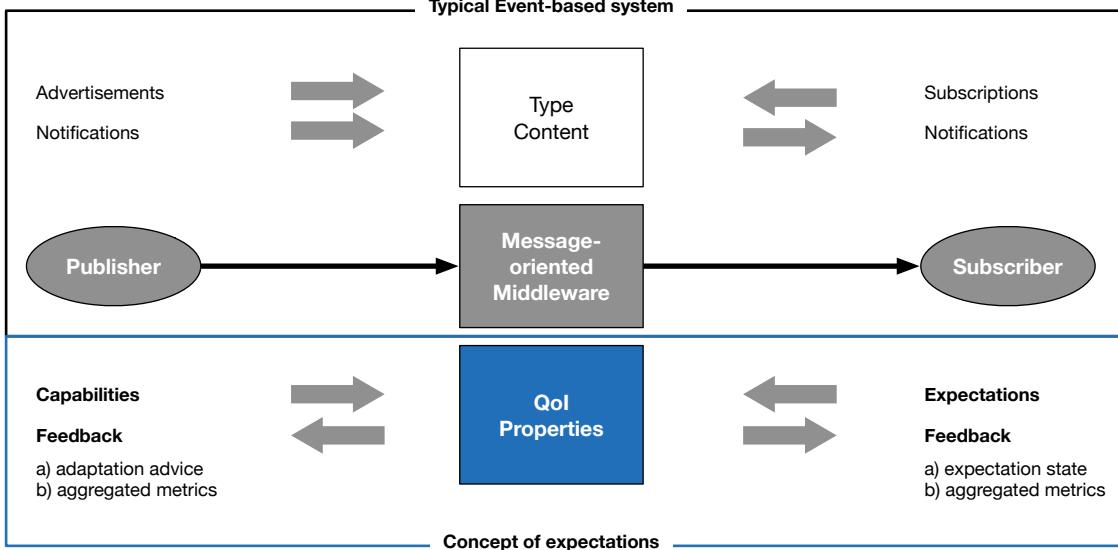


Figure 1.1.: Our concept extends the model of EBS (top) with capabilities, expectations and bidirectional feedback for runtime adaptation (bottom, bold).

anonymity of the associated participants in an EBS that is essential for scalability, flexibility, extensibility and robustness. Individual feedback about expectations or capabilities as well as aggregated feedback about the system enables participants to assess their current situation and adapt their behavior at runtime.

Expressing capabilities in a uniform way and associating generic actions with them, we can connect to the huge body of existing work to manipulate and optimize dedicated lower-level properties such as latency, throughput, or precision. We can utilize these related approaches by decomposing requirements about more abstract generic properties expressed in our approach into lower-level requirements. For example, latency is a property supported by both IndiQoS and Harmony on the transport protocol level. In our approach, we could enforce requirements about this property by associating it with a generic action that maps our definition of latency to the representation used by IndiQoS or Harmony. Depending on the deployment of the system, we could then trigger an underlying instance of either MOM for enforcement.

The fact that QoI requirements or capabilities of a participant can be influenced by changes to its context or state is essential to our approach, as it requires the EBS to react dynamically at runtime. For example, a mobile publisher's publication latency can temporarily increase due to bad connectivity, forcing the MOM to evaluate the impact of this change on subscribers that rely on low-latency data from this publisher. However, we do not necessarily have to know and understand the current context or state of a participant for this. Neither do we require all participants to be aware of their own context or state in the sense that they are able to infer and interpret their current context based on a specific context model.

A key element of our approach is the realization that QoI requirements and capabilities can change dynamically at runtime. These changes have to be detected by the MOM using runtime monitoring or by participants notifying the MOM about updates of their requirements or capabilities. In neither case do we need to know the *reason* for a change and thus do not require the use of a specific context model. Instead, we provide subscribers with an expressive model that allows them to map their preferences to requirements about objective properties supported by

the system without having to expose the model they use to infer and determine their own context and state. Respectively, publishers can express their current and potential supply for properties depending on their state and context without having to expose their context model either. Thus, approaches to infer and determine the ambient context of a participant are considered to be secondary. Security and privacy aspects are important but orthogonal to our approach; they are out of scope of this dissertation.

By supporting QoI in an EBS in a holistic way, we contribute to the vision of reactive software systems that consist of *coequal* pull- and push-based software architectures [18, 19, 76, 78]. Pull-based software architectures like SOA operate on persistent data and offer extensive runtime support for SLAs defined over QoS-related properties; efficient monitoring approaches provide usage information about SOA-based systems and SLAs defined for them at runtime. Push-based software architectures like EBS, however, lack a comparably generic concept to support QoI-related properties at runtime. Providing the necessary usage information about an EBS and QoI properties at runtime is still an open issue. We contribute concepts to support and monitor guarantees about QoI-related properties in EBS at runtime.

Referring to the research questions regarding the support of QoI in EBS, the contributions of this dissertation are in particular:

Generic model to express QoI requirements and adaptation capabilities (Q1+Q2). We propose a generic model to express and manage malleable requirements about QoI-related properties in an EBS at runtime. Subscribers define requirements about an arbitrary set of properties as an *expectation* in a consistent and information-centric way. Multiple expectations by the same subscriber define a preference order without the need to define an explicit utility function. Publishers expose their support for QoI-related properties to the MOM as *capabilities*. Each capability describes the spectrum of support a publisher can offer for a specific property when using self-adaptation; the current state it is operating at is contained as well. A *capability profile* of each publisher is maintained at the MOM and includes capabilities for those properties determinable only by the MOM at runtime. Costs and benefits are essential components of our model: subscribers quantify the VoI of their requirements by ranking them with *utility values* while publishers and MOM manage *cost functions* for capabilities, e.g., the costs for maintaining a sampling rate of 100 notifications/second vs. a sampling rate of 50. The degree to which a subscriber is satisfied with the results delivered by the MOM is quantified by its *fidelity*.

Algorithms to negotiate and enforce QoI requirements at runtime (Q3+Q4). We present algorithms to automatically decide about QoI requirements at runtime, deduce the action plan and execute it. In particular, we discuss algorithms for matching expectations to capabilities, deciding on satisfiable expectations, safeguarding action plans and enforcing the decision. Our approach allows using custom decision strategies and takes into account the costs and utility values provided by participants. Decisions are enforced by transparently adapting the MOM and advising publishers to adapt.

Feedback for self-adaptation to extend the communication model of EBSs (Q1,Q2,Q4). We extend the unidirectional communication model of push-based systems with bidirectional feedback from the MOM to participants. At runtime, we provide subscribers, publishers and brokers with individual and aggregated feedback. Individual feedback contains status updates from the MOM about expectations or capabilities of a dedicated participant. Subscribers get updated about the state of their active expectations while publishers receive explicit *adaptation advices* containing a target value that a capability has to be adjusted to. Thus, we extend the scope of

supported QoI properties to those influenced by publishers. Aggregated feedback is delivered to all participants that have registered interest in aggregated metrics about the system population and state, e.g., the number of active subscribers for a certain type of notification or the average sampling rate of notifications of a certain type.

Novel concept for runtime monitoring of decentralized event-based systems (Q1+Q4). We introduce a novel approach to monitor the population and dynamics of a large-scale Distributed Event-based System (DEBS) effectively based on the concept of Application-specific Integrated Aggregation (ASIA). We use this monitoring information in three ways to support QoI requirements: a) to detect significant changes in the global state of the system that require a renegotiation of expectations; b) to decide on load-balancing; and c) to provide aggregated feedback to participants. Participants can individually balance the measurement costs with the freshness and precision of the reported monitoring information. Our approach exploits relaxations on information precision to limit the propagation of unnecessary updates within the system.

Reference architecture for runtime support of QoI requirements (Q3+Q4). We present the design of an architecture to support QoI requirements in an EBS at runtime. Our reference architecture consists of an extension to each broker in the MOM as well as additional handlers provided to participants. The extension makes a broker self-adaptive while the handlers enable participants to deal with feedback by the MOM and manage the lifecycle of expectations and capabilities. Our design can be applied to any push-based MOM as it extends the Application Programming Interface (API) of an EBS. The MOM extension consists of platform-independent components for negotiating between expectations and capabilities, monitoring the system state, and coordinating adaptation. Platform-specific components implement MOM-related reactions like routing adaptation or traffic shaping.

Prototypes implemented in Java as proof of concept (Q2-Q4). We present two prototypes of our reference architecture built on open-source MOMs with different features to show the feasibility of our approach: Apache ActiveMQ is representative of a centralized industrial-strength MOM focusing on high performance while the distributed REconfigurable Dispatching System (REDS) allows us to exploit routing strategies in decentralized networks of brokers. Both MOMs have been extended to support expectations, capabilities and provide feedback to participants. Extensions are written in Java without affecting existing code.

Examples for applications implementing our approach (Q1+Q3,Q4). We present examples for existing open-source applications that we have enhanced with our approach. Using expectations, capabilities and feedback, QoI requirements are now supported in the Ganglia monitoring system and MySQL master-slave replication by extensions to our self-adaptive Multi-cloud Cost-Aware Transport (McCAT) MOM. We have designed McCAT to combine the self-awareness in distributed and decentralized EBS, as provided by ASIA, with awareness about QoI that is provided by expectations and capabilities. Using McCAT as a transparent transport mechanism for existing distributed applications, we can save up most of the network traffic produced by out-of-the box deployments of Ganglia monitoring and MySQL master-slave replication without violating QoI-requirements defined in these systems. Using McCAT wrappers, we make static distributed systems self-adaptive without having to change existing code. For the evaluation of our approach, we have also extended the open-source benchmarking tools FINCoS and jms2009-PS to include expectations, capabilities and feedback in their workloads.

Evaluation regarding expressivity, benefits, and execution costs (Q3+Q4). We evaluate the expressivity of our approach by showing that we support arbitrary QoI properties while we also

support the limited sets of QoI properties addressed by related approaches. We show that using expectations, capabilities and feedback for adaptation results in significant benefits for subscribers, MOM and publishers in terms of higher fidelity, less network traffic and lower CPU utilization. In terms of execution costs we show that processing and negotiating expectations and capabilities at a single broker is feasible for more than 2000 active participants triggering renegotiations up to every five seconds without degrading the performance of the MOM in terms of throughput and latency. We identify the cost drivers for execution costs and explore their trade-offs using jms2009-PS, which is based on the industry-standard benchmark SPECjms2007. We evaluate the effectiveness of our proposed monitoring approach separately in terms of performance, scalability and precision. We show that our approach provides information about the population and dynamics of a DEBS at the individual level of precision defined by the consuming components. Our monitoring approach outperforms external aggregation systems by collecting and distributing information with a limited overhead while we do not significantly impact the performance of the MOM in terms of throughput and latency.

1.4 Related Activities and Publications

Research contributing to the work presented in this dissertation has been conducted in the context of several multi-institutional research projects and collaborations. The results have been published in a number of peer-reviewed publications and students' theses.

Research Projects and Collaborations

I have been involved in several joint research projects with industrial and academic partners to investigate deployment and operation aspects of enterprise software systems following the SOA or EBS paradigm.

With Deutsche Post MAIL, I have improved the maintainability of SOA-based application landscapes by developing a metric for IT Service Management (ITSM) to quantify the criticality of a SOA based on *Function, Integration, Traffic (FIT)*. This work has been applied to participants in an EBS as part of a cooperation with Software AG in the context of the BMBF Software Campus project WEFITM¹. The lessons learned have been incorporated into the algorithms for negotiating and enforcing QoI requirements at the MOM as presented here.

Within the BMBF Software-Cluster research projects Software-Cluster EMERGENT² and SINNODIUM³, I have worked closely with research partners from industry and academia on enabling emergent behavior in enterprise software systems by integrating the paradigms of EBS and SOA on the architectural level. Both research projects have focused on improving interoperability, reliability and adaptability of enterprise software provided by different vendors. EMERGENT focused on software engineering and design time aspects of software systems spanning across multiple organizations. Work in SINNODIUM has been centered on runtime aspects of multi-organizational systems, such as monitoring business processes and heterogeneous infrastructures, as well as aiding reactive behavior to respond to relevant business events. The work done in these projects has greatly influenced the concepts presented here. Valuable feedback and input from industrial research partners — major software vendors and corporate users — on runtime aspects of push-based enterprise systems helped to verify the concept presented here.

¹ <http://www.softwarecampus.de/en/home/>

² <http://www.software-cluster.com/en/research/projects/joint-projects/emergent>

³ <http://www.software-cluster.com/en/research/projects/joint-projects/sinnodium>

Working on aspects of self-awareness and self-adaptiveness in push-based systems, I have established a close research collaboration with Peter Pietzuch (Imperial College London, UK), Patrick Eugster (then Purdue University, US), David Evers (University of Otago, NZ), Alessandro Margara (University of Lugano, CH), and Emilio Coppa (Sapienza University di Roma, I). We have developed the concept of Application-specific Integrated Aggregation (ASIA) as a means to efficiently monitor the population and dynamics of large-scale decentralized DEBS. During my stay at Imperial College London as Visiting Researcher, the scope of that work was extended to encompass self-adaptive behavior applicable for Cloud environments, resulting in the self-adaptive and cost-aware system McCAT. The work on ASIA and McCAT is directly contributing to this dissertation.

Related Publications

I have authored and co-authored several peer-reviewed publications that cover work contributing directly and indirectly to this dissertation. As part of preparatory work we analyze different aspects of the concepts contributing to the IoT; on the device layer, we characterize RFID sensors and software systems to leverage their benefits for SCM systems in [176] while we evaluate the technological and economic enablers of Cloud computing in [174] from the perspective of data intensive applications on the infrastructure layer.

On the software architecture level, several publications discuss the benefits and challenges of integrating SOA and EBS to create reactive software systems and derive requirements for runtime quality management in EBS [76, 78, 170]. The FIT-metric to optimize the availability of SOA-based application landscapes is described in [169]. Focusing on open challenges in developing and maintaining event-driven and service-oriented architectures, I have been involved in contributing a container concept for encapsulating event-driven functionality [18], integrating this concept into approaches for Business Process Management (BPM) [19, 20] and dealing with heterogeneous data in an EBS by applying a transformation approach [166, 168]: *Eventlets*, the proposed container concept for event-driven functionality, is the main topic of Stefan Appel's dissertation [17] while the transformation approach *Actress* is the key contribution of Tobias Freudenreich's dissertation.

The work on ASIA, as described in this dissertation, has been published in [150, 171, 172, 177] while the self-adaptive system McCAT is outlined in [173]. The concept of expectations, capabilities and feedback is described in an early stage in [175].

Supervised Theses

I have been involved in supervising several students' theses that address aspects relevant for this dissertation. Routing mechanisms in EBSs and DEBSs are analyzed by Matuschek in [302]. The transformation approach for heterogeneous data in a DEBS, mentioned above and described in [166, 167, 168], is based on [165] by Freudenreich. In cooperation with Software AG, complementary work on modeling and checking SLAs for SOA-based systems has been presented by Dean in [132], while Stühlmeyer introduces a concept for describing and updating reactions in CEP engines at runtime [401]. Mechanisms for ensuring requirements about latency in EBSs and DEBSs are investigated by Eichholz in [138] and contribute directly to this dissertation.

1.5 Structure

This dissertation presents the concept of expectations, capabilities, and feedback to support QoI requirements in EBS. The remainder of this document is structured as follows:

Chapter 2 provides background information about push-based and pull-based paradigms. In particular, Publish/Subscribe (PS) and Complex Event Processing (CEP) are covered as key concepts within the paradigm of EBS. Related approaches such as Wireless Sensor Networks (WSNs), Data Stream Management Systems (DSMSs), Cloud computing, Service-oriented Architectures (SOAs), and System-of-Systems (SoS) are discussed in aspects relevant for this work.

Chapter 3 introduces the model of expectations, capabilities and feedback in detail. We revise the relationship between Quality of Information (QoI) and related terms to establish a coherent terminology. We review the spectrum of properties associated with these concepts and reduce their generalizable features to the modular and reusable building blocks of our approach. We present expectations to express malleable requirements about those generic properties and capabilities to describe support for them. We define the different types of feedback introduced in our approach and define the fidelity metric to quantify the conformance between a subscriber's requirements and the data provided by the MOM.

The conceptual foundations for negotiating and enforcing QoI requirements at runtime are presented in Chapter 4. We describe the whole process of runtime monitoring, runtime negotiation, and runtime adaptation in a push-based system. This includes algorithms to monitor the system state in a decentralized setup, adapt the MOM, as well as detect and resolve conflicts.

Chapter 5 describes the architecture for supporting our proposed concept in an EBS and a DEBS. We describe the extended API provided to subscribers and publishers that enables them to express expectations and capabilities and receive feedback. We describe the platform-independent parts of our architecture first before describing two prototypes built on top of ActiveMQ and the REconfigurable Dispatching System (REDS) as proof of concept. We also describe the steps necessary for publishers and subscribers to utilize our approach, using our modifications of McCAT, the jms2009-PS benchmark, and FINCoS for illustration.

We evaluate our proposed concept and the described prototypes in Chapter 6 in regarding expressiveness, benefits for participants and execution costs for the MOM.

In Chapter 7, we discuss related work focusing on runtime quality in the area of EBS, WSN, Cloud computing, and SOA. In particular, we compare our concept to related approaches providing support for quality-related properties in push-based systems and distributed systems in general.

We summarize our contributions in Chapter 8 and point out future research in Chapter 9.

2 Background

In this chapter, we present background information about basic concepts and paradigms referred to in this dissertation. As shown in Figure 2.1, the remainder of this chapter is structured into three parts that focus on push- and pull-based interaction models.

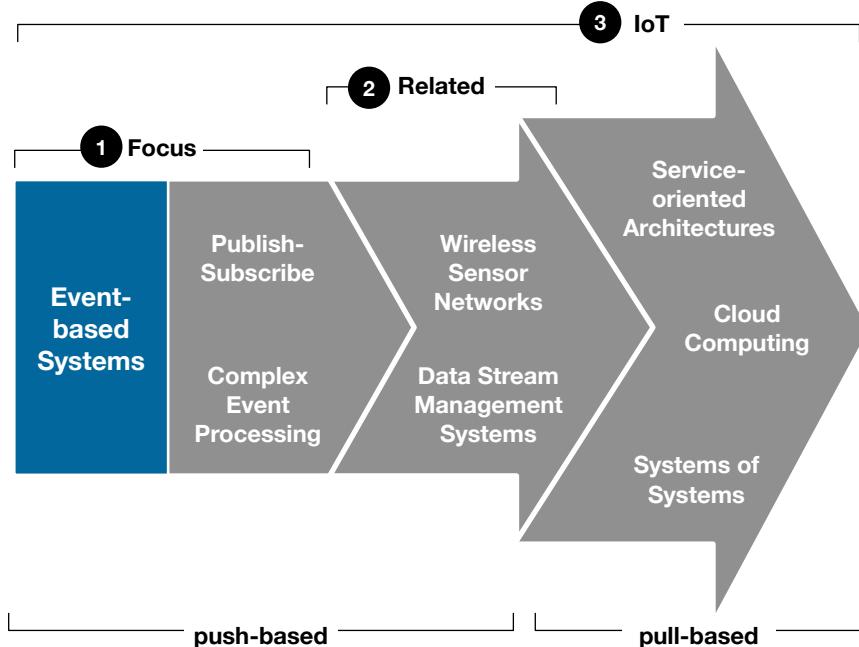


Figure 2.1.: Chapter structure: background on push- and pull-based approaches.

First, we focus on the push-based communication model of an Event-based System (EBS). In particular, we introduce Publish/Subscribe (PS) and Complex Event Processing (CEP) as the two aspects of an EBS relevant for this dissertation. Based on this conceptual foundation, we present an overview of Wireless Sensor Networks (WSNs) and Data Stream Management Systems (DSMSs) as directly related concepts. Finally, we include pull-based concepts also contributing to the notion of the Internet of Things (IoT), namely Service-oriented Architectures (SOAs), Cloud computing, and System-of-Systems (SoS).

2.1 Event-based Systems

An EBS is a reactive *sense-and-respond system* designed around the concept of *events*. An event is defined as a significant change of state in the physical or digital environment of a system. What is of significance is defined by the application using the EBS. In principle, there are change events (e.g., job completed), status events (e.g., position or ambient temperature), and interval events (e.g., process duration). As time is constantly changing, even two status events describing the same state of the environment at different points in time are considered to be different events [94, 208].

Observations about events are represented by *notifications*. A notification consists of at least an identifier, a timestamp and a payload that describes the observed event. The timestamp could

represent a point in time (e.g., the time of detection or the time of publication) in absolute or relative terms, or it can represent an interval of time (e.g., a duration for which the observed event lasted). The payload consists of a set of attributes, objects, or semi-structured data describing the event [318]. With regard to the information density carried by a notification, three kinds of events are usually distinguished [208]: *simple* events are fine-granular observations on a low level of abstraction, such as raw sensor readings or stock ticks, while *composite* events are aggregations of simple and composite events. *Complex* events, in contrast, are derived at a high level of abstraction based on the observation and interpretation of events (e.g., shipment XYZ 80% complete). Event algebras describe the rules for aggregating simple and composite events while enrichment can be used for deriving complex events.

An EBS is able to *detect* events of interest, *notify* affected components and *react* to the observed situation based on certain *rules*. For that, it consists of at least a monitoring component, a transmission mechanisms and a reactive component as shown in Figure 2.2.

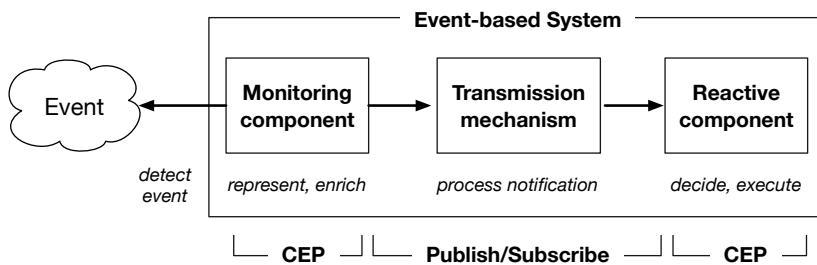


Figure 2.2.: Minimal set of components of an EBS as described in [208].

The transmission mechanism for notifications can be based on different interaction models as shown in Figure 2.3. While it is not mandatory, communication in an EBS is usually *push-based*, e.g., using messages or publish/subscribe. With the producer initiating the interaction, information is delivered to the consumer upon availability without the consumer having to constantly poll the producer for potential updates. This reflects the reactive nature of an EBS [208, 318].

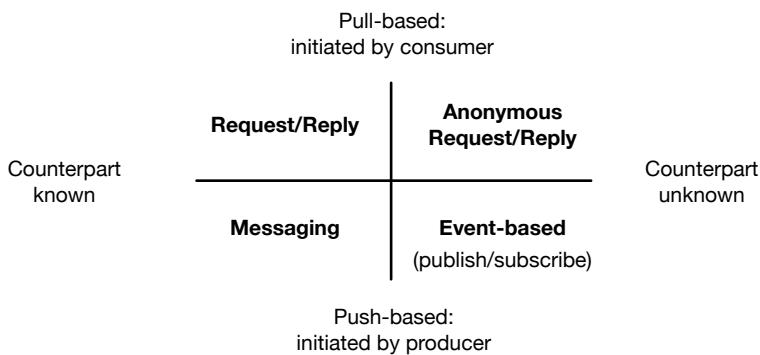


Figure 2.3.: Interaction models as discussed in [77].

In the remainder of this dissertation, we focus on two complementary approaches that can be used in an EBS to address key challenges in reactive applications: notifications are dispatched from many different producers to all interested consumers using the Publish/Subscribe (PS) interaction model while Complex Event Processing (CEP) turns notifications about simple and composite events into meaningful information, detects complex events and reacts to them.

2.1.1 Publish/Subscribe for Dispatching Event Notifications

Publish/Subscribe (PS) systems dispatch notifications from data sources to consumers in an asynchronous, push-based fashion. They are anonymous in that they do not require participants to know each other's identity to exchange information. Dispatching is usually done using a Message-oriented Middleware (MOM) that decouples data sources from consumers in time, space and synchronization [87, 147, 172].

As illustrated in Figure 2.4, publishers report each detected event by *publishing* a notification about this event to the MOM. The MOM forwards each notification to those subscribers that have expressed their interest by registering *subscriptions* at the MOM; subsequent lack of interest is expressed by unsubscribing from already subscribed events. Publishers can announce the events they are able to detect to the MOM using *advertisements*.

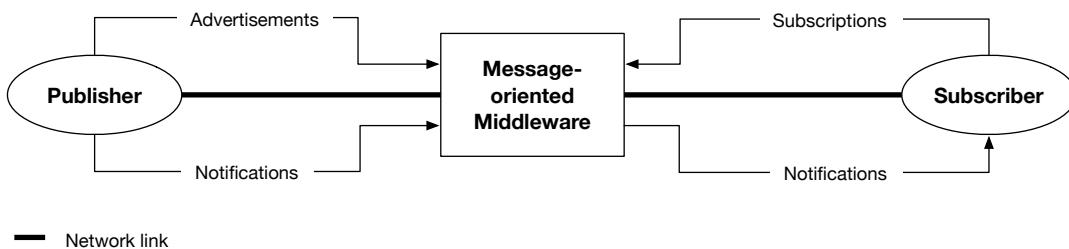


Figure 2.4.: Components of a PS system to dispatch notifications: publishers, subscribers, MOM.

Subscribers and publishers use an Application Programming Interface (API) offered by the MOM for advertising, subscribing and publishing notifications [350]. Subscribers provide a callback function that is triggered by the MOM whenever publishers provide matching notifications. Publishers, on the other hand, publish notifications in a fire-and-forget fashion using a callback method provided by the MOM. Thus, subscribers and publishers are fully decoupled and participants of either type can be added or removed at runtime without interrupting the system. Furthermore, subscribers and publishers are assumed to be unaware about the presence of other participants when deciding on subscriptions, publications or advertisements. This increases flexibility and supports scalability [87, 147, 172, 318].

Efficiently identifying the set of subscribers interested in a certain notification is a key challenge in PS systems and done by matching subscriptions to advertisements or notifications using different filtering mechanisms [208, 318]. As discussed in [17], the granularity of these filter mechanisms is either *per-connection* or *per-event*.

Channel-, subject- and topic-based approaches are per-connection and use static routing to identify interested subscribers: publishers publish their notifications on a given channel, e.g., sports. All subscribers subscribed to this channel receive every notification published there, regardless of its content [7, 17, 87, 329]. Hierarchical addressing (e.g., sports.soccer.germany) and wildcards (sports.*.germany) have been introduced to increase the flexibility of the otherwise limited expressiveness [87, 147].

Contrastingly, content-based [317, 363], type-based [146, 147, 148], or concept-based approaches [15, 104] identify the affected subscribers per event. Type-based schemes allow to

advertise and publish notifications of a certain type that subscribers can subscribe to. This approach overcomes the restrictions of hierarchy-based subject- or channel-based approaches while allowing for type-safety checks and advertisements due to subtyping. Concept-based approaches have been introduced to deal with the increasing heterogeneity between subscribers and publishers [15, 104]. In contrast to the other subscription schemes, this approach does not assume a common understanding of the namespace used by publishers and subscribers. Thus, it can be implemented on top of any of the other subscription schemes.

In practice, MOMs often support a mix of subscription schemes [139, 172, 387] to balance the costs of per-event approaches with the performance gains of per-connection approaches [368]. For example, combining topic- or type-based approaches with a content-based scheme enables the MOM to efficiently filter notifications based on advertisements while providing subscribers with additional expressiveness for their subscriptions [131, 172, 390, 404]. In MOMs supporting Java Message Service (JMS), this can be done by subscribing to dedicated topics with additional filters that are matched to attributes of each JMS; in the REconfigurable Dispatching System (REDS), subscriptions are defined for a given type of notification together with optional constraints on the set of attributes contained in the notification.

A MOM can be realized as a centralized message broker or as a Distributed Event-based System (DEBS) relying on a distributed and decentralized network of brokers. In a DEBS, the network of brokers forms a decentralized *overlay* on top of network links that interconnect publishers and subscribers as illustrated in Figure 2.5.

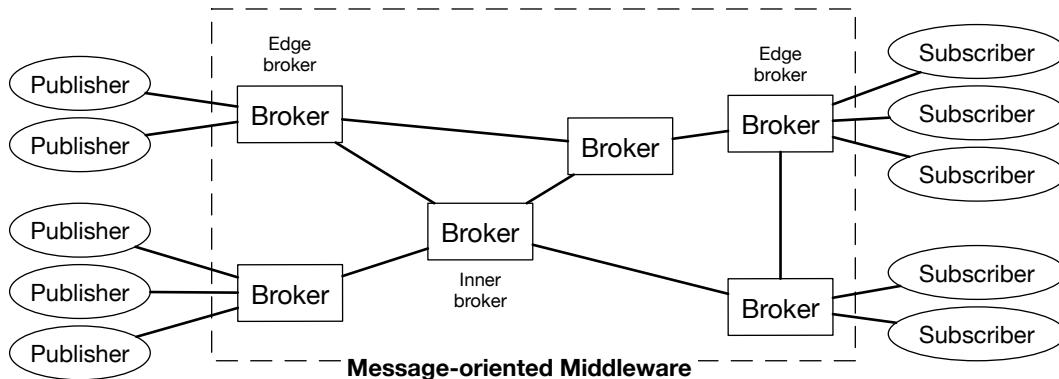


Figure 2.5.: Distributed network of brokers (B) forming a MOM.

Edge brokers are directly connected to subscribers or publishers while inner brokers perform efficient filtering and forwarding of notifications from publisher-side edge brokers to subscriber-side edge brokers. To avoid flooding the broker network when propagating notifications is a key challenge in any DEBS. To this end, each broker stores information on directly connected clients and neighboring brokers, e.g., advertisements and subscriptions. Based on this information, *routing trees* from publishers to all interested subscribers are established and maintained [318].

Examples for centralized MOMs are Apache ActiveMQ [390] and IBM Websphere [204] while distributed MOMs are mostly research prototypes, e.g., REDS [126], HERMES [349], SIENA [88, 242], REBECA [15, 432], DREAM [77], CREAM [105], or PADRES [157]. Further examples are surveyed in [32, 58, 118, 129, 281].

2.1.2 Complex Event Processing for Reasoning and Deciding on Events

CEP engines are software components that enrich, combine and interpret notifications from multiple publishers to infer whether a meaningful event has taken place [78, 89, 145, 284]. Upon detecting such an event, the CEP engine can propagate this knowledge as a new notification or react directly, e.g., by triggering a suitable business process [18]. In the PS communication model of an EBS, CEP engines thus act as subscribers *and* publishers.

Within a sequence of notifications about events, CEP engines try to identify patterns that indicate a complex event has taken place or is about to take place [145, 251, 268]. Patterns are expressed using *operators* such as selection, combination, negation, aggregation or production of new notifications about composite or complex events [124]. Patterns about multiple related events can be pre-defined or learned by the CEP engine using machine learning [294].

On the technological level, pattern matching is realized by continuous queries that have to be stopped explicitly. Unlike pull-based queries in database systems that reflect the state of the database at the time of posing the query, continuous queries (a.k.a long-running, standing, persistent queries) continuously monitor the streams of incoming notifications and push results to the consumer every time the query matches the content of one or multiple streams [30, 137, 186, 278]. In distributed setups, *distribution strategies* have to deal with the operator placement problem: decide at which node an operator is to be executed [123, 124].

Examples for query languages are TESLA (Trio-based Event Specification LAnguage) [120] or EP-SPARQL [14]. Examples for CEP engines or MOMs that distributed CEP functionality are T-Rex [121], CommonSens [392, 393], or RACED [119]. A more comprehensive overview of query languages and CEP solutions can be found in [120, 122, 137].

Dealing with *uncertainty* is a major challenge in complex event processing. Information consumed by the CEP engine can be imprecise, incorrect, or incomplete, resulting in uncertainty about the validity of the derived information. Reasons for this can be incomplete, delayed, or out-of-band notification propagation by the PS system as well as inaccurate raw data provided by publishers. In addition, pattern definitions can be defined incorrectly based on wrong assumptions [122, 124, 125]. Thus, the complex events derived are flawed with a certain level of uncertainty that is expressed by a confidence of detection.

2.2 Related Concepts and the Role of EBS

In the previous sections, we have introduced PS and CEP as the two aspects of a push-based EBS that this dissertation focuses on. In the remainder of this chapter, we introduce other concepts and paradigms that apply the concept of an EBS, use it, or are similar in their motivation and characteristics.

2.2.1 Wireless Sensor Networks and Cyber-Physical Systems

On the device level, increasing miniaturization and decreasing production costs enable a myriad of sensors to be used in monitoring real-world conditions while actuators can manipulate objects and processes in the real-world. Wireless Sensor Networks (WSNs) and Cyber-physical Systems (CPSs) refer to large federations of *sensor nodes* — low-cost physical devices that combine different sensors and actuators with wireless transceivers, processing units, and a power unit [11, 35, 90, 111, 355, 397].

The resulting variety of data sources bridges the gap between the physical and the digital world by providing software systems with continuous streams of fine-granular but heterogeneous data about real-world events, processes, and objects [43, 66, 254].

Sensor nodes in a WSN or CPS act as publishers that report to a subscriber outside the sensor network using a push-based communication model. Sensor nodes have comparably limited sensing, processing and transmission capabilities but are deployed densely in large numbers. Thus they have to cooperate with each other to sense a given phenomenon and process the information for the subscriber. For this, they autonomously establish and maintain multi-hop routing topologies to forward information from sensing nodes in a peer-to-peer fashion. Intermediate sensor nodes switch between sensing information and forwarding information received from peers. Thus, routing topologies can be hierarchically structured and change frequently based on which sensor nodes are available and capable for forwarding at runtime [11, 193, 235, 236, 273, 355, 396, 435].

In order to ease the cooperation of nodes, sensor nodes in a typical WSN are assumed to be homogeneous, perform a dedicated sensing task and report to a single subscriber [11, 355, 397]. Contrastingly, a CPS in the context of the IoT is assumed to consist of interacting heterogeneous devices, ranging from passive Radio-Frequency IDentification (RFID) tags to mobile devices or body-sensor networks [27, 193, 273, 376, 377, 378, 379, 397, 431].

In a CPS, the heterogeneity of sensor nodes in terms of sensing and processing capabilities requires new approaches to identify those sets of sensor nodes that can provide relevant data for a certain task [52, 140, 340, 342, 392, 411]. In this regard, sensor nodes have to become self-aware and able to describe their current capabilities and exchange this information with their peers [102, 109, 110, 140, 411, 418].

The major challenge in any WSN or CPS is managing energy consumption for sensing, processing and transmission as sensor nodes have limited power supply; many are battery-powered and replacement is usually infrequent or not possible at all. Hence, all tasks are priced at runtime based on their energy consumption and sensor nodes adapt their capabilities based on their current power level. Location, power level and role of a sensor node change at runtime and can result in inaccuracy and uncertainty of the provided data as the ambient context and situation affect the capabilities of a sensor node [54, 56, 96, 99, 224, 271, 300, 422, 431, 435].

Consequently, research in the area of WSN and CPS focuses on energy-efficient sensing, processing, and transmission. This encompasses compression but also distributing CEP logic in a federation of sensors to minimize data transmission by aggregation, fusion and reasoning inside the network [45, 99, 152, 194, 224, 287, 343, 344, 354, 356, 376, 402]. Besides minimizing energy consumption in general, certain approaches focus on explicitly trading off energy consumption with the sensitivity, precision, completeness and reliability of sensors [79, 80, 201, 325, 386, 431].

Examples for push-based MOMs that address the specific characteristics of sensor nodes are Tiny-DDS [65] or TinyCOPS [200]; examples for other frameworks and MOMs for sensor networks can be found in [206, 344, 354, 392, 376].

2.2.2 Data Stream Management Systems

The intention of a Data Stream Management System (DSMS) is very similar to that of a CEP engine: generate new information on a higher level of abstraction by processing continuous

streams of incoming lower-level information. A DSMS applies continuous queries to streams of homogeneous information provided at high volume from known data sources [30, 98, 278]. Examples are transaction logs of highly utilized web-based applications or financial ticks [10, 73, 186, 245, 262, 266, 409]. The challenges faced in a DSMS are similar to those in CEP, e.g., high-throughput and low-latency execution of continuous queries [231, 245, 323], or operator placement in distributed setups [100, 351].

Work on DSMSs and CEP engines increasingly overlaps but has originated from different research communities. DSMSs and CEP are introduced in different sections of this dissertation following the distinction described in [91, 122, 124, 186]: DSMSs *transform* incoming streams of information into new streams of information with a higher information density using sequences of predefined operators such as averaging, grouping, and selections [122, 186]. In contrast, CEP engines *identify meaningful situations* and publish this information as new notifications in addition to the existing streams [122].

In regard to the contributions of this dissertation, the distinction between DSMS and CEP engines is not of utmost importance as DSMSs also apply a push-based communication model and act as both subscribers and publishers in a PS system. From the perspective of this dissertation, DSMSs focus on a subset of tasks addressed by CEP.

In particular relevant for this dissertation is the fact that a DSMS explicitly trades off Quality of Information (QoI) against costs when applying distribution and optimization strategies: QoI is expressed in terms of accuracy, correctness, processing latency, and output rate¹ while processing costs are quantified in terms of memory and power consumption [10, 186, 245, 246]. Surveys and more detailed descriptions about DSMSs are provided in [29, 181, 185, 398].

2.2.3 Service-Oriented Architectures

Host-centric Service-oriented Architectures (SOAs) are the backbone of current backend software systems. They implement business processes by relying on persistent data and stable workflow definitions that identify the participating applications. SOA enables reuse and modularization by encapsulating functionalities and their data in *services* that can be only accessed using implementation-independent interfaces. Services interact with each other by relying on implementation-independent formats, e.g., using the Extensible Markup Language (XML) [154, 155, 156, 169, 333, 407, 423].

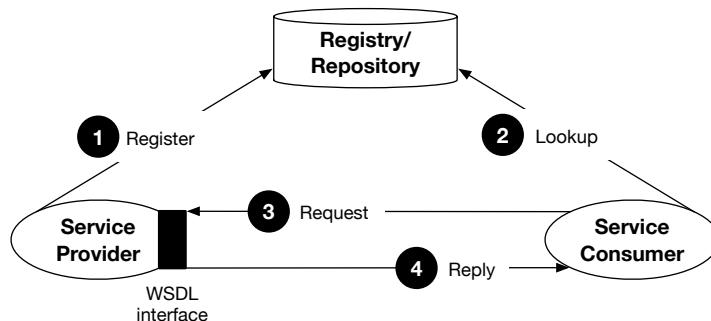


Figure 2.6.: Service consumer and service provider in a SOA.

Figure 2.6 shows the steps of the request/reply-based interaction between the participants in a SOA with service consumers *pulling* data from service providers: a service provider registers its

¹ Sampling rate of the DSMS.

capabilities at a Registry. A service consumer requiring a certain service performs a lookup at the Registry that returns the endpoint to reach the service provider. The service consumer then sends a request directly to the interface of the service provider, e.g., described in the Web Services Description Language (WSDL).

SOA had been intended to facilitate the integration of inter-organizational software systems; at present, it is used mostly for intra-organizational application integration. Especially large companies, such as Deutsche Post DHL, use SOA to integrate and optimize their historically grown heterogeneous application landscape. From an architectural point of view, applying the concept of a SOA reduces complexity and redundancy: the application landscape has to be restructured according to functionality and data ownership as services are the basic entities within a SOA and are organized in domains without overlap. From an infrastructure point of view, however, services are usually provided and consumed by applications in practice. These applications provide and supply multiple services and have to be available according to their business criticality. The desired level of availability is specified in Service Level Agreements (SLAs) in terms of Service Level Objective (SLO) per application. These SLAs and SLOs have to be broken down and enforced on a per-service level [67, 169, 205].

The WS-Agreement protocol is one example for approaches to automatically formalize and directly negotiate SLAs between a service provider and a service consumer. WS-Agreement has been proposed by the Open Grid Forum (OGF) and is based on XML. It is favored in both academic and industrial systems for service-based systems as extensions allow for automated renegotiation of agreements as well as for multi-round negotiations. The *WS-Agreement for Java framework (WSAG4J)*² provides a basic set of libraries to process SLAs in Java based on the WS-Agreement specification [38, 227, 264, 267, 285, 332, 331, 380, 436].

While the scope of this dissertation is on push-based systems, modern enterprise software systems increasingly combine push- and pull-based paradigms [18, 78, 170]. For example, services in a SOA are invoked by components of an EBS that receive or detect meaningful events. Service invocations, in turn, can result in meaningful changes to a system, triggering components of an EBS to publish notifications about these events [17, 115, 307]. Push-based approaches are already widely used to monitor the behavior of pull-based processes and architectures [13, 34, 97, 218, 309, 419, 433]. Proposed proprietary standards such as WS-Notification, and WS-Eventing try to integrate push-based communication into SOA [17, 69, 187, 229, 416].

2.2.4 Cloud Computing

On the infrastructure level, the combination of several technological and economic concepts, known as *Cloud computing*, has caused a disruption in the way large pull- and push-based applications are built, deployed and operated. The term Cloud computing does not refer to a single technology or specific product; it refers to offering computing resources as a commodity over the Internet on a pay-per-use basis [303]. Cloud computing resources are hosted in multiple data centers allowing for instantaneous scaling and load-balancing of applications on a global scale without the need for upfront investments [24, 25, 162, 164, 174, 184, 189, 232, 434].

In the terminology of Cloud computing, resources are rented out to *tenants* by *providers*. A tenant uses the resources to provide functionality to *users*. In a *public* Cloud, multiple tenants share the same physical resources provided by the provider. In a *private* Cloud, all resources are

² <http://wsag4j.sourceforge.net/site/index.html>

exclusively used by a single tenant. In a *hybrid* setup, sudden bursts in demand are compensated for by temporarily adding public Cloud resources to the resources available in a private Cloud.

Different provisioning models can be distinguished for Cloud computing depending on the level of abstraction that the resources are accessible to tenants: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS); on-premise and MSH are shown in Figure 2.7 as alternative models [174, 184, 190, 303].

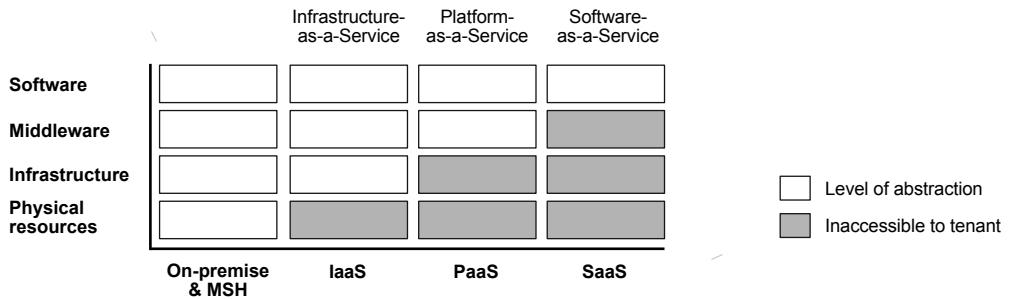


Figure 2.7.: Resource provisioning models for Cloud computing.

On-premise and Managed Service Hosting (MSH). The whole technology stack is controlled by the tenant. Physical servers are owned by a single tenant (on-premise) or rented exclusively to it (MSH). These are common approaches that require upfront investment and include the risk of ill-sized resource planning, i.e., under- or overprovisioning.

Infrastructure-as-a-Service (IaaS). Tenants request Virtual Machine (VM) instances hosted on a shared resource pool. The structure, location and usage of the resource pool are transparent to the tenant. Thus, provisioning, scheduling and load-balancing can be performed by the provider. Pricing is based on the number, configuration and uptime of the active instances.

Platform-as-a-Service (PaaS). Middleware functionality, such as a specific database management system (DBMS), is provided through an API. The implementation and detailed configuration of the underlying infrastructure is inaccessible to the tenant. The provider is responsible for automatically scaling the underlying resources to the utilization of the middleware.

Software-as-a-Service (SaaS). An application is provided to tenants to be configured and integrated into their application landscape. SaaS mostly aims directly at end-users by providing high-level user-experience via Rich Internet Applications (RIA). The provider is responsible for scaling all underlying resources. Pricing is based on usage or subscription.

Comparing Cloud computing to Utility computing and Grid computing shows that Cloud computing is the synthesis of several preceding concepts and technologies. All concepts aim at delivering abstracted computing resources to multiple tenants [164]. While Utility computing describes the general approach, the other concepts focus on different ways of realization [290, 364]. The concept of Grid computing emerged in the 1990s and aims at delivering abstracted computational resources drawn from a distributed inter-organizational resource pool [184]. Each participating organization keeps controlling the resources committed to the pool while being entitled to use the whole Grid. The abstraction of the resource pool and job scheduling is done using a multilayered software fabric. While multiple tenants are served by a Cloud provider at the same time, tenants queue to use the whole Grid one at a time. Furthermore, a Grid consists of resources owned and operated by different organizations while a Cloud is provided by a single

provider. On the technological level, Cloud resources are provided using a standardized interface over a network while Grids require running the Grid fabric software locally on each tenant's infrastructure [163].

Cloud computing is the result of five technological and five economic enablers reinforcing each other as shown in Figure 2.8.

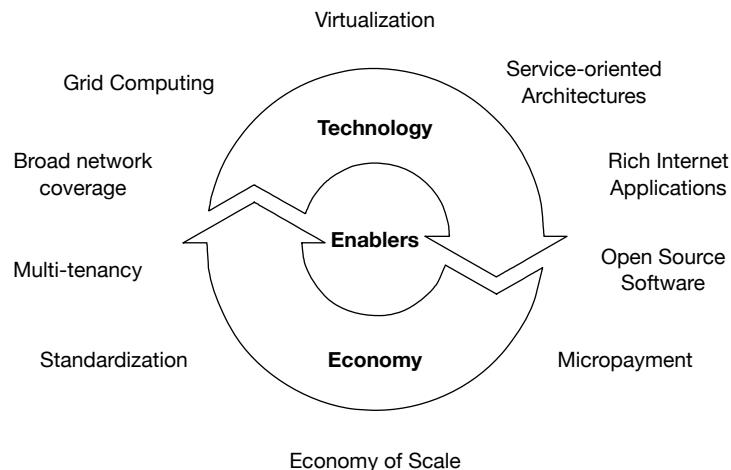


Figure 2.8.: Key enablers for Cloud computing as described in [174].

Virtualization and *Grid computing* form the fabric that enables resource pooling and rapid elasticity [164, 184]. SOA reshaped software architectures to run on virtualized distributed resources efficiently as loosely-coupled, component-based services [169, 303]. *Economy of scale*, *standardization* and *open source software* make Cloud services cost-effective by rewarding scale, reducing complexity and minimizing license fees as well as lock-in effects [103, 189, 338]. *RIA* and *broad network coverage* simplify the delivery of computational functionality as on-demand self-services. *Micropayment* permits effectively charging the usage of resources. *Multi-tenancy* of Cloud resources maximizes both utilization and risk-reduction for the provider [33].

Cloud resources conveniently available on demand have boosted analytical applications, such as Hadoop, for processing and storing data on a large scale. Operating pull- and push-based software architectures for timely dissemination of data on those resources, however, comes literally at a cost: traffic in and out of data centers is charged for while the virtualization of resources limits vertical integration for performance tuning. In addition, scheduling, load-balancing and maintenance operations are transparent to the tenant and can affect the completeness and latency that notifications are processed with [173, 174, 197].

2.2.5 Systems-of-Systems and the Vision of Emergent Software Systems

Advances in software engineering and programming language design enable software components to become increasingly context-aware and self-adaptive to react autonomously to changes in their context and state [2, 3, 60, 61, 63, 68, 71, 114, 136, 221, 222, 220, 353].

In the IoT, self-adaptive software systems utilize the fine-granular information provided by WSNs and a CPS to become self-aware [301]. The push-based nature of an EBS allows them not only to be informed about changes in their physical or digital environment, but using CEP, they

become reactive. The resulting applications are distributed and form a federated System-of-Systems (SoS) with a high degree of heterogeneity and dynamics. Participants join or leave at runtime depending on their state or the situation they are in. Thus, the notion of *context* is essential to IoT applications as it captures the effect both state and situation of a participant have on its requirements and capabilities at runtime.

Self-aware, self-organizing and reactive software systems fuel the vision of proactive software systems with *emergent* behavior [170, 222]. Emergence enables SoS to autonomously adapt to changing situations and expose new functionality that has not been explicitly designed into them beforehand [179, 217]. Emergent behavior refers to the successful combination of uncoordinated interactions by the different autonomous entities making up a SoS. Emergence materializes if the interactions inside a system create an advantage for the system within a given context. With these new emergent abilities a system can now pursue objectives that would be too complex for a single entity to handle [63]. In that, emergent systems are not only state-preserving (i.e. self-organizing and adaptive) but *proactive* as they are able to actively utilize changes in their environment to their own benefit [178]. Consequently, the concept of emergence has long since been subject to multiple areas of research spanning from biology to philosophy to mechanical engineering [178, 399]; in computer science work on autonomous systems, organic computing, and SoS from the military domain address emergence [59, 63, 130, 228, 426].

In such federated and autonomous systems, decisions are based on local knowledge and information exchanged between participants. The quality of this information is crucial for participants to make correct decisions and coordinate with each other [71, 127, 136].

2.3 Summary

In this chapter, we have introduced push- and pull-based concepts; QoI is crucial in all of them. This dissertation focuses on push-based Event-based Systems (EBSs) that use Publish/Subscribe (PS) for the efficient dissemination of notifications about events and Complex Event Processing (CEP) to generate new knowledge about situations.

Wireless Sensor Networks (WSNs) and Cyber-physical Systems (CPSs) are concepts directly applying PS and CEP while facing challenges such as specialization of sensor nodes and resource constraints. The trade-offs between sensing, processing and energy-consumption result in dynamically changing capabilities of sensor nodes at runtime, affecting the precision and reliability of the published information.

We have discussed Data Stream Management Systems (DSMSs) as directly related to CEP. Both concepts deduce new knowledge by querying streams of notifications. They have to deal with imprecise, incorrect or incomplete information as well as incorrect query definitions.

Widening the scope to include pull-based approaches fusing with push-based concepts, we have briefly discussed Service-oriented Architectures (SOAs) as backbones of today's enterprise software systems and technological enablers of Cloud computing. We have discussed the notion of Cloud computing with its technological as well as economic enablers regarding the implications for deploying and operating reactive applications. Finally, we have briefly touched upon the vision of System-of-Systems (SoS), which describes federations of self-aware, self-organizing, and reactive autonomous applications that leverage the concept of emergence to realize new functionalities without requiring up-front design.

3 A Generic Model to Express Quality of Information Requirements in EBS

People are extremely bad at specifying what they really need.

JOHN WILKES (2015)

Supporting Quality of Information (QoI) in an Event-based System (EBS) means to satisfy requirements of subscribers about data being produced by publishers and processed by a Message-oriented Middleware (MOM) [42, 85, 213]. This requires the MOM to match the requirements of subscribers to the current state of the system and adapt both publishers and itself to satisfy the requirements at runtime if necessary [170, 175, 177].

In this chapter, we present the model of *expectations*, *capabilities*, and *feedback* to support requirements about QoI in an EBS. As illustrated in Figure 3.1, subscribers express their requirements about generic properties of notifications as expectations while publishers and the MOM define their support for these generic properties as capabilities. Expectations and capabilities depend on the context and state of subscribers, publishers, and the MOM at runtime. At runtime, the MOM matches expectations to capabilities, decides on suitable adaptations to satisfy expectations, and gives feedback to both subscribers and publishers. We will discuss the algorithms for these decision processes in detail in Chapter 4 while we focus on describing the different components of our model and their semantics in this chapter.

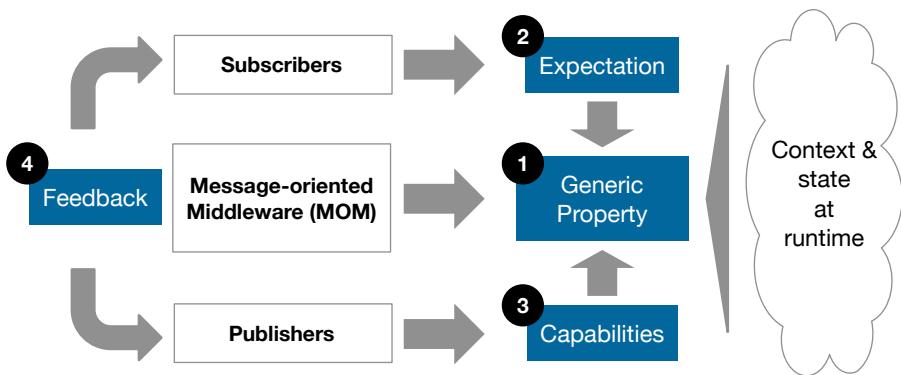


Figure 3.1.: The model of expectations, capabilities and feedback.

First, we discuss the relationship between QoI and related concepts such as Quality of Service (QoS), Quality of Context (QoC), or Quality of Experience (QoE) in Section 3.2. We revise a spectrum of properties that characterize information in addition to its content or type in an EBS and the Internet of Things (IoT). We identify common features in the representation of these characteristics to derive a *generic property* format for defining expectations and capabilities.

Second, we introduce *expectations* to express requirements subscribers have about QoI in an EBS at runtime in Section 3.3. We give special attention to the effect that context changes have on

these requirements and their lifecycle. We introduce the notion of *fidelity* to quantify the degree to which the system satisfies QoI requirements from the perspective of a single subscriber.

Third, we introduce *capabilities* to expose each participant’s current state and potential spectrum of adaptation to the MOM in Section 3.4. The MOM uses capabilities to determine the support for properties and assess potential benefits to be gained from adaptation. As capabilities depend on a participant’s context and state at runtime, we allow participants to price-in their current situation and context into the costs for adaptation.

Fourth, we describe the types of feedback the MOM provides to participants at runtime in Section 3.5. Feedback is a key component of our approach as it enables self-adaptation. Participants use it to revise their expectations or capabilities and trigger renegotiation at runtime.

3.1 Quality of Information and Related Concepts

Describing and evaluating the usability of a system at runtime is approached from various perspectives using different concepts all related to the abstract notion of runtime quality. Quality of Device (QoD), Quality of Service (QoS), Quality of Information (QoI)¹, Quality of Experience (QoE), and Quality of Context (QoC) all focus on quantifying the different characteristics of a system in order to derive a degree of suitability [226]. Thereby, they try to bridge the gap between objectively measuring system characteristics and deriving subjective application-specific evaluations. Each concept focuses on different aspects of runtime quality depending on the area of research each concept has originated from: the quality perceived by users in the area of Human-centered Computing (HCC) and Human Computer Interaction (HCI) is central to QoE [74, 226, 234], the performance of devices and entities to QoD [75] and QoS [21, 23, 28, 82, 402], while QoI focuses on the suitability of information generated and delivered by the system in general [55, 421]. As a subset of QoI for the domains of Ambient Intelligence (AmI), HCC, and HCI, QoC evaluates the suitability of a given context model for identifying a specific context and situation by relying on a limited set of QoI characteristics [6, 75, 324]; QoE captures the overall satisfaction perceived by a user [159, 234].

In this section, we discuss these different concepts and their relationships from the perspective of QoI. Our goal is to establish a nomenclature to be used throughout the remainder of this dissertation. We aim at separating the objectively measurable domain- and application-agnostic properties of a system and their dependencies from subjective application-specific aggregated assessments of these characteristics. Requirements about those objectively measured properties become comparable across different subscribers and should be subject to mechanisms enforcing QoI requirements in a generic way; application-specific assessments, however, should be encapsulated within the business logic of each subscriber and not exposed to the MOM.

As shown in Figure 3.2, we see runtime quality as an area of conflict between subjective gain on a high level of abstraction that results from having information of suitable quality available (top, red) and objectively measurable capabilities and costs for providing this information on a lower level of abstraction (bottom, green). In data-centric reactive software systems like EBSs, QoI is crucial as it focuses on the information that is being produced, processed and exploited [21]. From this perspective, all other runtime quality concepts can be centered around QoI depending on whether they impact it or rely on it [226]. We traverse the graph of concepts shown in Figure 3.2 bottom-up to discuss the relationships.

¹ Sometimes confusingly referred to as Quality of Data, resulting in the ambiguous abbreviation QoD [425].

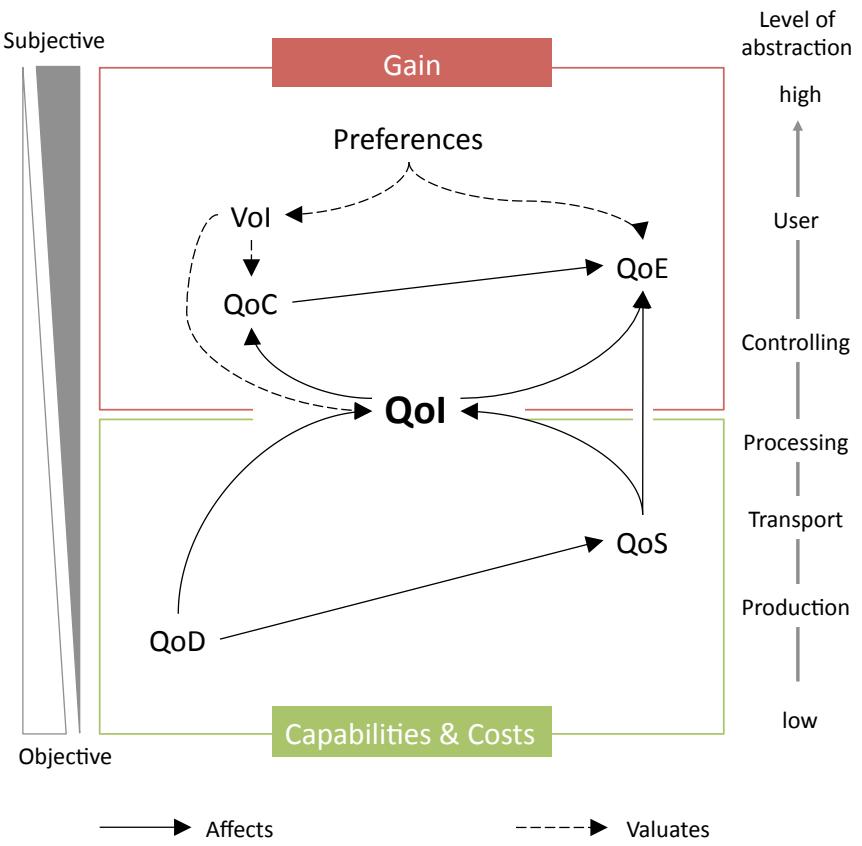


Figure 3.2.: Different concepts related to the notion of runtime quality and their relationships (affecting or valuating); levels of abstraction are based on [21].

Quality of Device (QoD). On the level of data production, QoD describes the performance and capabilities of a specific device as shown in the lower part of Figure 3.2. This includes physical devices, such as sensors, as well as software components that act as publishers. For example, a camera sensor is part of an intrusion detection system and provides pictures with a certain *resolution* at a given *sampling rate*; a Complex Event Processing (CEP) engine fuses this data to publish notifications once an intruder has been detected using different fusion mechanisms that result in more or less false-positives and false-negatives denoted as *confidence of detection* [54, 224]. Other examples: a temperature sensor measures a room's temperature with 98% *precision* while another temperature sensor has a precision of 85% [75]; a Global Positioning System (GPS) sensor can have a certain *drift* when measuring its *location*, resulting in biased location information [293, 324, 340].

Quality of Service (QoS). The properties exemplarily discussed here for QoD are often subsumed under the broader notion of QoS. QoS describes the overall performance and capabilities of a system that consists of multiple entities. QoS spans across several levels of abstraction and includes the communication and interaction between entities, e.g., by considering network links [226], and consequently addresses broader aspects of systems such as scalability, availability and dependability [207]. In contrast to QoD properties, QoS properties include aspects of lower-level transportation and processing such as *compression, bandwidth, latency, jitter, loss, and security* [22, 56, 210, 288, 354]. For EBS, additional QoS

properties like *order*, *completeness*², and *delivery semantics* are defined for sequences of notifications [42, 47, 95, 125, 122, 208, 210, 288, 318].

QoD influences QoS as the capabilities of the system depend on the capabilities of the physical or virtual devices making up the system [75, 226]. We follow [75] in discussing QoD and QoS separately as this maps to the tripartition of an EBS, which consists of publishers, subscribers, and MOM with different requirements and capabilities regarding runtime quality [41, 42] as discussed in Section 2.1.1.

Quality of Information (QoI). While QoD and QoS are objectively measured [226, 263, 291], the concept of QoI focuses on whether information is "fit for use" for a specific application, task or user [55, 75, 224, 226, 250]. As this depends on the individual situation and purpose of each subscriber, the concept of QoI is split into two parts: objectively measureable *QoI properties* that describe the inherent quality of an information item and application-specific *Value of Information (VoI) properties* that quantify the subjective utility of those QoI properties for the given application, situation and context [57]. For example, QoI properties commonly referred to are *accuracy, freshness, precision, spatial and temporal resolution, confidence, or completeness*³ [31, 37, 244, 250, 324]. QoI properties are inherent to a single notification or a sequence of notifications.

Whether these properties of a specific notification are distinctive enough to satisfy the individual requirements of a dedicated consumer is evaluated by VoI functions like *integrity, coherence, currency, validity, relevance, or understandability* that reflect preferences of a subscriber [37, 244, 271, 291, 366]. QoD and QoS influence certain QoI properties. For example, the drift of a GPS sensor, its sampling rate and the latency of the network link have an impact on the accuracy of notifications about a user's position in terms of spatial and temporal resolution: location based services could assume an incorrect or outdated position of the user if notifications are received late, with insufficient resolution or biased position data [293, 324, 340, 374].

Quality of Context (QoC) and Quality of Experience (QoE). The dependency on a specific context or a single user's preferences is more evident for QoC and QoE as these concepts are connoted with direct human involvement in the area of AmI, HCI, and HCC [74]. QoC approaches runtime quality from the domain of AmI and focuses on correctly deriving the current physical and even emotional context of a user to react on it [64, 339]. Thus, QoC relies heavily on the quality of the context model used to describe different user contexts as well as on QoI of data supplied by sensors to correctly identify the context a user is currently in [75, 226, 340]. Most models and approaches dealing with QoC assess subjectively whether data is *reliable, significant or consistent* enough based on application-specific accumulation of objective properties like precision, accuracy, sampling rate, and location. Especially spatial or temporal resolution is an important factor when determining the fitness of a context model [291, 292, 293, 324]. QoE has an even wider scope as it embraces aspects of QoC and QoS, focusing not only on the suitability of data provided by a service but also on how the performance of a service is *perceived* by a user [74, 159, 234, 425].

3.2 Properties: the Basic Building Blocks

Analyzing the relationships between the different concepts addressing runtime quality shows that QoI is essential for tasks that require data of sufficient quality for reasoning and inference.

² Percentage of published notifications that are delivered to an interested subscriber by the MOM [42, 95, 208].

³ The number of attributes in a single notification compared to the reference number of attributes [5, 6, 182].

Although the final assessment whether some information is of sufficient quality or not is based on the individual utility function of a subscriber, it depends on properties being explicitly supported by the system on lower levels of abstraction [159, 234]. For example, highly aggregated domain-specific VoI functions like *currency* or *validity* for QoC accumulate objective lower-level properties such as the overall *freshness* (QoI) of a notification, which depends on the *latency* (QoS) of the system and the *sampling rate* (QoD) of a data source [6, 271].

Support for a property implies not only that requirements about this property can be defined or decomposed from requirements about other properties. Rather it entails the ability of the MOM to be aware of the state of this property and to manipulate it with suitable actions. In an EBS, some properties can be determined and manipulated only by publishers (e.g., precision) while others depend on the capabilities of the MOM. These dependencies have to be taken into account by the system when offering explicit support for certain properties. For example, a system that cannot adapt publishers cannot actively enforce requirements about precision other than by discarding notifications with insufficient precision.

From the perspective of a subscriber, subjective assessments of information quality at some point require a decomposition and mapping to requirements about those properties that are explicitly supported by the system [381]. Properties describe objectively measurable system characteristics on different levels of abstraction, here denoted by QoD, QoS, and QoI in Figure 3.3 (far left).

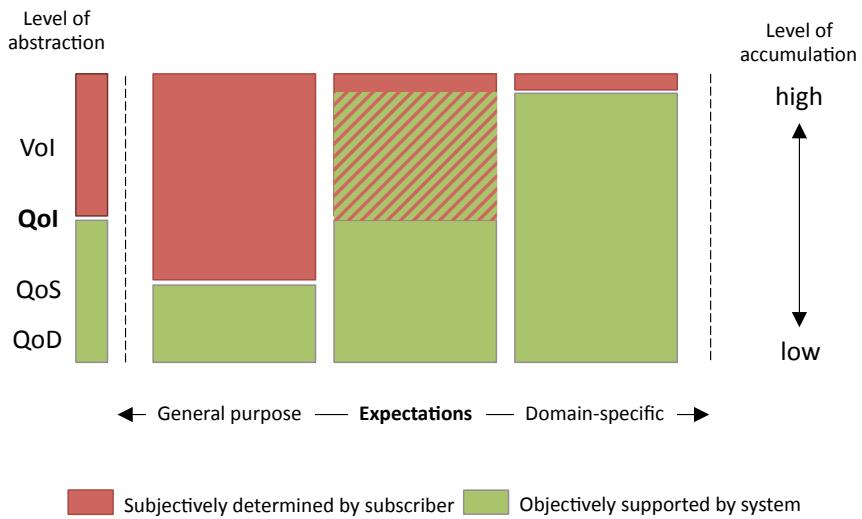


Figure 3.3.: Schematic view on the levels of abstraction that requirements about properties are provided at in general-purpose (left) and domain-specific systems (right).

Providing generic support for QoI in an EBS requires a flexible trade-off between the level of abstraction that properties are exposed on to subscribers and the degree of accumulation that is transparently performed in the system [35]. General-purpose systems as shown in Figure 3.3 (left) do not make assumptions about the connected applications but explicitly support objective low-level properties. They require subscribers to break down and map their preferences to the exposed lower-level properties for QoS, as these are the properties actively controlled by the system. Examples are IndiQoS [85], Adamant [213], Data Distribution Service (DDS) [197, 269, 334, 345], or Harmony [428] for *bandwidth* or *latency*. However, this lim-

its the support for information quality as some QoI properties cannot be directly inferred from lower-level properties alone [21]. At the other end of the spectrum, domain-specific systems (c.f., Figure 3.3, right) allow for requirements about pre-defined VoI properties like *significance*. They already aggregate and accumulate lower-level properties in a domain-specific way while hiding lower-level properties to the subscriber; the definitions and relationships of properties are predefined in the system, limiting support for applications from other domains. An example is the QoC extension [293] to the COSINE system [243].

We enable an EBS to decide on the level of abstraction that support for QoI is provided on at runtime as illustrated by Figure 3.3 (middle). As part of our approach, we model properties in a uniform way as shown in Figure 3.4, separating the *definition* of a property from its *relationships* to other properties and the *actions* that indicate the directions it can be manipulated.

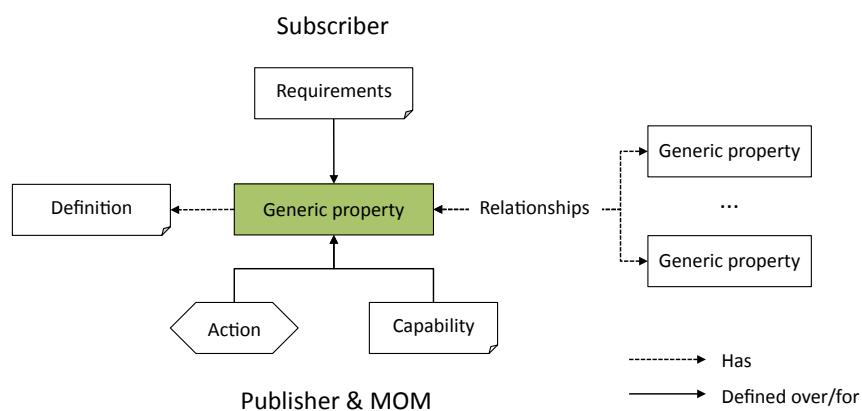


Figure 3.4.: Modular representation of a generic property: separating the definition from relationships and actions enables modularization, reuse, and adaptation.

A uniform representation for generic properties makes requirements and capabilities comparable; using explicitly modeled relationships between properties, we can model higher-level generic properties by defining dependencies to already defined properties.

We associate each generic property with at least one action that indicates whether the current value of a property can be increased or decreased from the perspective of the MOM. An action does not describe explicitly how the generic property is manipulated but leaves this to the participant executing the manipulation. This separation aids modularization, reusability and extensibility as we abstract from the implementation and design of a participant: we can rely on a small set of already defined lower-level properties when providing support for different higher-level properties: instead of hard-wiring the manipulation of higher-level properties that implicitly entails manipulating multiple lower-level properties, we orchestrate the manipulation of lower-level properties that result in the desired change in the higher-level property.

This modular design is inspired by the concept of service-encapsulation and reuse in Service-oriented Architecture (SOA) [169]. In order to apply this approach to EBS, we *a*) identify objectively measureable properties over which subscribers can define comparable QoI requirements; *b*) deduce properties that influence those QoI properties relevant to subscribers in an EBS; and *c*) identify the type of participant supporting each property, i.e., MOM, publishers or both.

In literature, a multitude of different properties is proposed and discussed in context of the different concepts that all address runtime quality (c.f., Section 3.1). Unfortunately, they are overlapping and sometimes contradicting in their semantics, granularity, or representation as the different concepts are blending into each other.

We derive a generic format to represent properties by successively applying five filters to the properties associated with runtime quality in literature as shown in Figure 3.5.

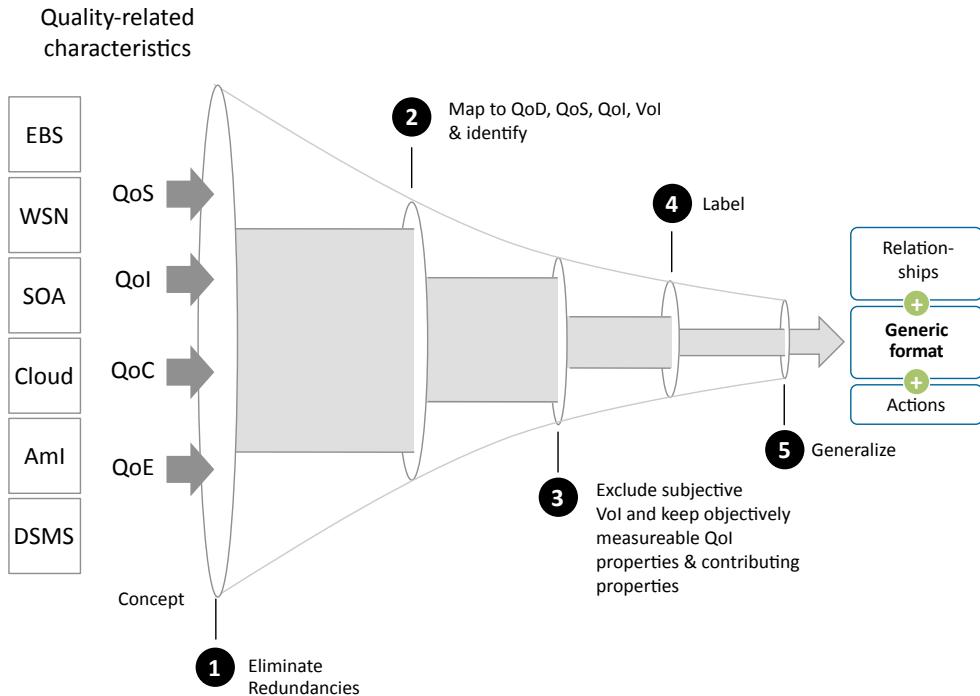


Figure 3.5.: Steps and filters applied to properties discussed in literature to derive a generic property format to express the basic building blocks of our approach.

Step 1 Eliminate synonyms. Ambiguous and synonym terms used to describe the same semantics are harmonized. For example, both *tunability* [366] and *ease of manipulation* [37] are used to describe the effort necessary to modify information, while both *confidence of detection* [226] and *probability of correctness* [261] measure how dependable the information is, i.e., the conviction of the publisher that the described event has actually happened; *delay* and *latency* are used interchangeably to describe the time elapsed between sending data at one entity and receiving it at another — we use the terms *ease of manipulation*, *confidence of detection* and *latency*. In general, we stick to the terms used by the majority of papers.

Step 2 Map to QoD, QoS, QoI, or VoI to identify dependencies and ownership. Classifying properties into the levels of abstraction and accumulation discussed earlier helps to identify dependencies between properties and the types of participants that determine and control those properties. For example, resolution (QoD) is manipulated by publishers.

Step 3 Reduce set to objectively measurable QoI properties. Using a top-down approach, we identify objectively measurable properties that are used by subscribers to describe data quality in EBS in a first step and filter out subjective VoI properties. In a second step, we identify the contributing QoS and QoD properties that influence each identified QoI property. They do not necessarily have to be exposed to the subscriber but they require publishers and MOM to model capabilities about them.

Step 4 Label. The remaining properties are labeled by the data types and scales of measurement they are represented by in literature [400].

Step 5 Generalize representations. We derive a generalized representation to model requirements about and support for QoI, QoS, and QoD properties.

The following Section 3.2.1 describes the results of our analysis while Section 3.2.2 presents the derived generic format in more detail. In particular we introduce the different types of actions we associate with a generic property, and discuss the types of relationships between properties and approaches to represent their mapping.

3.2.1 Categorization of Properties

In the remainder of this section, we identify the common features of properties about runtime quality and generalize them into a generic format. Throughout this work, we use this generic representation to define requirements and capabilities. We revise the properties discussed in more than 70 peer-reviewed publications that address runtime quality issues in EBSs, Wireless Sensor Networks (WSNs), Cyber-physical Systems (CPSs), Data Stream Management Systems (DSMSs), SOAs and Cloud computing as well as in AmI, HCC and HCI. We started with surveys as well as papers with high citation counts that explicitly address QoD, QoS, QoI, QoC, or QoE and proceeded with cross-references.

We map the surveyed properties to the four levels of abstraction QoD, QoS, QoI, and VoI we have discussed in Section 3.1. Condensing the different concepts into these four categories allows us to associate each concept with the type of participant that determines and manipulates each class of properties in an EBS: publishers control properties associated with QoD, while the MOM controls most properties associated with QoS. Properties associated with QoI depend either directly on publishers, the MOM, or on a cooperation between both. Only the subscriber can determine properties classified as VoI. Here, VoI encompasses properties and functions from the areas of QoC and QoE as these are application- or domain-specific accumulations. A detailed list of the reviewed references and their mapping to the discussed properties can be found in the Appendix in Table A.1 (QoD), Table A.2 (QoS), Table A.3 (QoI), and Table A.4 (VoI).

The results of both mapping properties to the different categories and identifying dependencies are shown in Figure 3.6 as stacked swim lanes. Each swim lane comprises the properties mapped to a specific category. The swim lanes are stacked to show increasing levels of abstraction. Associated properties are shown as blocks while arrows between them denote relationships. Please note that we do not show every possible relationship between properties but concentrate on the most obvious ones for clarity. VoI properties marked red are out of scope of this dissertation while properties marked green or gray are in-scope. Properties marked gray denote properties that influence QoI properties and have to be modeled for publishers and MOM to define capabilities but are subordinate from the perspective of a subscriber.

At the bottom of Figure 3.6, properties associated with QoD are controlled by publishers. Most of these properties are influencing QoI properties but do not have to be exposed to the subscriber as long as the depending QoI properties are exposed. QoS properties are controlled by the MOM. As for most of the QoD properties, they do not necessarily have to be exposed to the subscribers as long as the depending QoI properties are exposed. The upmost green swim lane encapsulates those QoI properties that are neither device- nor system specific but describe quality-related properties of notifications and the data enclosed therein. QoI properties, however, depend on

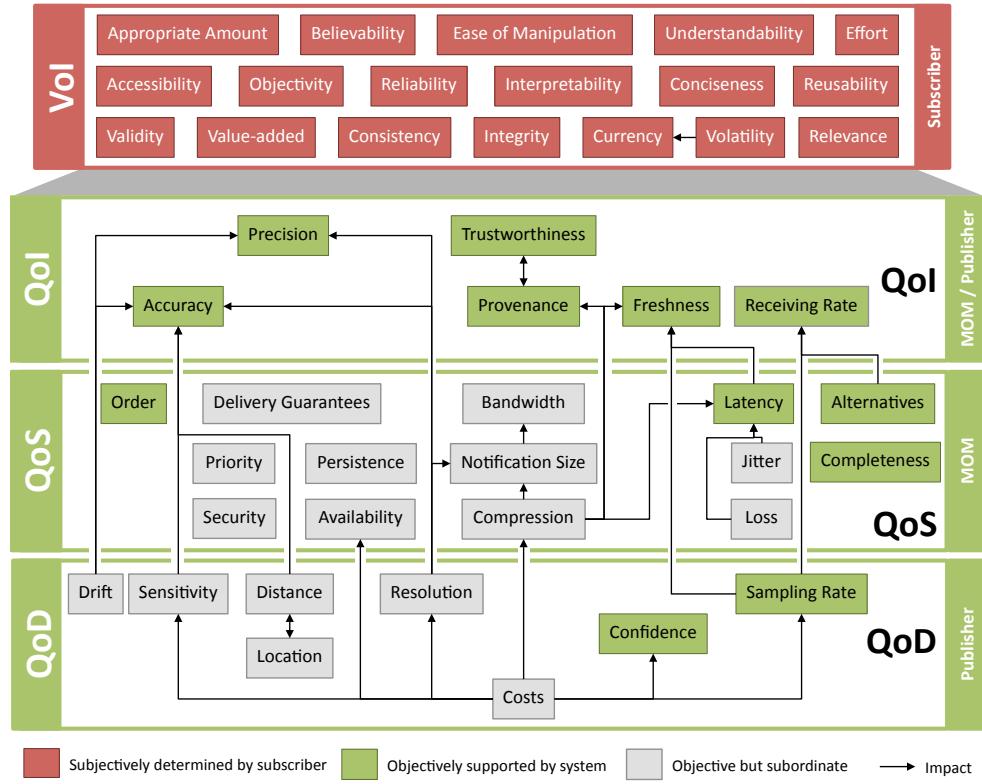


Figure 3.6.: Mapping quality-related properties commonly referred to in literature to the objective concepts of QoD, QoS, QoI (in scope, green) and the subjective concept of VoI (out of scope, red). For improved readability, not all relationships are shown.

QoS or QoD properties controlled by publishers or the MOM as shown by various relationships across level of abstraction. The red swim lane shown on top of Figure 3.6 contains all properties associated with VoI. They are out of scope of this dissertation as they are subjective interpretations and accumulations of the objective lower-level properties and require a specific application scenario to be valid. They are individually determined by each subscriber and do not have to be exposed by the system except for domain-specific solutions and deployments.

Analyzing the properties discussed in literature and eliminating redundant terms results in a total but not exhaustive set of 47 properties. In the remainder of this section, we discuss the semantics of each property as well as their representation. We describe each property, starting bottom-up with properties associated with QoD and controlled by the publisher. We discuss explicit dependencies and ambiguous use of names as we move up the levels of abstraction to the subjective VoI functions shown in red on top of Figure 3.6.

Please note that some properties can be defined on different levels of abstraction. *Completeness*, for example, can be defined for packages on the transport protocol level (QoS), or for notifications (QoI) on a higher level of abstraction; *compression* can be associated with a device (QoD) as well as with the system (QoS). When not explicitly stated otherwise, we define each property on the highest possible level of abstraction as our approach aims at providing support for properties that are used by subscribers to define QoI. From the perspective of a subscriber, properties defined on lower levels of abstraction become subordinate to those properties defined on a higher level of abstraction that is of greater interest to the subscriber.

Quality of Device (QoD) Properties

Properties associated with QoD describe characteristics of a publisher, which could be a physical or digital device. They are determined by the publisher and depend on the configuration as well as on the capabilities of each instance. Some properties can be changed at runtime but require the cooperation of the publisher if the MOM wants them to change.

Cost

Sensing events, fusing lower-level data for reasoning, purging and finally publishing notifications comes at a cost for publishers, specified by this property [37]. Costs can be based on energy consumption [54, 354, 356, 386], network costs [173] or total costs for invoking contributing services or applying different techniques, e.g., for compression or fusion [46, 356, 384]. Represented by cost per unit in either abstract [6, 46, 143, 356] or domain-specific form such as energy-consumption per notification [354].

Distance

For sensors observing entities in the physical world, this property expresses how far away the sensor is from the entity it is monitoring or the event it has detected. Represented by a distance metric such as meters [75]. Ideally, the distance of the sensor to the monitored object is as small as possible.

Drift

Denotes a constant or varying factor that measurements become biased by over a certain amount of time [108, 418], represented by its magnitude [422].

Location

Describes where the publisher is located and can refer to real-world or abstract locations in absolute or relative terms [109, 208]. Representation can be pair-wise (e.g., GPS coordinates) or key-value-pairs (e.g., latitude and longitude or degrees, minutes, seconds) [55, 62]. In an EBS, *location* information is considered to be part of the content of a message and not quality-related metadata [22].

Resolution

The granularity of information in terms of time, space, or domain-specific parameters [57, 75, 109, 418]. Temporal resolution refers to the period of time that a measurement is associated with (e.g., a measurement duration of 10 minutes vs. 1 second) while spatial resolution denotes the physical area that a measurement is associated with (e.g., a sensing area of 2 square meter vs. 10 square miles). The resolution of published data depends not only on the capabilities of the publisher but also on the costs for sensing and processing; it has a significant impact on the size of a notification [57]. Represented by domain-specific metrics, e.g., 0.5 megapixels for images, or National Image Interpretability Rating Scales (NIIRS) score of 1 for the interpretability of aerial photographs [352]. In general, a higher level of detail is regarded as positive. In regard to temporal and spatial resolution, this might result in a small value that describes a maximum closeness in terms of time or space; in terms of image resolution, a high value is usually considered to be better than a low value as this represents a higher level of detail that can be captured on the picture.

Sampling rate

The frequency a sensor's measurements are published with as notifications [62, 172, 340, 415]. Represented by the number of notifications per time unit.

Sensitivity

Describes how sensitive a sensor reacts to a change in the value that it observes. Please note that resolution and sensitivity might correlate but are not interdependent: while a high sensitivity might result in a higher resolution, the former describes the maximum granularity of the input data of a sensor while the latter describes the granularity of the published data. The sensitivity of a publisher depends on its capabilities but also on the costs for sensing. Sensitivity is represented by the ratio between the observed change in a value and the change actually taking place, e.g., $value_{before} = 200, value_{changed} = 100, value_{sensed} = 110 \Rightarrow 90/100 = 90\%$ sensitivity [386, 418]. A high sensitivity indicates that small magnitudes of change can be observed.

Quality of Service (QoS) Properties

Properties associated with QoS describe characteristics of the system and include aspects of transportation and interaction between entities over network links. QoS properties are primarily determined by the MOM and can be manipulated to a certain degree at runtime.

Alternatives

Defines the number of different data sources that must provide information with the same set of quality-related properties. In collaborative environments, a certain number of alternative sensors is crucial to fulfill a sensing task or to improve the sensing results [35, 66, 109, 339, 386, 415]; users may also directly require a number of data sources that all satisfy certain quality criteria [340]. Using data from multiple sources is mandatory in general to double check events (e.g., to verify the sighting of a certain RFID tag by receiving a sighting event from at least two separate readers) and for fault tolerance [41]. Alternatives can be represented by an absolute number or by the percentage of data sources that should be selected out of all available data sources.

Availability

Quantifies the degree to which a system or service is working correctly over time according to the requirements. For pull-based systems, this can be the percentage of times a service responds within a given time over a given period, e.g., an availability of 0.9305 or 93% [357]. For push-based systems, the availability of a publisher could be defined as the percentage of time that the publisher provides notifications at the given rate [271].

Bandwidth

Low-level metric applied on the transport protocol level [263]. Capacity of a communication link between entities. Represented by the amount of traffic being processed per time unit as bits per second or number of homogenous messages per second [159]. Sometimes also expressed as the utilization of the available bandwidth in percentage [122].

Completeness

Describes the degree to which all necessary parts of the information are available for the subscriber [182, 271]. For the area of EBS, completeness has two complementary aspects: the completeness of a single notification and the completeness of a sequence of notifications. For a single notification, completeness is represented by the ratio of the number of attributes contained in the notification compared to the total number of attributes of a reference type [254, 271]; these attributes can also be weighted to reflect varying importance [291]. For a sequence of notifications, completeness quantifies the percentage of

notifications that have been dropped by the MOM, i.e., notifications that have been published but not delivered to a matching subscriber in a given period of time [42, 95, 208]. Completeness is represented as a fraction in both cases [6, 292].

Compression

Denotes whether a notification has been altered to reduce the amount of data to process or transmit. Compression refers to removing parts of the data's representation that are deemed expendable or to aggregating the content of different data items (e.g., averaging, summarizing). There are lossy and lossless types of compression. Lossless compression (e.g., GZIP) is reversible as it allows a complete reconstruction of the original granularity of the data while lossy compression is nonreversible [356]. Compression can be represented either as a binary value (compression available/applied), the number of data items being merged, or by the degree to which the size of a notification has been reduced or by a set of labels that denote the applied compression mechanisms ranked by their efficiency [263].

Delivery guarantees

Subscribers can define how often they want to receive a single notification that is published by a publisher and processed by the MOM [208]. Depending on the negotiated guarantee, the MOM has to ensure that either no message is lost or that no duplicates are generated. Possible values are *best effort* (a notification might not be received at all or multiple instances could be received), *at least once* (guaranteed delivery of either one or more instances of the same notification), *at most once* (either none or exactly one instance of a notification is processed) or *exactly once*, where the last one is the most challenging one to guarantee as it combines the semantics of *at least once* and *at most once* [42, 47].

Jitter

Low-level metric applied on the transport protocol level [263]. Quantifies the variation in the delay of packages being processed on the network. Represented by the standard deviation of the latency of network packets [134, 210]. As notifications are split up into several packages on the transport protocol level, a high jitter can increase the latency of the notification as transport protocols have to re-send those parts of the notification that are wrongly assumed to be lost before the notification can be reassembled and processed by the MOM or subscribers.

Latency

Quantifies the time elapsed between two actions. Here it denotes the time (minutes, seconds, milliseconds, nanoseconds) between sending a notification at one entity and receiving it at another. From the perspective of a subscriber, latency denotes end-to-end latency, i.e., the overall time elapsed between the notification is sent by the publisher and received by the subscriber. From the perspective of the MOM, latency is the sum of *publication latency* and *processing latency*. Processing latency is affected by jitter and loss [210] while both publication and processing latency are also affected by the overall performance of publisher and MOM. Thus, applying compression to notifications can also result in higher latency as compression is resource-intensive or notifications are not forwarded at all as they are aggregated [356].

Loss

Low-level metric applied on the transport protocol level [263]. Refers to the percentage of packages being lost on the transport protocol level, forcing a re-submit when using dedicated transport protocols [210]. As with jitter, a high degree of loss can increase the

latency of the notification as notifications are split up into several packages on the transport protocol level and all have to be received before the notification can be reassembled and processed.

Notification size

The size of a single notification in terms of bytes or attributes. The size of a notification is affected by the kind of compression used as well as the resolution of the information and thus influences the consumed bandwidth [31, 85]. The notification size depends on the resolution, completeness and representation of the notification.

Order

Denotes the order that notifications arrive in at the subscriber. Represented either by binary states *random/unordered* and *total order* [288], or by different values that denote guarantees with increasing strictness: *no order*, *publisher First In – First Out (FIFO) order* (i.e., arrive in the order they have been published by each publisher), *causal order*, and *total order*; the latter two guarantees require synchronization when multiple publishers for notifications about the same type of event are involved [47].

Persistence

Describes a feature of the MOM that allows notifications to be stored for a certain amount of time in the system. Persistence is used to support mobility and periods of disconnection, provide delivery guarantees through re-transmission of notifications, or allow subscribers to receive historic notifications that have been published before the subscriber had joined the system [112].

Priority

Indicator for the importance of a certain type of notification and represented by a score or ranked set of labels assigned by subscribers or publishers [7, 47, 112]. The MOM can ranking notification types by their priority to decide on which notifications to process preferentially [47].

Receiving Rate

Defines the number of notifications per time unit received at the subscriber. Especially resource-restricted subscribers benefit from having control over the maximum number of notifications per time unit they have to process [277]. Requirements about the received rate can be decomposed into requirements about the maximum sampling rate to be consumed from a number of alternative publishers. For example, a maximum received rate of 100 notifications per second can be modeled as a requirement about a maximum of 20 events per second (*sampling rate*) from a maximum of five different publishers (*alternatives*).

Response Time

Quantifies the time between a request and a response [418]. Usually defined for pull-based systems and represented like latency; an unusual interpretation of publishers' sampling rates for push-based systems to express the speed of detecting and reporting an event [418].

Security

Covers aspects from encryption to access control, identity management, and authentication. Can be represented either by binary values to denote that data is encrypted or not to a set of labels denoting increasing levels of security [6, 113, 263].

Objective Quality of Information (QoI) Properties

Properties related to QoI describe characteristics of the data being produced by publishers and processed by the MOM. The properties discussed here can be measured objectively as they are defined independently of the requirements of a specific subscriber. QoI properties are determined by publishers or by the MOM. Manipulating them at runtime might require a coordination between MOM and publishers.

Accuracy

Quantifies how well the data enclosed in a notification reflects the fact or event it describes [53, 95, 418], measured by its deviation from the true value. Accuracy is hard to determine in reality, as it requires knowledge about the true value for each measurement [306]. For sensor readings, this can be compensated by calibration and the use of reference values [324]. Thus, the accuracy of a measurement depends on the *sensitivity* of the sensor as well as on its *drift* and *resolution*. Depending on the kind of sensor and the sensing task, *distance* and *location* as well as the current context of a sensor do also influence the accuracy of measurements [109, 110, 182]. Accuracy is quantified in domain-specific ways: the ratio between false positives or false negatives and the total number of notifications describing an event [95, 226]; the root square mean deviation for the estimation error of a prediction model [99, 254]; the percentage of deviance from the true value, e.g., $\pm 0.3\text{m/s}$ or $\pm 2\%$ for a wind sensor [110]. These domain-specific representations can be normalized into a score in the interval [0;1] where 1 denotes perfect conformity with the described fact [261].

Confidence of detection

Quantifies the conviction of a publisher that the reported event has been captured correctly [54, 75, 224]. Also referred to as *probability of correctness* [261, 291] or *coverage probability* denoting that the true value is covered [53]. High confidence indicates less false positives and false negatives of the reported events [96, 430]. With different connotations more related to trust [6] or precision [182] when referring to multiple records of the same information to be the same across time. Represented by a score in the interval [0;1] where 1 denotes absolute conviction.

Freshness

Quantifies the actuality of the information contained in the event notification. Freshness is the sum of two distinct aspects of timeliness: (a) timeliness of capturing the event including details of the capturing frequency as expressed by the sampling rate of the sensor [31, 47, 51, 57, 250]; and (b) timeliness of processing the event notification from publisher to subscriber which entails processing latency and transport latency [6, 248, 271]. Sometimes also referred to as *up-to-dateness* [6, 75, 292]. Represented like latency in units of time. Lower sampling rate, higher compression and content aggregation in particular decrease freshness as they increase latency [356].

Precision

Quantifies the variation of replicate measurements in describing the same true value, measured as imprecision. Precision complements the notion of accuracy as it describes a different aspect of measurement deviation [53, 306, 324]. Using the graphical analogy of shots fired at a target [324], accuracy describes the distance of each hit to the bull's eye while precision describes the degree to which the hits are clustered on the target. In contrast to accuracy, precision is easier to compute for numerical values by the use of statistics,

i.e., normalized standard deviation. Represented as a ratio with 100% referring to perfect alignment [324]. We try to illustrate the relationship between precision and accuracy by returning to the example of a temperature sensor as given in Section 3.1: a temperature sensor measures a room's temperature of around 20 degrees Celsius with a bias of $\pm 0,4$ degrees Celsius – or 98% *precision* – while another temperature sensor has a bias of ± 3 degree or 85% precision [75]. In both cases, the precision denotes the *average* deviation around the true value for each recurring measurement. The *accuracy* of a single measurement, however, could be much better or worse. In the area of IoT systems, *precision* is increasingly used to describe the semantics of accuracy as the true value itself is seldom known but is reconstructed from different measurements that are not exactly repeatable [53, 54, 75, 182, 324].

Provenance

Metadata describing the pedigree and level of originality of data. The intention of provenance is to keep track of where data has originated from, how it has been generated and how it has been altered before being delivered. Thus, source provenance and process provenance can be distinguished. Provenance is important for auditing, attribution, or replication of data [35, 385]. Metadata describing provenance can be attached to the data item itself or be stored in a provenance store to be queried [191, 314]. In database systems, provenance information can be provided by reverting the queries that produced the current data. Provenance is quantitatively or qualitatively represented based on the domain-specific requirements about the granularity and extent that data should be traceable [314]. The spectrum of quantitative representation ranges from a Boolean value to denote whether data has been altered at all [225], to a cosine distance in the interval $[0; 1]$ between reference data and the received copy to quantify the degree to which data has been altered (with 1 denoting a maximal similarity while 0 describes two completely different data items) [224]. Representing provenance qualitatively ranges from structured metadata in XML [191, 385] to ontologies using Resource Description Framework (RDF) and Web Ontology Language (OWL) such as the W3C PROV family of ontologies [312]. Provenance is not only affected by the fusion done by CEP engines but also by notifications being compressed or aggregated while processed by the MOM.

Trustworthiness

Indicator for the degree to which the consumer can rely on the data being correct and not being manipulated on purpose by the producer or the MOM [182]. Thus, trustworthiness affects both the publisher and the data it provides: it is either estimated by the MOM for a publisher and then vested on the data provided by this publisher, or the trustworthiness of different data items is evaluated and an accumulated degree of trustworthiness applied to the publisher providing the data [127, 274]. Trustworthiness is closely related to provenance and security as both properties help to address aspects of data deception, i.e., malicious parties provide false data on purpose [275]. Some approaches for assessing the trustworthiness of both data and publisher depend on source provenance and process provenance [127, 274] while others use feedback provided by subscribers to assess the trustworthiness of a publisher [46, 47, 324]. Trustworthiness is quantitatively represented in the interval $[0; 1]$ by numeric trust scores [5, 275, 292] or by ranked trust labels [6, 57, 75, 109, 188, 324, 384]. Higher trust scores or trust levels indicate a higher degree of trustworthiness [6, 75, 109, 188, 324, 384].

Subjective Value of Information (VoI) Properties

Properties grouped here all depend on objective properties discussed before. However, they aggregate and accumulate objective properties based on domain- or application-specific definitions. Thus, we denote those accumulations as subjective VoI properties or functions that do not necessarily have to be provided by the system except for domain-specific deployments as they cannot be evaluated independently of the subscriber. We exemplarily discuss *volatility* and *believability* in more detail to show the general characteristics.

Volatility

Describes the probability that an updated version of data contained in a notification becomes available before the current data item expires (i.e., currency = 0). Defining volatility requires specific knowledge about the dynamic behavior of updates for this type of data and its general *freshness* [271].

Believability

A quantification for measuring the resilience of data from the subscriber's point of view. Depends on domain-specific knowledge about the data, the subscriber's configuration and the requirements of the concrete task that the data is intended to be used for. Like *accessability*, *appropriate amount*, *integrity*, *interpretability*, *ease-of-manipulation*, *objectivity*, *understandability*, *validity*, *value-added*, or *relevance*, the notion of *believability* revolves around the effort necessary for subscribers to assess, detect, purge, repair or replace data of insufficient quality according to their needs [143, 250, 292, 366, 421].

The discussed examples show that the VoI properties grouped here rely on individual combinations of requirements about QoI properties such as *accuracy*, *precision*, *confidence of detection*, *provenance*, or *trustworthiness* [47, 224, 225, 244, 247, 248, 366].

3.2.2 Deriving a Generic Property Representation

Revising the properties discussed in this chapter shows that although different properties do not have to be comparable (e.g., *trustworthiness* vs. *sampling rate*) they can be conflicting when supported due to interdependencies and resource constraints (e.g., trading-off *accuracy* vs. *latency* for energy-efficiency as described in [99]). Some properties are interdependent across levels of abstraction (e.g., *freshness*, *latency*, and *sampling rate*) while others are orthogonal (e.g., *spatial resolution*, *loss*).

Regarding their representation, however, they all share two features that we can exploit to derive a generic format for representing them: (a) they can all be modeled over a range or a list of values that apply a total order depending on the semantics of the property, and (b) they can all be improved by either maximizing or minimizing them based on this ranking.

Range or list of values

Having categorized the properties already by their level of abstraction, we can group them by the data types they are represented by: binary variables, list of String values, integers, or floating point notations (double, float). Examples are: integers for the number of alternatives; floating point notations for latency, accuracy, or precision; ranked labels for delivery semantics where the labels are ordered from least strict (*best effort*) to the strictest guarantees (*exactly once*) – same

for order, trustworthiness or compression while the latter can alternatively be represented by binary outcomes (compression or no compression).

We can further generalize these different data types by associating them with different scales of measurement. Each scale has different features that reflect characteristics of the objects categorized on this scale. Examples for those scales of measurement are the four types of scales *nominal*, *ordinal*, *interval*, and *ratio* that have been introduced by Stevens [400, 322]:

Nominal

Values on a nominal scale represent items in a set. The only operator applicable to these values is the equality operator to check if two values are the same or not. Based on this, the cardinality of a set can be computed but values cannot be ranked as they represent only labels or type numbers that are equally potent. Examples are taxonomies in biology, or back numbers of football players.

Ordinal

Objects represented as values on the ordinal scale can be ranked in a relative order according to their semantics in addition to the equality operation. Examples are values such as "low" < "medium" < "high". All operations that are order preserving are applicable to values on the ordinal scale while statistical operations such as computing means or standard deviations are not applicable as there is no interpretable distance between two values but only a relative rank-order. This is due to only a relative minimal value (the value with the lowest rank) being available but no absolute zero value.

Interval

Like values on the ordinal scale, values on the interval scale are ranked. Examples are scales of temperature ($^{\circ}\text{C}$ and Fahrenheit) or scales of time. In contrast to the ordinal scale, ranking two values on the interval scale allows to interpret the distance between these values but not their ratio, e.g., March 20th is not twice March 10th, but a period of 10 days is twice as long as a period of 5 days.

Ratio

The ratio scale has no restrictions regarding operations. Values can be ranked, absolute distances can be computed and ratios can be interpreted because ratio scales have absolute zero values. Examples for ratio scales are the numeric scales themselves.

Out of these four scales, all discussed properties with their different data types and semantics map to either ordinal, interval or ratio scales as shown in Table 3.1. Please note that this mapping is done based on the semantics of each property, which allow us to define a ranking for all its defined states.

Using those semantics, we can even map binary data types to an ordinal scale. For example, representing the property *completeness* by binary values, we can define that the binary outcome "true" (complete data) is more desirable for this property than the binary outcome "false" (incomplete data); in fact, we are mapping binary values "false"/"true" of a nominal scale to the ranked labels "bad" < "good" of an ordinal scale.

Finally, we can represent these three scales either by a *list of values* (ordinal scale) or by a *range of values* (interval, ratio) as shown in the topmost row in Table 3.1.

Table 3.1.: Scales of measurement used to represent different properties about runtime quality.

Property	List		Range				Improvement	
	Ordinal		Interval		Ratio		Max	Min
	Binary	Labels	Integer	Double	Integer	Double		
Accuracy	□	□	□	□	■	■	↗	
Alternatives	□	□	■	□	■	□	↗	
Availability	□	□	□	□	□	■	↗	
Bandwidth	□	□	□	□	□	■	↗	
Completeness	■	□	□	□	■	■	↗	
Compression	■	■	□	■	■	□		↙
Confidence	□	■	□	□	■	■	↗	
Costs	□	□	□	□	□	■		↙
Delivery Guarantees	□	■	□	□	□	□	↗	
Distance	□	□	□	□	□	■		↙
Drift	□	□	□	□	□	■		↙
Freshness	□	□	■	■	□	□	↗	
Jitter	□	□	□	□	□	■		↙
Latency	□	□	■	■	□	□		↙
Loss	□	□	□	□	□	■		↙
Notification size	□	□	□	□	□	■		↙
Order	■	■	□	□	□	□	↗	
Precision	□	□	□	□	□	■	↗	
Provenance	■	■	□	□	□	■	↗	
Resolution	□	■	■	■	□	□	↗	
Sampling Rate	□	□	□	□	■	□	↗	
Security	■	■	□	□	□	□	↗	
Sensitivity	□	■	□	□	■	■	↗	
Trustworthiness	■	■	■	■	■	■	↗	

Improvement Direction

Knowing the semantics of a property, we can decide if a specific state is generally more desirable than another. For example, the label "known device" for *trustworthiness* (alternatively represented by a trust score of 100%) denotes a higher degree of trustworthiness than the label "unknown device" (a trust score of 0%, respectively) [6]. On the other hand, a *latency* of 500ms is considered to be better than a latency of 2000ms.

Generally speaking, maximizing or minimizing it depending on the semantics of the property can improve each property. This is of particular importance for properties that can be represented in multiple ways with differing semantics. The property *resolution*, for example, is maximizing if modeled in terms of image resolution or the number of attributes describing a complex fact, while it is minimizing if modeled in terms of temporal or spatial resolution. For properties with ranked labels such as trustworthiness labels (e.g., "untrusted" < "erratic" < "trusted" < "authoritative"), a stricter value improves a relaxed one, e.g., "randomOrder" < "totalOrder".

As all properties can be represented on a scale that supports at least a ranking between different values of the same property, we can assign each property with an improvement direction as shown in the last two columns of Table 3.1.

Definition of a Generic Property

As illustrated in Figure 3.7, we can formalize the definition of a generic property as follows:

Definition 1 (Generic property). A generic property p describes a characteristic of a notification, a publisher, or the MOM in an EBS. A property p_k is a tuple $(p_{name}, \tau, \sqcup, \text{DIRECTION})$ with τ the data type, and DIRECTION being the improvement direction indicating maximization (\nearrow) or minimization (\swarrow). Different embodiments of this characteristic are defined as a list or range of values $\sqcup = \{v_k \in \tau \mid v_{min} < v_k < v_{max}\}$ on at least an ordinal scale. The values $v_k \in \sqcup$ for p_{name} are ranked in the order they are defined in \sqcup ranging from v_{min} to v_{max} and reflecting their semantics. A value v_k dominates a value v_j iff $v_j < v_k$ and $v_k > v_j$. \square

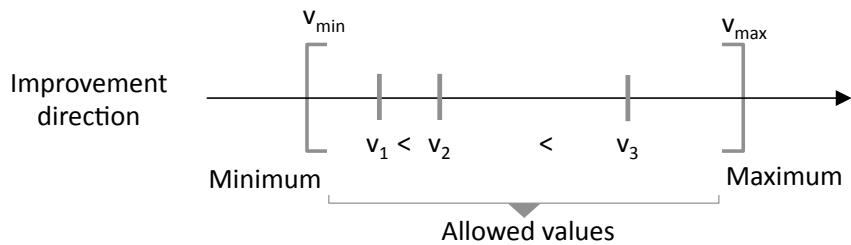


Figure 3.7.: Generic property: range/list of allowed values, ranking and improvement direction.

Expressing generic properties in software systems can be done using Extensible Markup Language (XML) or a line-based syntax. The Backus-Naur Form (BNF) notation for a line-based syntax based on the above definition is shown in Listing A.8a while Listing A.8b shows an example for *confidence of detection*, *latency*, *sampling rate*, and *trustworthiness* using this syntax.

Actions to Manipulate Generic Properties

Each property can be manipulated by a publisher, the MOM, or in coordination between both. For example, the *sampling rate* of a publisher can be increased or decreased at runtime. However, only the publisher can *increase* the sampling rate while *decreasing* can be done either by the publisher or by the MOM applying a filter before delivering notifications to the subscriber [173, 356]; *confidence of detection* and *precision* can be increased or decreased only by the publisher by changing the sensor frequency [62] or applying additional cross-validation using multiple sensors which increases communication and processing costs [225]. The MOM can decrease *latency* to a certain degree by using traffic-reduction mechanisms based on prioritization and filtering [138]. These examples show that applying a mechanism comes at a cost for the executing participant, depending on its capabilities, context and state [173, 225, 356].

In an EBS, these mechanisms and their owners are hidden from the subscriber, as the MOM is handling requirements by subscribers about properties. As satisfying those requirements might require increasing or decreasing properties, the MOM has to know whether a property can be manipulated in either direction. The details of the mechanisms, however, do not have to be known at this step while the MOM has to be aware of the costs for executing a mechanism. We encapsulate these aspects into an action to be associated with a generic property in the MOM:

Definition 2 (Action). Actions encapsulate dedicated activities to manipulate a property p_e . An action a_{ID} is a tuple $(p_e, \text{DIRECTION}, ID, costs_{ID})$ with p_e being the associated property, DIRECTION denoting whether this action increases (\uparrow) or decreases (\downarrow) the property, ID as an identifier for this tuple and the name of the action, while $costs_{ID}$ quantify the costs arising from applying ID. \square

From the perspective of the MOM, an action can be invoked by *self-adaptation* or *delegation* depending on who is responsible for executing the necessary mechanisms. Mechanisms applied by the MOM are executed by self-adaptation while delegation is used for those mechanisms that are applied by publishers⁴. Delegation is done by triggering the respective participant to manipulate a specific property to the desired degree. In both cases, the details of how the property is actually manipulated do not have to be defined inside the action.

Please note that sequences of actions can be defined as a new action as well. Alternative actions can be defined by associating multiple tuples for a property. They can have different costs but must have different IDs.

For example, we can model the two alternatives for decreasing the *sampling rate* of a publisher in our concept by associating two tuples given as follows:

$$a_{adaptPublisher} = (rate, \downarrow, adaptPublisher, costs_{adaptPublisher}) \quad (3.1)$$

$$a_{applyFilter} = (rate, \downarrow, applyFilter, costs_{applyFilter}) \quad (3.2)$$

Relationships Between Generic Properties

Relationships between generic properties are explicitly modeled inside the MOM and hidden from the subscriber. They are necessary for the MOM to determine the support of the system for a given property p_a about which subscribers have defined requirements. Relationships can be modeled explicitly to determine if and how a given generic property is supported by the system.

Relationships allow to determine if the property p_a addressed by the subscriber is directly supported by capabilities and can be manipulated using an associated action or whether actions and capabilities are defined for a generic property p_b that matches p_a in syntax (i.e., granularity, scale, improvement direction and data type) and semantics. This is important in heterogeneous environments in particular [165, 166, 168].

Furthermore, relationships identify the properties that p_a depends on from the perspective of the MOM and how these properties influence p_a . From the subscriber's point of view, for example, increasing *publication latency* and *processing latency* deteriorates (end-to-end) *latency* while increasing the sampling rate improves (end-to-end) *latency* from the MOM's perspective. This knowledge is required by the MOM to evaluate or manipulate different lower-level properties when negotiating requirements.

The relationships between p_a and p_b can range from simple to complex as illustrated in Figure 3.8. Exemplarily, we have defined four types of relationships with increasing complexity: *identity*, *mapping or transformation*, *graph*, and *ontology*.

⁴ In a distributed MOM consisting of several interconnected brokers, neighboring brokers are treated like publishers when notified by the edge broker handling requirements of a connected subscriber.

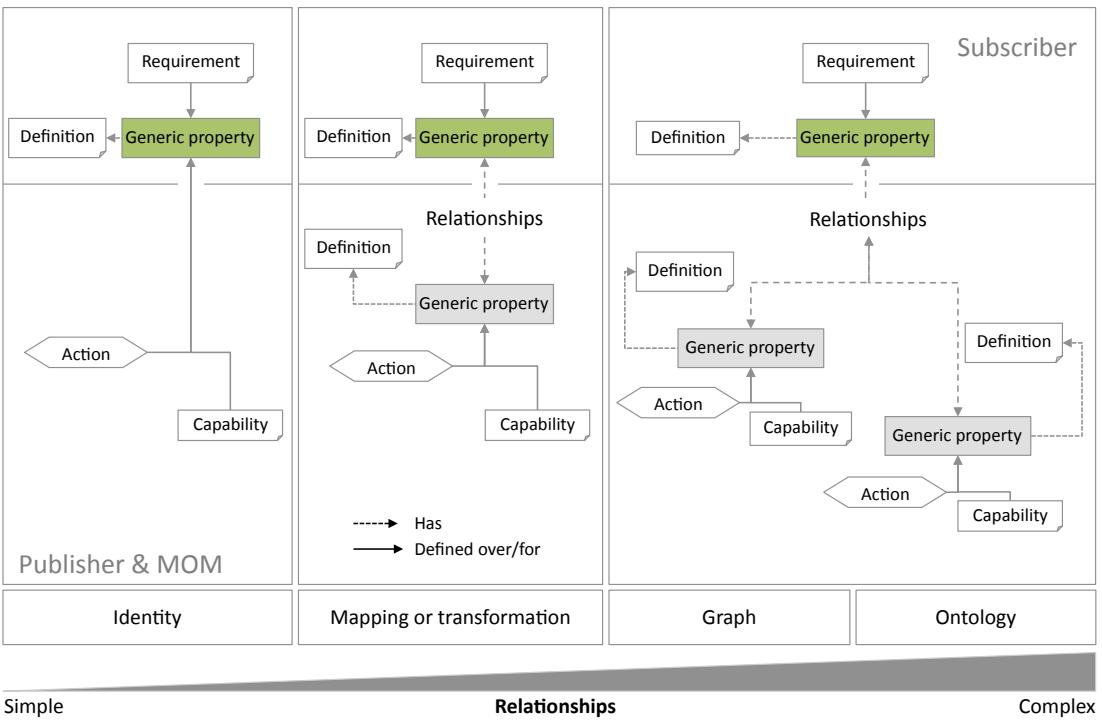


Figure 3.8.: Types of relationships between generic properties in an EBS.

Identity. In this case, the property p_a that a subscriber has defined requirements over is identical to the property p_b as supported by the system in that capabilities and actions are defined for it (c.f. Figure 3.8, left). Examples are *accuracy*, *precision*, or *confidence of detection*.

Mapping or transformation. A relationship requiring a mapping or a transformation exists if p_b has the same semantics as p_a but a different syntax or name. In this case, p_b has to be mapped to p_a using transformations [166] if necessary (c.f. Figure 3.8, middle). For example, a subscriber has defined a requirement about the generic property *trustworthiness* which is defined over the trust scores $0 < 0.5 < 1$ [6] while the system supports the generic property *trust* defined over the trust levels "untrustworthy" < "trustworthy" < "very trustworthy" [324].

Graph. A graph captures a relationship with multiple generic properties contributing to a property the subscriber has defined requirements about, e.g., *freshness* depends on *processing latency*, *publication latency* and *sampling rate* controlled by the MOM and publishers.

Ontology. More complex relationships may require the use of a suitable ontology as semantical transformations and performance models have to be modeled as an extension to the graph-based relationship. For example, the ontology described in [418] allows to describe parameters that influence the performance of a sensor regarding different properties and to express how these properties are affected by different contexts and conditions at runtime [109].

We do not make any assumptions about the complexity or representation of relationships between properties as we encapsulate both within the mapping function Θ . A mapping function Θ encapsulates the relationships between a subscriber-side property p_a and generic properties that are supported by the system. $\Theta(p_a)$ returns a generic property p'_a that matches p_a in syntax and semantics.

3.3 Expectations: Requirements About QoI Properties

In an EBS, subscribers require data not only of a certain type or content but also of adequate Quality of Information (QoI) to perform their individual tasks. They have individual preferences about quality-related properties such as precision, sampling rate, latency, or confidence of detection based on their subjective VoI functions that are determined by the application logic and context at runtime [22, 35, 56, 182, 340, 381].

For example, an application M has to monitor the temperature of a chemical process during manufacturing to detect anomalies. Information about the current temperature has to be either highly reliable to minimize false-positives/negatives or has to be very fresh to minimize the impact of false-positives/negatives on the evaluation result at subscriber M . In terms of quality-related properties, M requires notifications about events of type `temperatureEvent` to be delivered with high confidence of detection (e.g., 75-95%) at a low sampling rate (e.g., 5-10 events/second). Alternatively, the subscriber could also use updates with a low confidence of detection (e.g., 50-60%) but at a high sampling rate (e.g., 40 - 60 events/second). When given the choice, M would prefer notifications matching the first set of requirements to notifications satisfying the second set of requirements.

Preferences and Utility Functions

Preferences reflect the avoidable costs arising from having to identify, purge, repair, or replace data of inadequate quality at the subscriber [95, 143, 250, 277]. As shown in Figure 3.9, preferences map to requirements about specific sets of QoI properties (A, B) and utility (u_1, u_2) generated from meeting these requirements [424]. Preferences can be interdependent for some properties while they can be independent for others. For interdependent preferences, several requirements about different properties have to be satisfied simultaneously while any state is accepted for independent preferences [249]. For example, the subscriber requires a certain combination of sampling rate and confidence of detection (interdependent properties) in the example given above but does not care about loss or spatial resolution (independent properties).

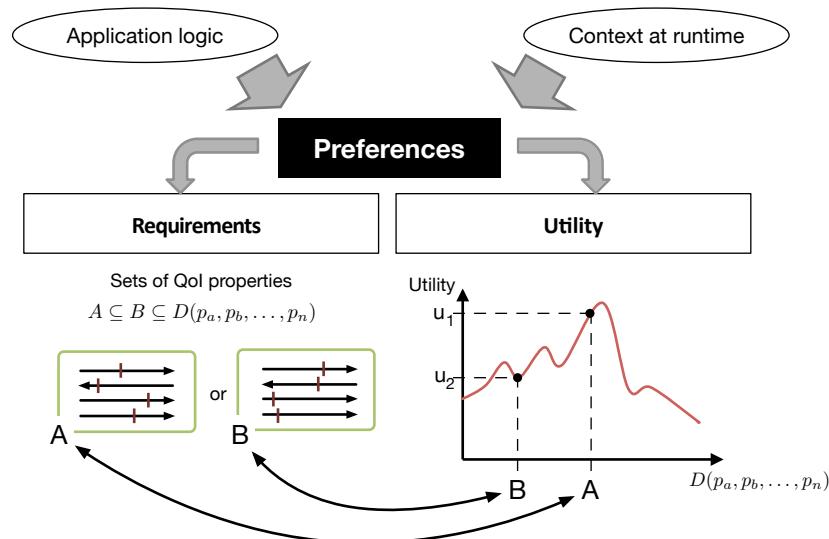


Figure 3.9.: Preferences of a subscriber map to requirements and utiliy values.

Preferences are mapped to values of a multi-dimensional utility function that is defined over all properties the subscriber has requirements about. A utility function $f_u : D \rightarrow \mathbb{N}_0^+$ maps a set of satisfied requirements $A \subseteq D(p_a, p_b, \dots, p_n)$ from a multi-dimensional space $D(p_a, p_b, \dots, p_n)$ to a score – the utility value u – with higher utility values reflecting a higher preference [424].

Increasing utility values for QoI correspond to the fact that the costs for ensuring data quality at the subscriber generally decrease with increased quality of the data provided by the MOM [143]. Conversely, utility values are an indicator for the subscriber's willingness to pay for receiving data that satisfies a specific set of requirements [381]. Please note that this does not necessarily result in a monotonic utility function as the utility value reflects the individual VoI of data received by a specific subscriber based on a combination of interdependent properties. As illustrated in the introductory example given earlier in this section, a subscriber would prefer data received at a lower sampling rate but with higher confidence of detection to data received with higher sampling rate and lower confidence of detection. Based on these two statements alone, we cannot automatically infer that receiving data at a high sampling rate and high confidence of detection has an even higher utility for the subscriber.

Modeling a multi-dimensional utility function mathematically, however, is challenging, as its exact form is often unknown even to the subscriber itself. In general, three different approaches can be distinguished that aim at overcoming this problem [249, 340, 424]: 1) using domain-specific utility functions that restrict D to a fixed set of properties with known interdependencies; 2) making constraining assumptions on the general form of the utility function to simplify modeling and handling the functions, e.g., monotonicity, continuity, differentiability, or general independence of properties; or 3) using a set of utility values and their associated sets of properties to reconstruct a utility function by extrapolating the available data points.

Using domain-specific functions restrict the generality of the resulting model and contradicts the intention of our approach. Simplifying the general form of the utility function and assuming independence between properties in turn allows to model and optimize each property independently and later decide on the optimal combination by adding the utility values of each independent utility function together [249, 424]. However, interdependent preferences as illustrated in the example cannot be modeled this way, as the resulting utility function requires that requirements about several properties are satisfied simultaneously.

Thus, our approach is inspired by the third alternative discussed above as it balances the expressivity and flexibility of the resulting model with the effort for defining it. Enabling subscribers to define discrete points on their individual utility function without having to know the form of the utility function between these points. Defining points on a utility function requires a subscriber merely to define target values for different sets of requirements they want to be satisfied and rank these sets comparatively by assigning different utility values to each set.

However, subscribers in reality often fail to provide exact values for all their requirements or define priorities between exact values in a comparative way [249, 340]. Thus, several approaches try to consider a certain degree of uncertainty when assessing requirements of subscribers. For example, Perera et al. [340] as well as Pernici and Siadat [346] require an exact value for a requirement but then distinguish between requirements that can be satisfied by an approximation (proximity-based [340]/fuzzy [346]) and those that have to be met exactly (point-based [340]/non-fuzzy [346]) using different evaluation strategies.

We allow a subscriber to define ranges of allowed values for each property. Each value that is at least within these ranges satisfies the requirement and generates the same utility. For in-

terdependent properties, these ranges represent *malleable* requirements: each value within the defined ranges of a requirement is equally accepted to satisfy the requirement, generating the same utility for the subscriber. In other words, the subscriber is indifferent to any value as long as it satisfies the requirement.

Consequently, all combinations of values satisfying a set of interdependent requirements simultaneously generate the same utility. We call such a combination of values a solution. The relationship between malleable requirements and solutions can be illustrated by the indifference curve of a set of requirements as shown in Figure 3.10a. An indifference curve represents a surface connecting all solutions the subscriber is indifferent to so they generate the same utility [249].



(a) Indifference curve for solutions satisfying *A*.

(b) Indifference curves for *A* and *B*.

Figure 3.10.: Indifference curves representing malleable requirements.

For alternative sets of requirements, indifference curves also illustrate the preference structure of the subscriber regarding sets of requirements as shown in Figure 3.10b for two sets of requirements *A* and *B*. Here the subscriber prefers to have *A* satisfied, indicated by *A*'s indifference curve being further away from the center than *B*'s — the subscriber would prefer any combination of values that would satisfy *A* over any other combination of values that would satisfy *B* but not *A*. The subscriber, however, is indifferent to how exactly *A* is satisfied, denoted by the different points on *A*'s indifference curve. Please note that these indifference curves represent preferences on an ordinal scale, i.e., they show a ranking between sets of requirements without interpretable distances between them [249].

3.3.1 Expectation Definition

We introduce *expectations* to encapsulate sets of malleable interdependent requirements, enabling subscribers to manage their individual preferences about quality-related properties at runtime. We do not require the subscriber to define its complete utility function or parts of it. We do not make any constraining assumptions beforehand about the form or type of a utility function, interdependencies of properties, or preference structures. Rather, we enable subscribers to define their requirements in a way that reflects their known preferences and allows them to adapt their requirements as soon as their context changes at runtime.

Our key idea is to allow subscribers to define a set of interdependent requirements about generic properties as an *expectation* and express the utility generated from satisfying these requirements. Alternative sets of requirements can be ranked by a subscriber according to its own preference structure without having to define the complete utility function.

Definition 3 (Expectation). An expectation describes a set of malleable requirements that a subscriber has about interdependent quality-related properties of notifications about events it has subscribed to. Each expectation X_i^e consists of a set of tuples (p_k, LB, UB) as well as a utility value $X_i^e.u$ which reflects the individual importance of this expectation for the subscriber and allows a ranking between alternative expectations. In each tuple, LB denotes the lower bound while UB denotes the upper bound of accepted values for the property p_k . \square

An expectation is associated with a subscription for notifications of a certain type or content that the subscriber has already registered at the MOM. However, an expectation is not part of the subscription itself as the requirements encapsulated in an expectation are subject to context changes of the subscriber that affect its requirements about quality-related properties but not those about a notification's content or type. Separating both types of requirements allows a subscriber to adapt its QoI requirements without having to change its subscription.

Each tuple in X_i^e refers to a requirement about a generic property p_k such as *sampling rate*, or *latency*. Instead of constraining a subscriber to define a single target value for each requirement, we allow for defining an interval $[p_k.LB; p_k.UB]$ of values that a subscriber would equally accept for the generic property p_k and the associated subscription as shown in Figure 3.11. This way, the subscriber expresses a preferential indifference about the values between $p_k.LB$ and $p_k.UB$, making the requirement malleable⁵.

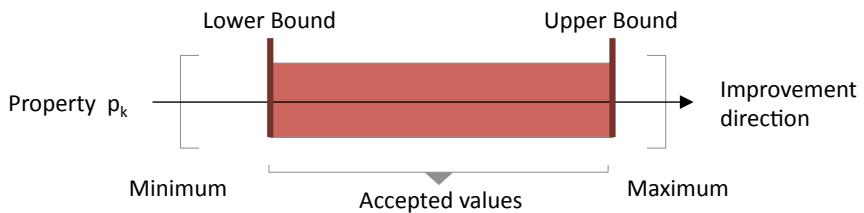


Figure 3.11.: Requirement about property p_k defined as part of an expectation.

Properties being part of an expectation are preferentially interdependent to the subscriber while they are preferentially independent to all properties not part of an expectation.

For example, subscriber M with expectation $X_1^e = \{(rate, 5, 10), (confidence, 75, 95)\}$ accepts notifications with $\{rate = 7, confidence = 90\}$ as well as notifications with $\{rate = 10, confidence = 80\}$. In addition, the subscriber would accept any value for properties such as loss or spatial resolution as they are not modeled in the expectation.

By combining requirements about different generic properties in a single expectation, each subscriber defines a trade-off between the ranges of those properties, making the requirements malleable. This reflects a preferential indifference of the subscriber regarding the state of each property as long as it is within the ranges defined in the expectation: each combination of values within the ranges of accepted values for p_a and p_b generates the same utility value.

Open and Closed Intervals

From the perspective of a subscriber, a requirement about a generic property p_k as part of an expectation could not only be satisfied but even overfulfilled. A requirement is satisfied if the system provides support for this generic property at a level that matches a value within $[LB; UB]$.

⁵ Please note that we can also express point requirements as well by setting $p_k.LB = p_k.UB$.

This is the case we have referred to so far. However, a requirement might also be overfulfilled by the system by providing a property at a level that matches a value $z \leq LB$ for minimizing generic properties and $z \geq UB$ for maximizing generic properties. For example, a subscriber requires the *trustworthiness* of data to be between 50%-75%, while the system provides data with a trustworthiness of 95%.

Requirements about most properties we have reviewed can be overfulfilled without causing a decrease in a subscriber's utility. For example, the utility of a subscriber does not decrease if data is provided with higher precision, accuracy, or trustworthiness than required. Requirements about other properties, however, should not be overfulfilled but exactly met by the system, as the subscriber perceives an overfulfillment as a disadvantage. For example, receiving notifications at a higher sampling rate or from too many alternative publishers than required results in more notifications that have to be processed by the subscriber: setting upper bounds for sampling rate and alternatives sets an upper limit to the number of notifications to be received by the subscriber (receiving rate). Overfulfilling such requirements is disadvantageous in particular for resource-constrained subscribers or those that have to purge data [143].

We denote requirements about generic properties that can be overfulfilled as *open intervals*. Requirements that have to be met exactly by providing values between $[LB; UB]$ are denoted as *closed intervals*. Figure 3.12 illustrates the relationship between the utility values of a subscriber and the values supplied by the system for the different types of generic properties.

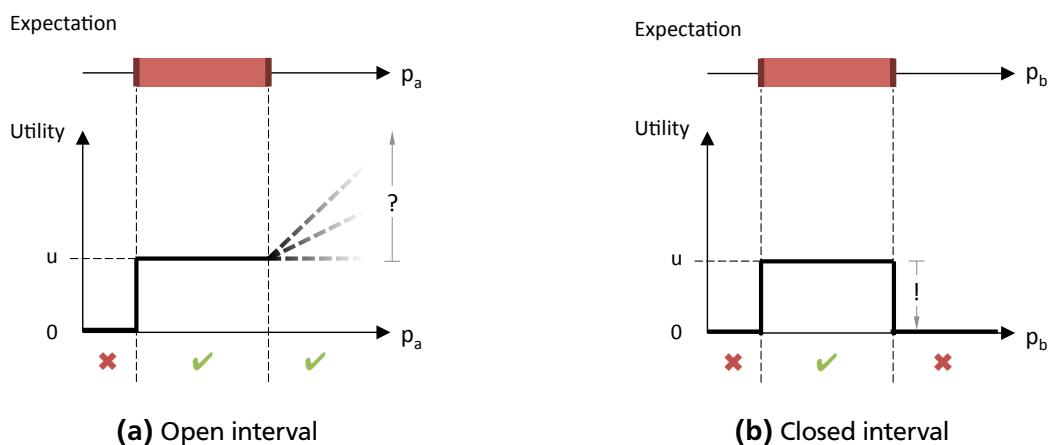


Figure 3.12.: Utility for a subscriber in case of open and closed intervals.

We illustrate the assumptions about open and closed intervals using two maximizing generic properties p_a (open) and p_b (closed) in Figures 3.12a and 3.12b. For open intervals, the utility is not decreasing if the system provides this property at a level that not only satisfies the requirement but that even overfulfills it by dominating even the upper bound of the expectation. Open intervals do not necessarily imply that the utility for the subscriber increases linearly or exponentially with the degree that the requirement is overfulfilled; in any case, we assume that it is not deteriorating. For closed intervals, the utility is decreasing if the property is provided at a higher level than required. For the sake of simplicity, we assume a utility value of 0 once a requirement is overfulfilled as shown in Figure 3.12b.

When illustrated with indifference curves, the indifference curve is only defined over the range of accepted values for closed intervals. Solutions outside this range of accepted values would not be represented by points on the indifference curve. Indifference curves for open intervals, however, would not end at the upper bound (for maximizing properties) or the lower bound (for

minimizing properties). Solutions outside the upper bound (for maximizing properties) or the lower bound (for minimizing properties) would be represented by points on that indifference curve nonetheless as they still generate the same utility for the subscriber.

Multiple Expectations to Express Alternative Sets of Requirements

As discussed earlier in this section, a subscriber might have different sets of requirements that result in different points on the utility function of the subscriber. For example, subscriber M needs highly reliable information but could alternatively do with less reliable information at a higher rate to compensate false-positives/negatives. However, receiving more data of a lower quality increases the costs for checking and purging the information contained in the notifications. Thus, the second set of requirements would not be preferred by M to the first set of requirements but it would be preferred to receiving data with best-effort properties, i.e., having no control at all.

We allow a subscriber to express those alternative configurations by defining multiple expectations all associated with the same subscription but ranked with different utility values [173, 175, 424]. Expressing alternative configurations that would generate the same utility for the subscriber is also possible with our approach.

We illustrate how to model and compare different sets of requirements by extending the requirements about *sampling rate* and *confidence of detection* as described in the initial example with complementary requirements about *trustworthiness*, *latency*, *precision*, and *accuracy*. The resulting two expectations X_1^e and X_2^e are defined in Table 6.3 and visualized in Figure 3.13. The subscriber's preference for the requirements formalized in X_1^e is quantified by associating a utility of 25 for X_1^e while meeting the requirements defined in X_2^e would only generate a utility of 5.

Table 3.2.: Example of two expectations defining the preferences of a subscriber.

X	Sampling Rate		Confidence		Trustw.		Latency		Precision		Accuracy		Utility
	LB	UB	LB	UB	LB	UB	LB	UB	LB	UB	LB	UB	
X_1^e	5	10	75	95	m	h	0	250	90	100	70	100	25
X_2^e	40	60	50	60	1	m	0	400	50	80	20	80	5

Note: Levels of trustworthiness are abbreviated with l = low, m = medium, h = high

The malleable requirements of X_1^e and X_2^e are shown as red overlays in the star plots in Figure 3.13a for X_1^e and Figure 3.13b for X_2^e . Using star plots⁶ [92] to visualize expectations allows us to visually compare expectations defined for more than two generic properties. Superimposing the star plot of an expectation with the corresponding capabilities of the system enables us to easily check if the expectation is already satisfied, could be satisfied by adaptation or whether it could not be satisfied even with adaptation (cf., Figure 3.21 in Section 3.4.2).

Star plots visualize multi-dimensional data in a two-dimensional way by generating a chart with a set of axes radiating from the origin, each axis representing a dimension and displaying only data of this dimension. Connecting the points of the same kind (e.g., all lower bounds) on all axes generates a profile that can be compared with profiles for other data sets. Star plots are used to identify similarities, outliers or concentrations on certain dimensions on an ordinary scale. They are unsuited, however, for interpreting the exact distances and ratios between different data sets, i.e., to interpret similarities or differences between data sets on an interval scale.

⁶ Star plots are also known as radar charts or spider web plots.

Looking at the star plots for X_1^e and X_2^e we can see that these two expectations are defined over the same set of generic properties but differ in at least two aspects. First, requirements contained in X_2^e are defined over a wider range of values than those being part of X_1^e . Second, requirements in X_1^e are stricter than those defined in X_2^e as they are located on the upper parts of the maximizable properties such as *trustworthiness*, *accuracy*, or *precision* (respectively, the more in the lower part of the minimizable property *latency*).

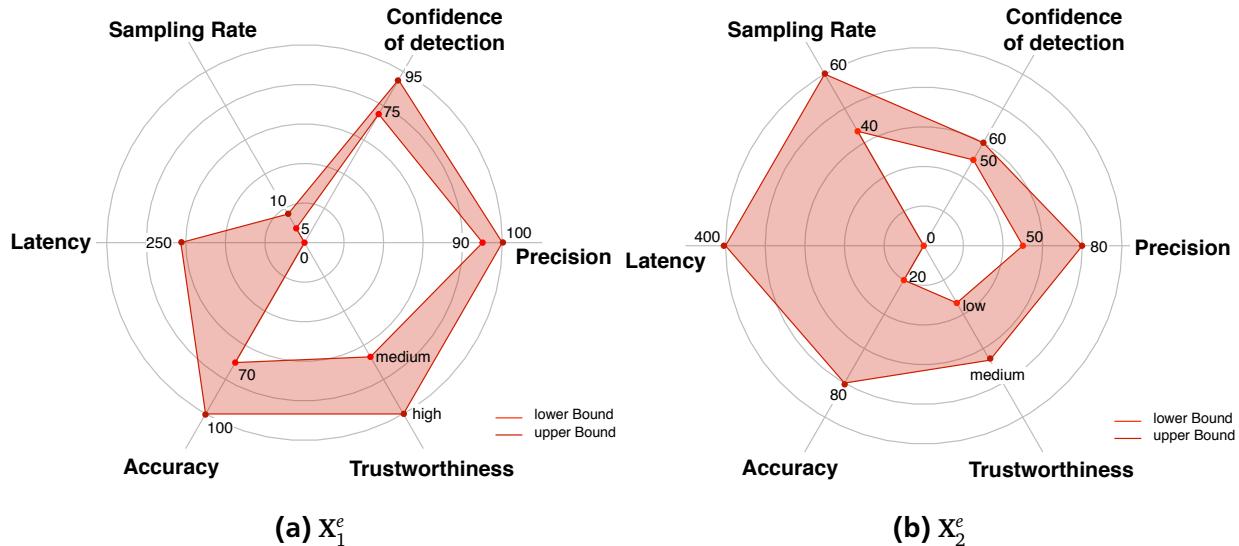


Figure 3.13.: Star plots for X_1^e and X_2^e as defined in Table 6.3.

Perspective of the MOM

From the perspective of the MOM, expectations for the same type of event can differ across multiple subscribers in their ranges and sets of required properties. We denote the total set of all expectations for e as \mathbf{X}^e .

3.3.2 Lifecycle of an Expectation

Each expectation has a lifecycle that can be manipulated by the subscriber at runtime as shown in Figure 3.14, enabling each subscriber to adjust its requirements at runtime based on changes to its context or state.

The lifecycle starts with *defining* an expectation locally at the subscriber based on valid property definitions. At this stage, only the subscriber is aware of the expectation and has not yet associated these requirements with notifications of a specific type or content.

Expectations have to be associated with a subscription when registering them at the MOM to make the system aware of the described requirements and trigger an initial negotiation. The same expectation can be associated with different subscriptions, though.

Registered expectations are active unless they are suspended or revoked by the subscriber. Active expectations are considered by the MOM when trying to satisfy a subscriber's requirements or optimize resource usage of the system. Passive expectations are ignored.

Changes in the context or state of a subscriber might influence its requirements about generic properties but not about the type or content of notifications. For example, a highly prioritized job

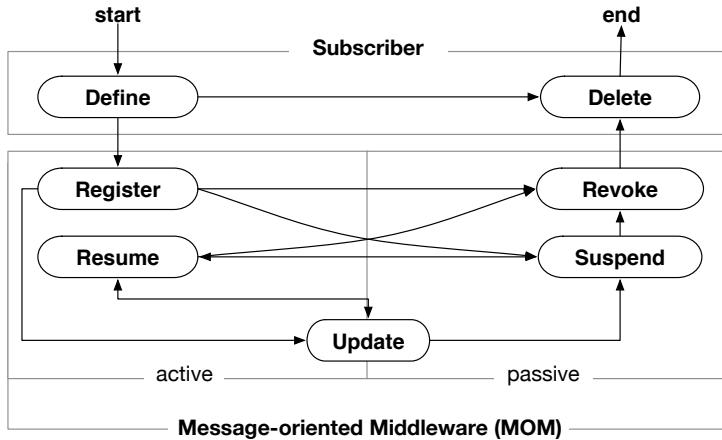


Figure 3.14.: Lifecycle of an expectation.

with tight Service Level Agreements (SLAs) has to be executed in a large-scale cloud deployment and is monitored by an application monitoring job executions. The application still requires the same kind of monitoring information (e.g., CPU utilization, memory usage, page misses) as for jobs with lower priority but now requires a higher sampling rate and higher precision to detect anomalies or congestion on the worker nodes and their virtual machines beforehand to guarantee that the job finishes without violating the negotiated SLA.

Those changes in the requirements of the subscriber can be reflected by: a) changing the lifecycle of a registered expectation; b) *updating*, *suspending*, or *revoking* already registered expectations; or c) by registering new expectations that reflect the new requirements. A suspended expectation is passive but can be updated as well. However, it has to be *resumed* into an active state before the MOM considers it again. An update can affect the set of generic properties an expectation is defined over, the ranges of accepted values, or the ranking of the whole expectation as expressed by its associated utility value.

Changes to the lifecycle of an expectation can be performed independently from the subscription if necessary, i.e., subscribers do not have to unsubscribe and subscribe anew. The lifecycle of an expectation, however, is tied to the lifecycle of a subscription in such a way that when a subscriber unsubscribes, the MOM revokes all associated expectations.

3.3.3 Fidelity: Quantifying the Satisfaction of a Subscriber

Subscribers express their preferences as expectations by mapping their subjective VoI properties and functions to requirements about generic properties that the system offers support for. The degree to which each subscriber is satisfied with the QoI of the data received from the system is measured by the subscriber's *fidelity* as shown in Figure 3.15.

The fidelity metric quantifies the conformance of the data a subscriber receives over time with the active QoI requirements that are formalized by an expectation and motivated by a subscriber's subjective VoI functions. The better the results delivered by the system are aligned with the requirements of the subscriber, the higher the degree of satisfaction [381].

The fidelity metric can be used to quantify the impact of different scenarios and data delivery strategies on the satisfaction of the subscriber. Measuring fidelity over time shows shifts in the degree of satisfaction based on variations of incoming data. We will illustrate both aspects with

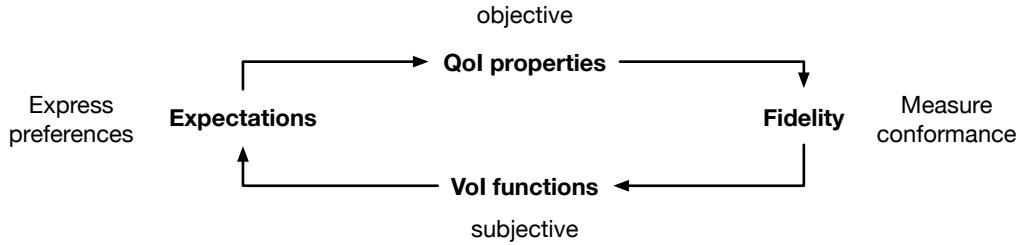


Figure 3.15.: Fidelity: quantify conformance of delivered QoI properties with own Vol functions.

an example later in this section using two small sets of notifications. Furthermore, we will quantify a subscriber's satisfaction for different adaptation strategies evaluated in our experiments in Chapter 6.

Positive values for a subscriber's fidelity indicate that the subscriber is satisfied with the data provided by the system while negative values indicate an overall dissatisfaction as data does not conform with the requirements and has to be purged, repaired, or replaced [424].

The degree of satisfaction is based on the total utility that is generated by the received data satisfying the currently active expectations. Dissatisfaction is expressed by applying a penalty for notifications that do not conform to the requirements. The penalty reflects the costs of a subscriber, as each notification with insufficient quality has to be purged or discarded. Expectations define requirements about single notification as well as sets of notifications. Thus, the overall satisfaction of the subscriber depends on the conformance of both single notifications (e.g., requirements about latency or accuracy) and sets of notifications measured over a given period of time (e.g., requirements about sampling rate or precision).

Equation (3.3) defines the fidelity metric for a set of n notifications e_1, \dots, e_n that have been received at the subscriber about events of type $\tau \in E$ and an active expectation X_j^e defined for a set of generic properties $\{p_l \in P | p_1, \dots, p_n\}$. The utility or penalty generated by single notifications and their contribution to the conformance of the respective set of notifications is summed up into the total fidelity for a given period of time.

$$Fidelity(n, X_j^e) = \sum_{i=1}^n \left[utility(e_i, X_j^e) - penalty(e_i, X_j^e) \right] \quad (3.3)$$

Computing the contribution of each notification to the overall utility and balancing this with an optional penalty requires the use of an auxiliary function f . As defined in Equation (3.4), f evaluates if a requirement of X_j^τ about p_l is satisfied or overfulfilled by a given notification e_i .

$$f(e_i, p_l) = \begin{cases} 1, & p_l \text{ is satisfied by } e_i \\ 0, & \text{otherwise} \end{cases} \quad (3.4)$$

The function *utility* as defined in Equation (3.5) returns the utility generated by notification e_i when satisfying \mathbf{X}_j^τ . As requirements of an expectation are interdependent by definition, \mathbf{X}_j^τ is satisfied if and only if all its requirements are satisfied or overfulfilled by e_i .

$$\text{utility}(e_i, \mathbf{X}_j^e) = u \cdot \prod_{l=1}^P f(e_i, p_l) \quad (3.5)$$

Conversely, an expectation is not satisfied if there is at least one requirement not satisfied or overfulfilled. In this case, the notification is not contributing to the overall utility. We apply a penalty that expresses a degree of dissatisfaction and reflects additional costs arising at the subscriber for trying to purge or repair the notification with respect to the insufficient property. All requirements that are part of an expectation are assumed to be interdependent. Thus, we cannot assess if the lack of a certain property is more disadvantageous to the subscriber than another⁷. Hence, we distribute the expectation's utility value equally across all properties when calculating the penalty for a notification. The resulting penalty is less severe for notifications with only a few insufficient properties. The function *penalty* is defined in eq. (3.6) for a notification e_i and a given expectation \mathbf{X}_j^τ .

$$\text{penalty}(e_i, \mathbf{X}_j^e) = \frac{u}{|P|} \cdot \left(|P| - \sum_{i=1}^P f(e_i, p_l) \right) \quad (3.6)$$

Equation 3.7 shows the total fidelity by filling in Equations (3.5) and (3.6) into eq. (3.3).

$$\text{Fidelity}(n) = \sum_{i=1}^n \left[u \cdot \prod_{l=1}^P f(e_i, p_l) - \frac{u}{|P|} \cdot \left(|P| - \sum_{i=1}^P f(e_i, p_l) \right) \right] \quad (3.7)$$

Please note that the fidelity metric allows us to measure the difference in the satisfaction of a subscriber if several alternative expectations with different utility values are registered (e.g., $\mathbf{X}_a^e \cdot u > \mathbf{X}_b^e \cdot u$). However, the fidelity metric explores the ordinal aspects of requirements, i.e., it does not evaluate how close we are to the upper and lower bounds defined by subscribers as long as we satisfy the requirement; using an interval scale for interpretation is part of future work.

Example for applying the fidelity metric

In the remainder of this section, we are going to use a small example to illustrate how the fidelity metric quantifies the satisfaction of a subscriber when receiving a set of notifications. Even with a few samples we can show how different parameter values for generic properties impact the overall quality of these notifications as perceived by a subscriber.

We assume that the subscriber has already registered two expectations \mathbf{X}_3^e and \mathbf{X}_4^e defined over the subset *latency*, *precision*, and *accuracy*⁸ of expectations \mathbf{X}_1^e and \mathbf{X}_2^e (cf., Table 6.3). We de-

⁷ Allowing subscribers to rank properties is a valid extension to the current model and part of future work but out of scope of this dissertation.

⁸ In some cases, knowledge about the true value \underline{v} might be available at the subscriber, e.g., by using reference values or calibrated reference sources.

fine \mathbf{X}_3^e as a subset of \mathbf{X}_1^e and \mathbf{X}_4^e as a subset of \mathbf{X}_2^e to simplify the scenario and concentrate on a few selected generic properties that can be easily evaluated at the subscriber. We assume requirements about all three generic properties to be open intervals for this example. \mathbf{X}_4^e has quite relaxed requirements compared to \mathbf{X}_3^e . However, the subscriber associates a utility value of 5 to \mathbf{X}_4^e , ranking it less important than \mathbf{X}_3^e with a utility value of 25. Figure 3.16 shows the star plots for both expectations.

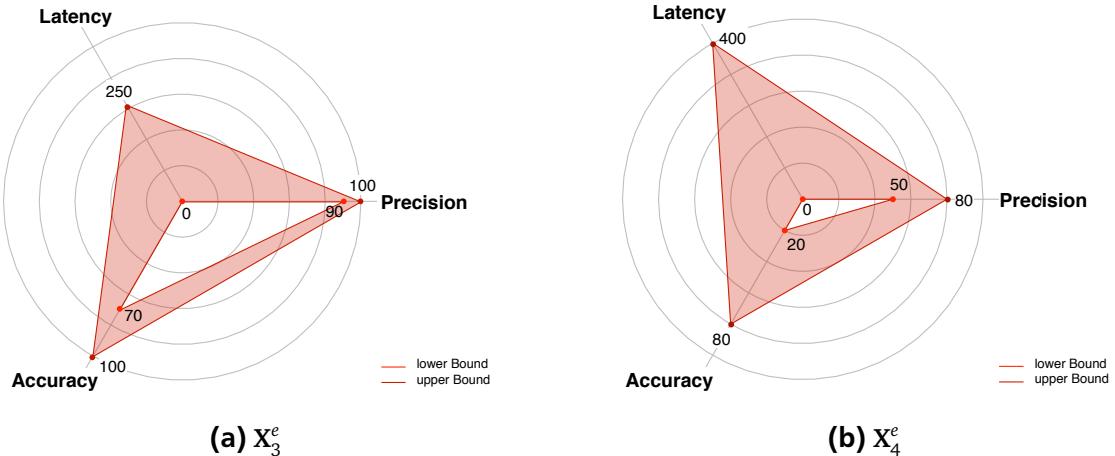


Figure 3.16.: Star plots of expectations \mathbf{X}_3^e and \mathbf{X}_4^e to be used in the example.

We check \mathbf{X}_3^e and \mathbf{X}_4^e against two sets of notifications, S_1 and S_2 , consisting of 10 notifications each. Each set of notifications is representing a different scenario. All notifications e_1, \dots, e_{10} are of the same type describing a numeric value v with $\underline{v} = 100$ being the known true value. Each notification e_k is represented by a tuple (e, t_s, t_r, v, pid) with e being the associated type of event, t_s the timestamp (milliseconds) the notification has been published by the publisher, t_r the timestamp (milliseconds) the notification has been received, v the value, pid a unique ID associated by the MOM to the publisher of e_k . Please note that this ID does not reveal the identity of the publisher but enables a subscriber to distinguish notifications from different origin. Using the information contained in a tuple, we can determine the generic properties at the subscriber as defined in the equations 3.8 (latency), 3.9 (precision), and 3.10 (accuracy) to check against an expectation by using the auxiliary function f as defined in Equation 3.4.

$$latency(e_i) = e_i \cdot t_r - e_i \cdot t_s \quad (3.8)$$

$$precision(e_1, \dots, e_n) = \left[1 - \frac{1}{\overline{e} \cdot \overline{v}} \cdot \sqrt{\frac{1}{n} \sum_{i=1}^n (e_i \cdot v - \overline{e} \cdot \overline{v})^2} \right] \cdot 100 \quad (3.9)$$

$$accuracy(e_k) = \left[100 - \sqrt{(e_k \cdot k - \underline{v})^2} \right] \quad (3.10)$$

Table 3.3 shows the contained value $e.v$, moving average $\overline{e} \cdot \overline{v}_k = \frac{1}{k} \cdot \sum_{i=1}^k v$, standard deviation s_k^2 , latency, precision and accuracy for each notification e_1, \dots, e_{10} . The parameter values of the generic properties computed for each notification are shown in Figure 3.17. As we can

see, the two sets of notifications are quite different for all properties. While S_1 has a higher latency on average than S_2 (cf., Figure 3.17a), both precision (cf., Figure 3.17b) and accuracy (cf., Figure 3.17c) are lower for S_1 than for S_2 . Table 3.4 contains details about the contribution of each notification to the total fidelity when applying \mathbf{X}_3^e while Table 3.5 contains details about the contributions when applying \mathbf{X}_4^e .

Table 3.3.: Values for sets S_1 and S_2 of notifications for the fidelity shown in Figure 3.17d.

Set	Notification				Properties (eqs. (3.8) to (3.10))		
	k	e.v	$\bar{e.v}_k$	s_k^2	Latency	Accuracy	Precision
S_1	1	50	50.00	0.00	250	50	100.00
	2	70	60.00	100.00	200	70	83.33
	3	30	50.00	266.67	300	30	67.34
	4	150	75.00	2075.00	280	50	39.26
	5	10	62.00	2336.00	290	10	22.04
	6	70	63.33	1955.56	330	70	30.18
	7	10	55.71	2024.49	277	10	19.24
	8	170	70.00	3200.00	280	30	19.19
	9	30	65.56	3002.47	260	30	16.41
	10	20	61.00	2889.00	260	20	11.89
S_2	1	90	90.00	0.00	150	90	100.00
	2	90	90.00	0.00	90	90	100.00
	3	100	93.33	22.22	105	100	94.95
	4	110	97.50	68.75	183	90	91.50
	5	100	98.00	56.00	230	100	92.36
	6	110	100.00	66.67	227	90	91.84
	7	80	97.14	106.12	215	80	89.40
	8	100	97.50	93.75	230	100	90.07
	9	100	97.78	83.95	215	100	90.63
	10	100	98.00	76.00	160	100	91.10

Computing the fidelity for S_1 and S_2 given \mathbf{X}_3^e and \mathbf{X}_4^e shows that the notifications contained in S_2 generate far more value for the subscriber as most of them satisfy not only \mathbf{X}_4^e but also \mathbf{X}_3^e . Plotting the contribution of each notification to the total utility in Figure 3.17d reveals those notifications that deteriorate the overall fidelity, e.g., $e_7 \in S_2$ when matched against \mathbf{X}_3^e .

Conclusion. As shown in this example, the subscriber would be dissatisfied with the data provided by the system if it would receive notification of S_1 as S_1 does neither satisfy \mathbf{X}_3^e nor the more relaxed requirements in \mathbf{X}_4^e as shown in Figure 3.17e. The overall dissatisfaction is shown by high negative values for the fidelity, reflecting the fact that the subscriber deems notifications of S_1 to be nearly useless for the intended purpose. On the contrary, the subscriber would be satisfied if the system provides notifications such as those contained in S_2 as these satisfy even the very restrictive requirements in \mathbf{X}_3^e . For that purpose, the system could filter out notifications or adapt to provide only notifications like those being part of S_2 .

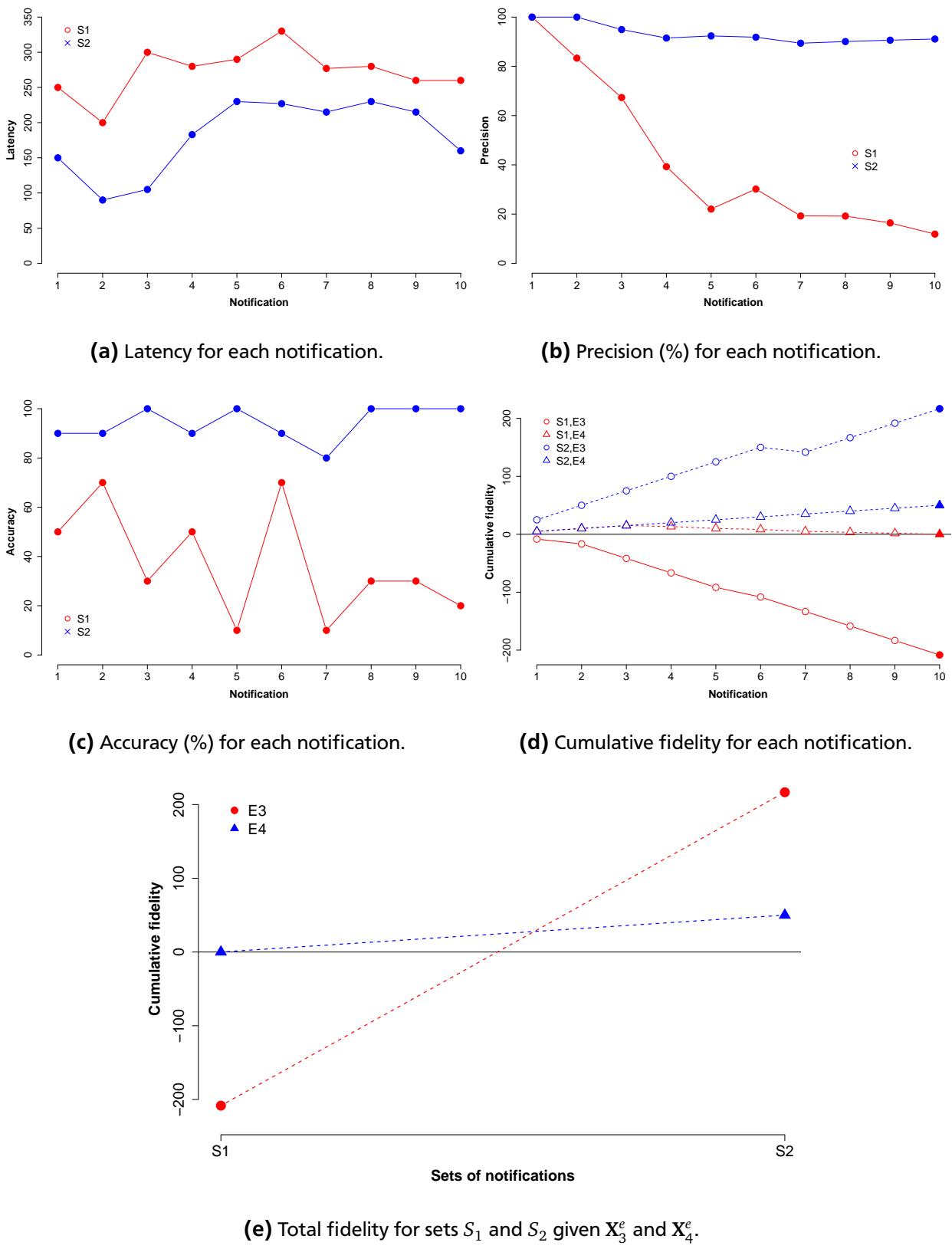


Figure 3.17.: Generic properties for S_1 and S_2 using eqs. (3.8) to (3.10).

Table 3.4.: Fidelity of notifications in S_1 and S_2 when using \mathbf{X}_3^e as shown in Figure 3.17.

Set	k	$f(e_k, p_l)$			Fidelity			$F(k)$
		$p_{precision}$	$p_{latency}$	$p_{accuracy}$	\mathbf{X}_3^e satisfied	$utility(e_k, \mathbf{X}_3^e)$	$penalty(e_k, \mathbf{X}_3^e)$	
S_1	1	1	1	0	0	0	8.33	-8.33
	2	0	1	1	0	0	8.33	-16.67
	3	0	0	0	0	0	25.00	-41.67
	4	0	0	0	0	0	25.00	-66.67
	5	0	0	0	0	0	25.00	-91.67
	6	0	0	1	0	0	16.67	-108.33
	7	0	0	0	0	0	25.00	-133.33
	8	0	0	0	0	0	25.00	-158.33
	9	0	0	0	0	0	25.00	-183.33
	10	0	0	0	0	0	25.00	-208.33
S_2	1	1	1	1	1	25	0.00	25.00
	2	1	1	1	1	25	0.00	50.00
	3	1	1	1	1	25	0.00	75.00
	4	1	1	1	1	25	0.00	100.00
	5	1	1	1	1	25	0.00	125.00
	6	1	1	1	1	25	0.00	150.00
	7	0	1	1	0	0	8.33	141.67
	8	1	1	1	1	25	0.00	166.67
	9	1	1	1	1	25	0.00	191.67
	10	1	1	1	1	25	0.00	216.67

Table 3.5.: Fidelity of notifications in S_1, S_2 when using \mathbf{X}_4^e as shown in Figure 3.17.

Set	k	$f(e_k, p_l)$			Fidelity			$F(k)$
		$p_{precision}$	$p_{latency}$	$p_{accuracy}$	\mathbf{X}_3^e satisfied	$utility(e_k, \mathbf{X}_3^e)$	$penalty(e_k, \mathbf{X}_3^e)$	
S_1	1	1	1	1	1	5	0.00	5.00
	2	1	1	1	1	5	0.00	10.00
	3	1	1	1	1	5	0.00	15.00
	4	0	1	1	0	0	1.67	13.33
	5	0	1	0	0	0	3.33	10.00
	6	0	1	1	0	0	1.67	8.33
	7	0	1	0	0	0	3.33	5.00
	8	0	1	1	0	0	1.67	3.33
	9	0	1	1	0	0	1.67	1.67
	10	0	1	1	0	0	1.67	0.00
S_2	1	1	1	1	1	5	0.00	5.00
	2	1	1	1	1	5	0.00	10.00
	3	1	1	1	1	5	0.00	15.00
	4	1	1	1	1	5	0.00	20.00
	5	1	1	1	1	5	0.00	25.00
	6	1	1	1	1	5	0.00	30.00
	7	1	1	1	1	5	0.00	35.00
	8	1	1	1	1	5	0.00	40.00
	9	1	1	1	1	5	0.00	45.00
	10	1	1	1	1	5	0.00	50.00

3.4 Capabilities: Support for Properties

In an EBS, support for expectations about generic properties depends on the degree to which these generic properties can be provided by publishers and MOM. As we have described in Section 3.2.2, generic properties can be provided either directly by publishers and MOM or indirectly by combining provided lower-level properties they depend upon. This support can change at runtime as many participants can adjust their support for a certain generic property dynamically by using self-adaptation [35, 96, 109, 324]. For example, publishers can adjust their sampling rate or confidence of detection while the MOM can use different mechanisms to determine the level of trustworthiness of publishers and their publications with varying degrees of confidence [62, 79, 99, 261, 362, 415].

Hence, support for generic properties in an EBS has two aspects: the actual value or level a generic property is provided at as well as the spectrum of potential support that a participant could realize by applying adaptation. As illustrated in Figure 3.18, actual and potential support for generic properties are subject to the heterogeneity of participants at design- and runtime.

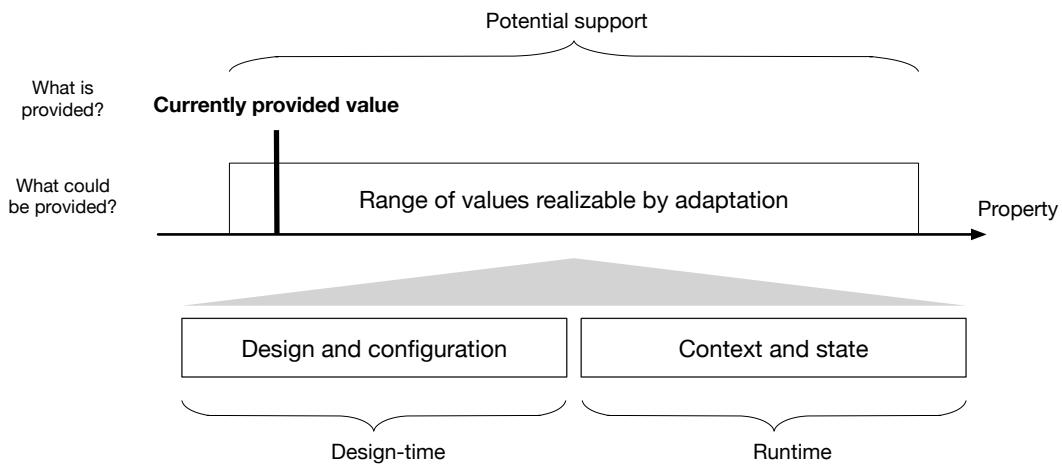


Figure 3.18.: Support for generic properties is influenced by the heterogeneity of participants.

Participants in an EBS can differ in their configuration as well as in their design, e.g., the software stack or hardware they run on, as we have discussed in Section 3.1 with regard to QoD and QoS. To different degrees, both design and configuration of a participant restrain the actual support for generic properties as well as the possible spectrum of adaptation. Some participants, for example, are cloud-based applications with less constraints regarding resources while others are run on mobile sensor nodes where energy-efficiency is a key issue and bandwidth is limited; while the former could process and provide high-resolution images for surveillance systems at a high sampling rate, participants on mobile sensor nodes might only be able to provide images with low resolution at a low sampling rate or they might not provide images at all.

Design and configuration also set the scope regarding the potential support for a generic property by enabling or restraining the ability of a participant to adapt at runtime. While some participants are designed as self-aware and self-adaptive applications that can autonomously adapt their supply for generic properties to changes in their context and state, others can adapt to predefined situations but need to be explicitly triggered and require detailed instructions; others cannot adapt their behavior at all.

Participants in an EBS are differently affected by changes to their context and state in addition to the heterogeneity based on designtime decisions. For example, sensors consume different amounts of energy depending on their context and current sensing task [35, 99, 109, 325, 343, 356, 415]. As sensors have a pre-defined power range they are expected to operate efficiently in, running low on energy may require battery-powered sensors to reduce their actual support for generic properties such as precision, sampling rate, accuracy or latency in favor of saving energy [62, 99, 261, 325, 415, 418]. This affects the actual and potential support of these generic properties.

In any case, providing a generic property at a certain level, or applying self-adaptation, comes at a cost for a participant [35, 37, 143, 250, 415, 424]. Direct costs arise either in terms of network usage for cloud-based applications, energy-consumption for energy-restricted participants, or general costs that are not always disclosed by a participant but influence the degree to which it supports a given property [6, 54, 62, 79, 261, 325, 354, 356, 384, 386, 415, 424]. Indirect costs arise from trade-offs that a participant has to make to provide a property at a given level or apply self-adaptation.

Self-adaptation does not only change the actual supply for one generic property but might require the participant to reduce or restrict the actual support for other generic properties as well. It can also influence the potential spectrum of adaptation still realizable for other generic properties. We illustrate this using two examples: 1) applying compression at a broker increases latency but might have to be applied instead of transmitting uncompressed data in a WSN or cloud-based settings [110, 173, 356]; and 2) a sensor might have to trade-off the accuracy of its measurements against the latency and frequency for reporting them [99, 325, 415].

As participants change their support over time, the MOM needs to know about the actual and potential support for a generic property available from active participants as well as the costs for providing each generic property at a given level.

3.4.1 Capability Definition: Spectrum of Support and Costs

We introduce the notion of *capabilities* to expose the actual and potential support a participant has to offer for a generic property. Using capabilities, a participant can describe *a*) the set of generic properties it supports; *b*) the actual support for each generic property; *c*) the potential support expressed as the full spectrum of realizable adaptation; and also *d*) quantify its current costs for providing that support.

In our approach, capabilities do not merely describe the current state of the system. Rather, they encapsulate information about the full spectrum a participant could and would support a generic property with by applying adaptation. Capabilities enable publishers and MOM to describe and price-in at runtime any restrictions imposed on them by their current state and context. Thus, capabilities contain all information required by the MOM to decide at runtime if and to which degree a participant might have to adapt in order to provide support for a given generic property that satisfies the expectation of a subscriber.

Definition 4 (Capability). A capability describes the extent to which participant j can support property p_k . Each capability C_k^e is a tuple $(p_k, LB, UB, CV, p_k.cost_{operate}(x))$ that defines *a*) the spectrum of support participant j in principle is capable of providing p_k at by applying adaptation, denoted as a range of values $[C_k^e.LB; C_k^e.UB]$; *b*) the actual support j is currently providing p_k at denoted

by the value $C_k^e.CV$ within the adjustable spectrum; and c) the cost function $p_k.cost_{operate}(x)$ for providing p_k at level x . \square

A capability describes the actual and potential support of a participant for a generic property as an interval of values denoting the realizable spectrum of adaptation as shown on the left-hand side of Figure 3.19. Providing p_k at a specific quality comes at a cost, described by the cost function $p_k.cost_{operate}(x)$. The cost function is defined over the interval $[C_k^e.LB; C_k^e.UB]$, i.e., $p_k.cost_{operate}(x) = \infty \quad \forall x. (x < C_k^e.LB \vee x > C_k^e.UB)$. As shown on the right of Figure 3.19, the cost function for a capability can be progressive, degressive, linear, or a step function [62, 79, 143, 424]. We do not impose any restrictions on the form of the cost function as it depends on the configuration, design, state and context of each participant and may vary at runtime.

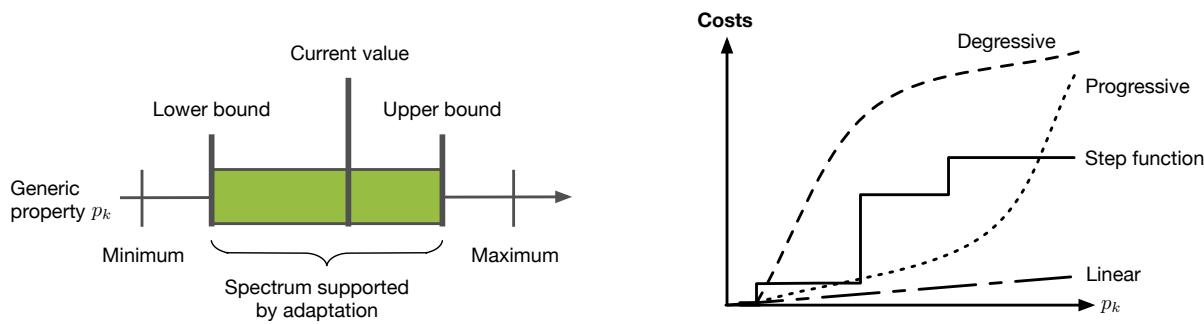


Figure 3.19.: Capability: actual and potential support for a generic property.

Please note that participants that cannot adapt at runtime can also use capabilities. Their capabilities are defined over an interval of length zero with $C_k^e.LB = C_k^e.CV = C_k^e.UB$. Consequently, the cost function is only defined for the current value, i.e., $p_k.cost_{operate}(x) = \infty \quad \forall x \neq C_k^e.CV$.

Capabilities abstract from the application-logic of participants as they do not describe concrete mechanisms or algorithms to manipulate a generic property. Thus, support for generic properties is represented in a way that is independent of the implementation and design of a specific participant, enabling the integration of a heterogeneous population of publishers and brokers. As we have discussed, providing support for a specific generic property might result in side effects on other generic properties at a dedicated participant. As these interdependencies depend on each individual participant, they are not explicitly modeled in a capability. Rather, a participant can express them by adjusting the spectrum, current value or costs for each generic property that would be affected.

Types of Capabilities

A capability describes support for a generic property. As we have already briefly discussed in Section 3.2, support for a generic property in an EBS has two aspects: determining the current state of that generic property and the ability to manipulate this state at runtime, i.e., increase or decrease it. In an EBS, different types of participants might have to take ownership for either of these aspects. The current state of the generic property *sampling rate* of a publisher, for example, has to be determined by the MOM while only the publisher can increase the sampling rate – the MOM, however, can also decrease it. While subscribers are oblivious to this due to their top-

Table 3.6.: Mapping generic properties to the types of participants determining and manipulating them based on [42].

Generic Property	Publisher	MOM		
	Atomic	Broker-side	Complex	Interdependent
Accuracy	■	□	□	□
Alternatives	□	■	■	■
Completeness	■ ¹	■ ²	■ ²	□
Compression	■ ³	■ ⁴	□	□
Confidence	■	□	□	□
Delivery Guarantees	□	■	■	■
Drift	■	□	□	□
Freshness	□	■	■	□
Latency	□	■	□	□
Order	□	■	□	■
Precision	■	□	□	□
Provenance	■	■	■	□
Resolution	■	□	□	□
Sampling Rate	■	■	□	□
Security	■	■	■	□
Sensitivity	■	□	□	□
Trustworthiness	□	■	■	■

¹ Number of attributes per notification.

² Number of notifications being part of a sequence.

³ Image compression for image sensors [57].

⁴ Lossy and lossless compression applied to a notification [356].

down view when defining expectations, the MOM must have a bottom-up perspective to identify those participants that determine and manipulate different generic properties.

From the perspective of the MOM, capabilities about generic properties can be atomic, broker-side, complex, or interdependent as shown in Table 3.6. Multiple matches indicate that a coordination between publisher and MOM is necessary when having to manipulate the respective generic property. For example, to guarantee the completeness of a sequence of notifications, both publisher and MOM have to ensure that no notification is dropped.

A capability about a generic property is *atomic* if it is determined and manipulated by the publisher. From the perspective of the MOM, a capability about a generic property is *broker-side*, if only the MOM can evaluate the current state of the system for this property at runtime⁹.

For example, only the MOM at runtime can determine the *latency* property as experienced by subscribers, as it depends on two lower-level properties as shown in Figure 3.20 (left). The overall latency is the sum of the latency of processing notifications from the MOM to subscribers (*latency_{forwarding}*) as well as of the latency for receiving the notification at the MOM from the publisher first (*latency_{publication}*); the latter defining an effective lower bound for the latency.

In addition to being broker-side, a capability about a generic property can also be *complex* if the MOM has to consider several lower-level properties when evaluating expectations about this generic property. Thus, complex capabilities indicate relationships between generic properties

⁹ In principle, the MOM can crosscheck every generic property at runtime. However, this would require increasing resources for introspection and depends on the capabilities of the MOM. Thus, we mark a capability as broker-side only if only the MOM can assess it at runtime but not by the publisher on its own.

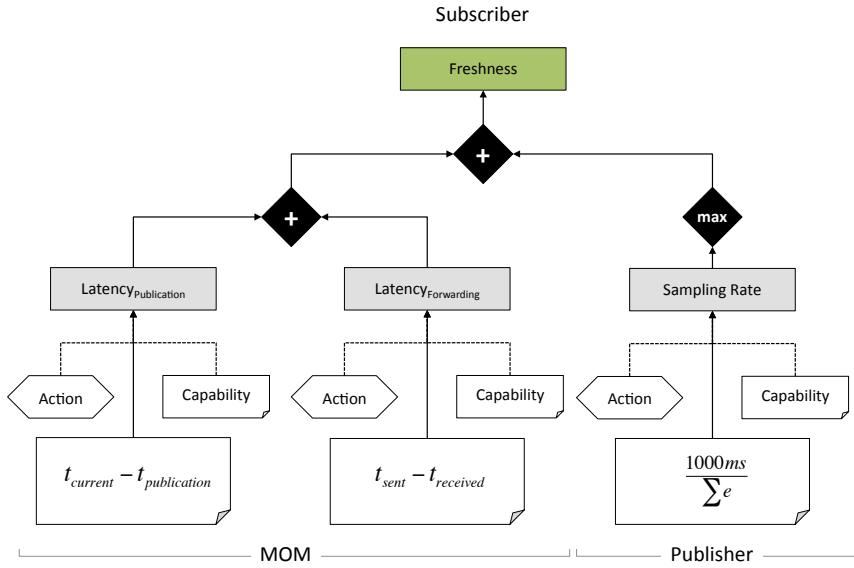


Figure 3.20.: Freshness to illustrate a broker-side and complex generic property.

that are modeled as graphs or ontologies as discussed in Section 3.2.2. *Freshness*, for example, depends on three different lower-level properties as shown in Figure 3.20. Determining the freshness of a type of notification requires the MOM to consider the overall broker-side latency for this type of notification as the sum of $\text{latency}_{\text{forwarding}}$ and $\text{latency}_{\text{publication}}$ together with its *sampling rate* as an indicator for the maximum amount of time that could have elapsed between the event actually happening and its detection by the publisher [418].

Finally, a capability about a generic property can also be *interdependent* from the perspective of the MOM. Interdependent capabilities denote support for generic properties that cannot be provided by a single participant alone. Rather it depends on the contributions of multiple participants. In contrast to complex capabilities, evaluating an interdependent capability does not necessarily require to evaluate several different generic properties.

A capability about the generic property *alternatives*, for example, is interdependent: requirements about alternatives can only be satisfied if a certain number of publishers is available where each publisher provides not only the required type of notification but also supports a set of generic properties to a degree that satisfies the other requirements defined in the expectation alongside the requirement about alternatives. For example, we could add a requirement about alternatives to expectation X_3^e (cf., Section 3.3.3), requiring notifications about events of type e to be provided by at least 2 and at most 4 different publishers that all provide these notifications about e with at least the required latency, accuracy, and precision. Thus, a capability about alternatives can be broker-side or complex but does not have to be if the expectation is defined only over atomic generic properties such as precision.

3.4.2 Capability Profiles: Characterizing Publishers

In an EBS, publishers provide all data as notifications while the MOM can only determine or manipulate certain generic properties of these notifications. Thus, the system is only able to satisfy an expectation if there is at least one publisher available that provides notifications with generic properties matching the set of generic properties required by the subscriber.

We introduce the notion of a *capability profile* to characterize a publisher in terms of its overall support for a set of generic properties. A capability profile bundles all capabilities of a publisher for notifications of a given type of event together with those capabilities supported by the MOM. Thus, a capability profile reflects the full set of capabilities available from a specific publisher for notifications of a given type of event and can be matched against expectations.

Definition 5 (Capability Profile). A capability profile CP_j^e is a set of capabilities $\{C_1^e, \dots, C_k^e\}$ associated with publisher j for events of type e . It consists of atomic as well as broker-side capabilities about generic properties p_1, \dots, p_k . \square

Multiple publishers can provide the same capability but with different ranges, current values, or cost functions, i.e., support for a specific generic property. However, each publisher can define only one capability per generic property. Consequently, capability profiles for the same type of event (CP^e) but associated with different publishers can be heterogeneous in terms of the

1. set of generic properties, e.g., $CP_2^e = \{C_{rate}^e, C_{latency}^e\} \subset CP_1^e = \{C_{rate}^e, C_{latency}^e, C_{confidence}^e\}$;
2. potential support available for each property through adaptation; and
3. the current values provided as actual support.

We use an example to show how capability profiles describe different publishers regarding their support for generic properties. We show how we can use capability profiles to visually determine if an expectation can be satisfied by notifications of a publisher or not. An algorithmic approach for this is presented and discussed in detail in Chapter 4.

We define three capability profiles CP_1^e , CP_2^e , and CP_3^e about the generic properties *sampling rate*, *confidence of detection*, *latency*, *precision*, and *accuracy*. Their lower and upper bounds as well as current values are shown in Table 3.7; cost functions are omitted for the sake of simplicity¹⁰.

Table 3.7.: Example of three capability profiles characterizing different publishers.

CP	Sampling Rate			Confidence			Trustw.			Latency			Precision			Accuracy		
	LB	CV	UB	LB	CV	UB	LB	CV	UB	LB	CV	UB	LB	CV	UB	LB	CV	UB
CP_1^e	0	10	60	50	80	95	n	m	m	100	200	240	50	95	100	60	80	100
CP_2^e	0	25	60	50	60	80	n	m	m	100	200	240	50	70	95	70	80	90
CP_3^e	0	30	60	50	61	80	n	1	1	200	300	400	40	60	80	50	70	80

Note: Levels of trustworthiness are abbreviated with l = low, m = medium, and h = high.

Similar to expectations, we can visualize capability profiles in Figure 3.21 using star plots with each axis representing the support for a generic property. On each axis, we plot three values: the actual support for this generic property denoted by the current value (black), the minimal value that could be provided after adaptation (light green), as well as the maximum value that could be provided after adaptation (dark green). Connecting the dots of each type of value between the different axes and filling out the area between the lower bound and the upper bound visualizes the profile of a publisher: the green area denotes the potential support, i.e., all combinations of values that could be provided by applying adaptation, while the black ring shows the actual support for all generic properties the capability profile is defined about.

¹⁰ In Chapter 4 we will have a close look at how different cost functions affect the decisions of the MOM during runtime negotiation.

As we can see from the star plots in Figure 3.21, CP_1^e and CP_2^e are quite similar with CP_1^e having higher current values for *sampling rate*, *confidence of detection*, *precision*, and *accuracy* (cf., Figure 3.21a and Figure 3.21c). Furthermore, a publisher supporting capabilities as bundled in CP_1^e would offer a wider spectrum of adaptation than a publisher characterized by CP_2^e . By contrast, the potential support for generic properties as described in CP_3^e is rather limited, most obvious for *trustworthiness*, *precision* and *accuracy* (cf., Figure 3.21e).

These differences are important when determining whether an expectation can be satisfied or not. We use expectation X_1^e we have discussed in Section 3.3.1 as an illustration. We visually compare each capability profile with X_1^e by superimposing the star plot of X_1^e (cf., Figure 3.13a) with the star plot of each capability profile. This is shown in Figure 3.21b, 3.21d, and 3.21f.

Matching X_1^e against CP_1^e (cf., Figure 3.21b) shows that the expectation would be satisfied by notifications published by a publisher characterized by CP_1^e : the current state of each generic property is satisfying the expectation's requirements as illustrated by the black line of the current values of CP_1^e being covered by the red area of X_1^e that defines its range of accepted values. Matching X_1^e against CP_2^e (cf., Figure 3.21d) in turn shows that the expectation would not be satisfied at the moment as the current values for sampling rate, confidence of detection and precision are not covered by the red area of X_2^e . However, the green area of the potential support of CP_2^e is overlapping with the red area of X_2^e . Hence, adjusting the current values for these generic properties could satisfy X_1^e . Matching X_1^e against CP_3^e , however, shows that CP_3^e neither does nor could satisfy the expectation. Even by applying adaptation, support for precision and trustworthiness could not be provided at a level required by this expectation as illustrated by the gap and missing overlap between the green and the red areas in Figure 3.21f.

3.4.3 Lifecycle of Capabilities and Capability Profiles

Each capability has a lifecycle as shown in Figure 3.22 that can be manipulated by a publisher or the MOM to reflect changes to the current value, the spectrum of realizable adaptation or the cost function of a capability.

Consequently, each capability profile has the same lifecycle states as a capability and is influenced by transitions in the lifecycle of each capability it consists of. A capability profile's lifecycle starts with *defining* its first capability at a local publisher. As with expectations, capabilities become active with the MOM becoming aware of a publisher's capabilities by associating them at the broker with an existing advertisement. A capability profile is *registered* at the MOM by registering the first capability for this capability profile. The lifecycle of a capability ends with *revoking* it; that of a capability profile ends with revoking the last capability it consists of.

During runtime, the situation of a publisher might change in a way that requires *updating* registered capability profiles without changing the advertisement. For example, a publisher might still be able to publish its GPS position but at a lower rate or with a reduced spatial resolution for consuming energy. Conversely, new resources might become available at runtime that improve the support for a given property. For example, higher *confidence of detection* can be achieved due to better contextual information during fusion [225].

A capability profile is automatically revoked if the associated advertisement is revoked by the publisher; in case of the same capability profile being associated with multiple advertisements it remains active and registered until the last associated advertisement has been revoked.

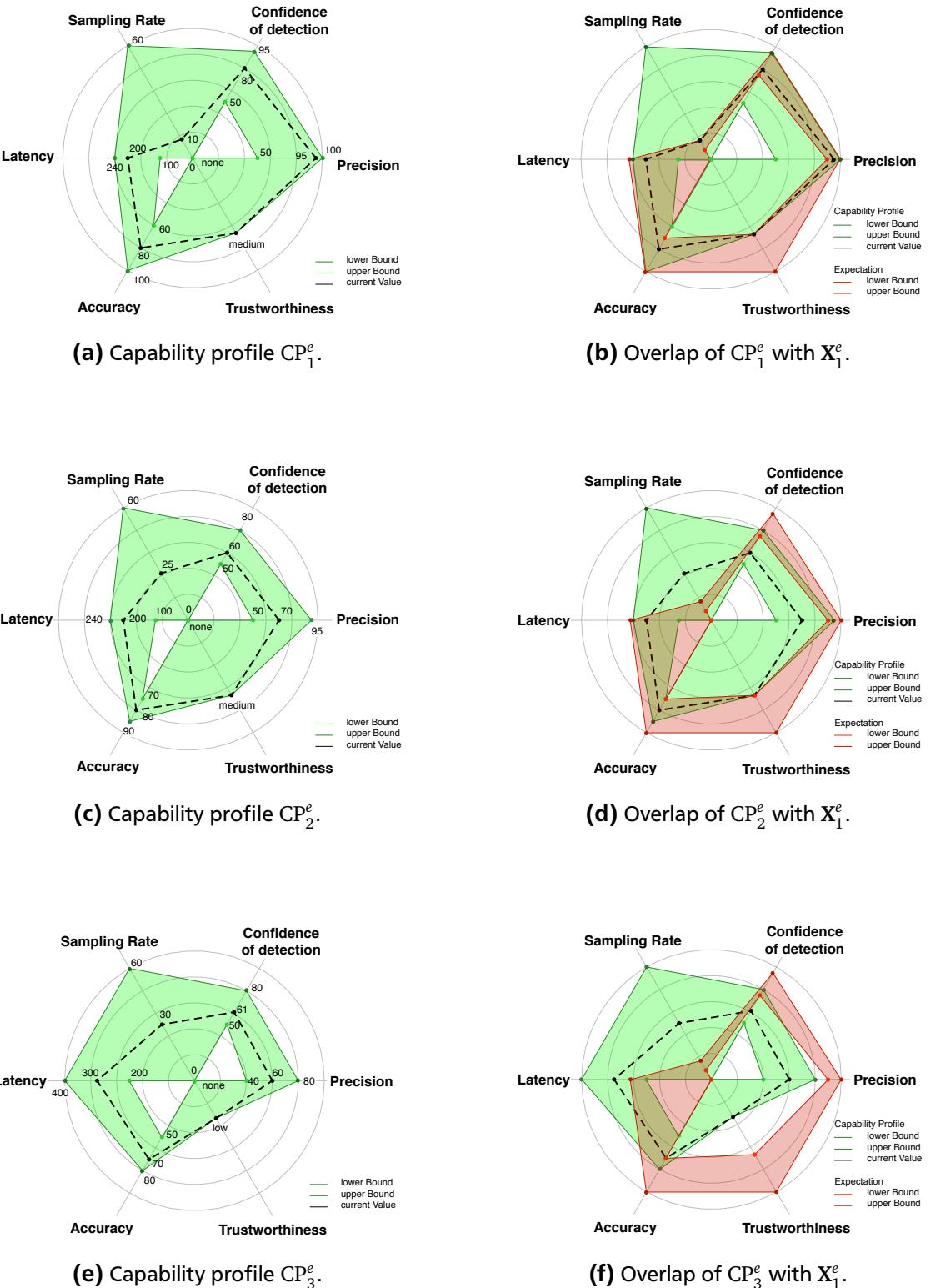


Figure 3.21.: Star plots showing examples for capability profiles CP_1^e , CP_2^e , and CP_3^e as well as their ability to support requirements defined in expectation X_1^e (cf., Table 6.3).

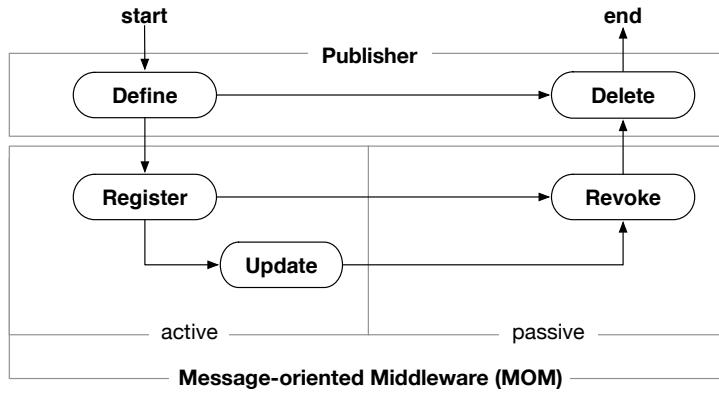


Figure 3.22.: Lifecycle states and transitions of a capability as well as a capability profile.

3.5 Feedback: Enabler of Self-Adaptation and Renegotiation

Feedback encourages a more efficient usage of system resources as it enables participants to adapt their behavior at runtime and enables multi-round negotiations with the MOM about the capabilities to provide and the expectations to satisfy [144].

In an EBS, publishers and subscribers are able and willing to adapt their behavior at runtime to react to changes in their context or the system state [2, 3, 60, 61, 63, 68, 114, 220, 221, 222, 353]. They require feedback about their individual expectations or capabilities as well as the system state to assess whether they have to adapt. Currently, however, such feedback is not available at runtime in an EBS as there is only a unidirectional flow of information from publishers to subscribers as neither subscribers nor publishers get any feedback [171]. For example, publishers do not know if their publications meet any demand by subscribers and vice-versa.

We provide participants with individual and aggregated feedback from the MOM at runtime as shown in Figure 3.23. The different types of feedback and their respective recipients are discussed in detail in the remainder of this section.

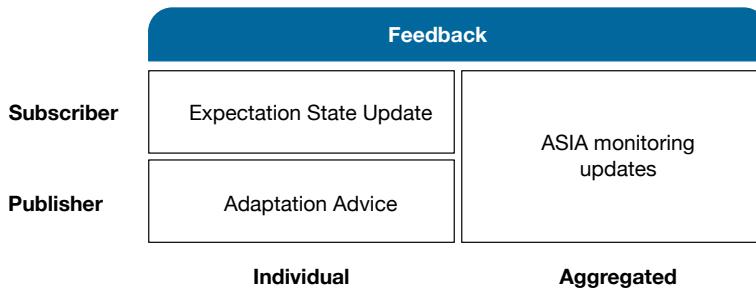


Figure 3.23.: Types of feedback given by the MOM and the respective recipients.

3.5.1 Individual Feedback

Individual feedback is intended for a dedicated participant. It contains *state updates* from the MOM about expectations of a subscriber or *adaptation advices* for capabilities of a publisher.

Expectation State Updates to Subscribers

Subscribers get informed if the state of their active expectations has changed. This individual feedback is provided by the MOM and can be used by subscribers to assess the currently consumed notifications regarding their QoI. It may trigger changes to expectations and start a renegotiation. From the perspective of a subscriber, an active expectation can be either *satisfied*, *unsatisfied* or *Pending* to be satisfied as shown in Figure 3.24.

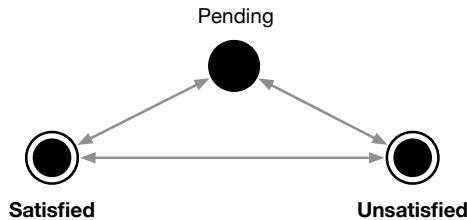


Figure 3.24.: States and state transitions of an expectation from the subscriber's perspective.

Receiving feedback that an expectation is currently satisfied informs the subscriber that from now on notifications being forwarded from the MOM correspond to the requirements formalized in the expectation to the degree pre-defined by the MOM. For example, 90% or 95% of notifications satisfy the mentioned expectation.

Sometimes, the system needs to adapt itself first before being able to satisfy an expectation. In this case, the subscriber is informed about the expectation being pending to be satisfied. As soon as the system has finished its adaptation and the expectation becomes satisfied, the subscribers is informed about this change of state as described above.

If an expectation's state changes to unsatisfied, subscribers are informed about the reason an expectation cannot be satisfied at the moment. For example, the context of a publisher has changed so that certain capabilities are not available anymore to the degree necessary to satisfy the expectation about the corresponding property; or a certain property is currently not supported by the system at all (e.g., as there are no active publishers with suitable capabilities).

Definition 6 (Reason). *The MOM expresses reasons for not being able to satisfy an expectation X_i^e as tuples (X_i^e, p_x, α) with α describing the value currently provided by the system for the generic property p_x that has an unsatisfied requirement defined as part of expectation X_i^e .*

Including the currently available value for a specific generic property into a reason provides the subscriber with a reference value to check its current requirements against. Knowing the current state of the system for its individual bundle of requirements, a subscriber becomes empowered to revise its requirement. It might be that the currently available value for this generic property turns out to be sufficient for subscriber in its current situation or context even if this had not been foreseen and expressed by a matching expectation beforehand. Without any reference value, however, the subscriber would just receive the information that its requirements are not satisfied but have no indication about the orders of magnitude it would have to adjust its requirements in order to get them satisfied.

As soon as the expectation can be satisfied or is pending to be satisfied, the subscriber is notified about the new state.

Adaptation Advices to Publishers and MOM

Publishers and neighboring message brokers in a distributed MOM receive explicit *adaptation advices* if the support for a generic property has to be adjusted. Reacting to context changes that affect publishers, subscribers or the MOM, optimizing the usage of system resources, or becoming able to satisfy expectations can be reasons for having to adjust support for properties. Using adaptation advices, we extend the scope of support for generic properties to those manipulated by publishers.

Referring to the notion of *actions* introduced in Section 3.2.2, an adaptation advice can be the result of the MOM choosing an action to increment or decrement the current state of a property as shown in Figure 3.25. For example, as shown in Equation (3.1), the action *adaptPublisher* defined in the MOM for $p_{samplingRate}$, indicates that the current state of the generic property *sampling rate* can be decreased by adapting a publisher with a registered capability for *sampling rate*. This would result in an adaptation advice given from the MOM to the respective publisher.

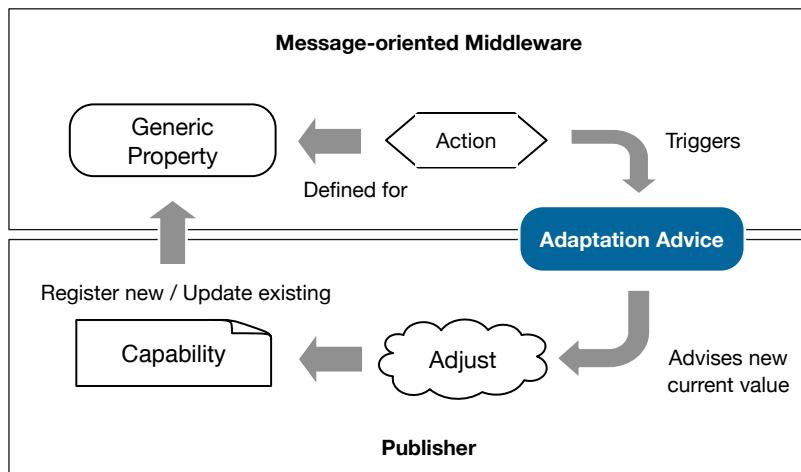


Figure 3.25.: Relationship adaptation advice, capability, generic property and actions.

Definition 7 (Adaptation Advice). An adaptation advice for a specific capability profile includes the list of capabilities to adapt. This information is provided as tuples (CP_j^e, C_k^e, β) where C_k^e denotes the capability to adapt as part of capability profile CP_j^e , and β defines the required target value for $C_k^e.CV$. \square

Please note that an adaptation advice does not state how a participant should adapt but rather defines the desired outcome of the adaptation for a certain generic property. Thus, we abstract from the implementation of the publisher and enable for triggering self-adaptation in heterogeneous populations of publishers. Participants can decide locally about the most suitable mechanism to apply in order to meet the new requirements set by the MOM.

In distributed networks of brokers, adaptation advices to publishers are forwarded to the edge broker being directly connected to the respective publisher.

For complex properties or those that require a sequence of actions to be executed, adaptation advices received from a neighboring broker are broken down into several other adaptation advices or parameters for locally applied actions.

After a successful adaptation, the publisher or broker would update the current value $C_k^e.CV$ of the capability C_k^e that has changed.

3.5.2 Aggregated Feedback

While individual feedback focuses on specific expectations and capabilities of a particular participant, aggregated feedback refers to runtime monitoring of an EBS. While only a single participant receives individual feedback, aggregated feedback is broadcasted to all participants that have registered interest in it – subscribers, publishers, and brokers in a distributed MOM. Aggregated feedback provides participants with updates about the population and dynamics of the system at runtime.

The population of an EBS is described by metrics such as the number of publishers or subscribers active for notifications of a certain type $e \in E$, set of generic properties, or content as defined in Equations (3.11) and (3.12). For example, the total number of active publishers providing temperature data about cargo container #50 with at least 75% confidence of detection.

$$publisherCount_e \quad \# \text{publishers active for } e \in E \quad (3.11)$$

$$subscriberCount_e \quad \# \text{subscribers active for } e \in E \quad (3.12)$$

The dynamics of an EBS refers to how demand and supply for notifications changes over time. Key metrics are the average sampling rates of notifications or rates that notifications are subscribed to or unsubscribed from. Equations (3.13) and (3.14) define these metrics for notifications about events of type $e \in E$.

$$samplingRate_e \quad \# \text{notifications/sec } e \in E \text{ is published with} \quad (3.13)$$

$$subscriptionRate_e \quad \# \text{subscriptions/sec for } e \in E \quad (3.14)$$

Using these metrics we can define composite Key Performance Indicators (KPIs) to describe the importance of every publisher j and subscriber i based on the set of event types e they publish ($E_j^P \subseteq E$) notifications about or subscribe to ($E_i^S \subseteq E$). For each publisher j we define the total number of subscribers that j is serving across all published types of notifications in eq. (3.15). The number of subscribers consuming notifications published by j is one indicator for the importance of j for the EBS.

$$servedSubscribers_j = \sum_e^{E_j^P} subscriberCount_e \quad (3.15)$$

However, j might not be the only publisher providing notifications about events of type e . Thus, we compute the Power of Supply (PoS) of j as defined in eq. (3.16). This KPI expresses the

relative importance of a single publisher for providing subscribers with notifications about type e ; this importance decreases with an increasing number of alternative publishers.

$$relativePowerOfSupply_{e,j} = \frac{1}{publisherCount_e} \quad (3.16)$$

Complementary from the perspective of a subscriber, the Power of Demand (PoD) puts into perspective the importance of a single subscriber as defined in eq. (3.17).

$$PowerOfDemand_e = \frac{subscriberCount_e}{\sum_e^E subscriberCount_e} \quad (3.17)$$

We use four examples to illustrate how *aggregation updates* about such metrics can be further utilized by the different types of participants in an EBS depending on their design or context:

Example 1: Some collaborative sensing tasks in the IoT require a minimum number of autonomous sensors to be active at the same time. An insufficient number of alternative publishers for notifications about a certain type of event or set of generic properties renders the sensing task impossible while an excessive number of active publishers has to be avoided to optimize the overall utilization of the system [339, 386]. Thus, a publisher requires knowing the number of other active publishers currently providing similar notifications. Depending on its business logic it might be able to turn itself into an alternative publisher if required or hibernate in case of too many publishers being available. Using Equation (3.16), a publisher would define two thresholds for its PoS: one that results in hibernation when under-run and one that activates the publisher when exceeded.

Example 2: Subscribers might require notifications to be supplied by a certain number of alternative publishers to perform a certain task, or they are resource-restricted and need to calculate a maximum receiving rate based on requirements about a maximum sampling rate and number of alternative publishers (cf., Section 3.2.1). For both types of tasks, subscribers need to know the current number of active publishers as defined in Equation (3.11) as well as the current sampling rate for a certain type of event and set of generic properties as defined in Equation (3.13).

Example 3: Brokers in a distributed MOM have to monitor the number of active publishers providing notifications with a specific set of properties to detect changes in the system state that require a renegotiation of expectations. Furthermore, approaches for load-balancing and optimization need to quantify the importance and replaceability of a given publisher as defined in Equations (3.15) and (3.16), or for a given subscriber (cf., Equation (3.17)).

Example 4: A company participates in inter-organizational business processes that rely on a distributed EBS across companies for exchanging information. Assessing the importance of each subscriber or publisher for runtime governance requires up-to-date monitoring information about the state of the system [78, 169, 170, 177]. The company's IT Service Management (ITSM) requires information about the supply for each subscriber in addition to the information required for load-balancing as described in the preceding example.

While the necessary monitoring information is easy to obtain for a centralized MOM at runtime, it is not available to brokers in a distributed MOM — they do only know about their directly connected neighbors [170, 171, 172].

Effective Runtime Monitoring

Providing effective runtime monitoring for decentralized and distributed EBS remains an open research topic due to the inherent anonymity and scalability of EBS.

Current approaches have limited effectiveness, as they require the deployment of additional monitoring overlays (e.g., SDIMS [427] or Adam2 [365]) that provide a fixed set of available metrics with limited granularity. On the conceptual level, they are unsuited for expressing subjective VoI, which requires different sets of QoI properties as discussed in Section 3.2.1 and Section 3.3. On the technical level, these additional monitoring overlays generate traffic overhead and additional effort for operation and maintenance, which increases monitoring costs [171, 172].

We propose a new approach to effectively monitor large-scale distributed EBS based on the concept of Application-specific Integrated Aggregation (ASIA). Our approach provides fine-grained aggregated metrics about the population and dynamics of an EBS at runtime without compromising performance and scalability of the monitored EBS.

Participants using our approach do not have to frequently pull information about the current state of the system. Instead, they are informed proactively only if the state of the system has changed to a degree that they have defined as being significant for them. The same metrics are available for publishers, subscribers, and brokers of the MOM.

Our approach allows participants to individually specify the *aspect* of the system state they are interested in, the *granularity* of the metrics they want to be kept updated about, and the *precision* of these updates. The number of active subscribers is one aspect of the system state, the current rate of subscription another. Granularity defines the level of detail an aspect of the system is measured at, e.g., the number of subscribers for any `temperatureEvent` versus the number of subscribers only interested in `temperatureEvents` for container #50 at confidence $\geq 78\%$). For that purpose, our approach allows each participant to define an individual level of granularity. Precision specifies the maximum degree to which the system state known at the participant is allowed to be inaccurate at any time by hiding smaller changes to the system state to the participant. For the same aspect of the system state, each participant might have an individual perception about what is significant or insignificant. Hence our approach supports different levels of granularity and precision for the same metric.

ASIA for Aggregated Feedback: Overview

The notion of *imprecision* is at the heart of our approach: participants can individually define what they consider to be *insignificant* changes they do not want to be informed about. For each monitoring metric they are interested in, a participant can specify an *imprecision* \hat{v} at runtime. Imprecision specifies how far the system state is allowed to vary from the most recent report of metric values to the participant. For example, a publisher wants to be notified only if the number of active subscribers for `temperatureEvent` has changed by more than ten subscribers compared to the last time the system state has been reported; smaller fluctuations in the population are not considered to be significant and the publisher does not want to be informed about them. This

relaxation is propagated throughout the network and is applied by every participating broker in the EBS, minimizing the number of update messages necessary.

Figure 3.26 illustrates the resulting trade-off: requests for monitoring data with high precision result in a large number of update messages as even minor changes are reported. In turn, higher imprecision reduces the number of aggregated feedback updates a participant receives from the system. However, this results in a coarse-grained representation of the system state.

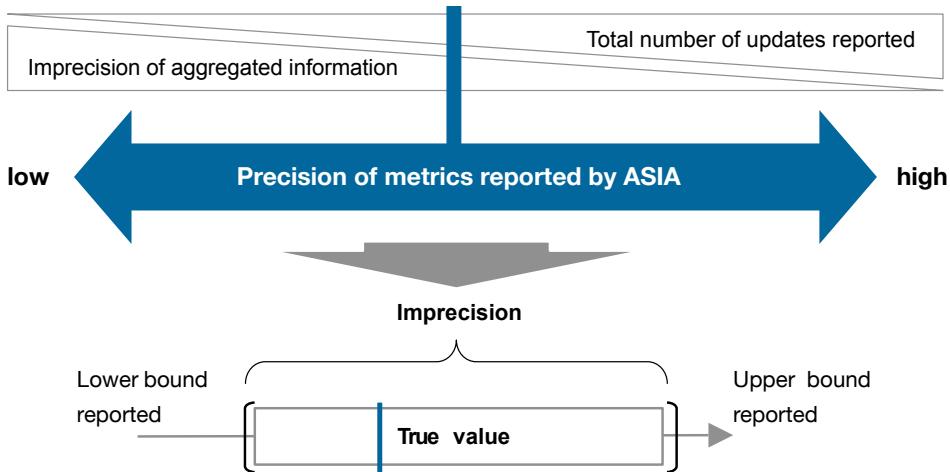


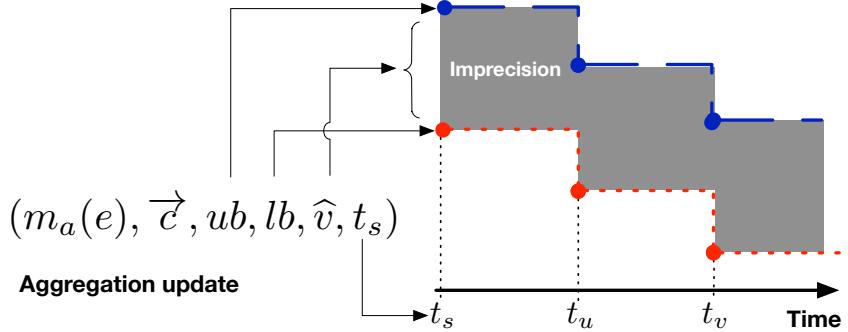
Figure 3.26.: ASIA imprecision trades data precision with costs for processing updates: upper and lower bounds of monitoring updates enclose the true value.

Definition 8 (Aggregated Feedback). Participants receive aggregated feedback as tuples $(m_a(e), \vec{c}, UB, LB, \hat{v}, t_s)$ with $m_a(e)$ the metric that represents the system state regarding a particular aspect at time t_s , LB the lower bound of the current interval the true value is contained in and UB the upper bound of the interval with length \hat{v} . The granularity of $m_a(e)$ can be increased by an optional list of constraints $\vec{c} = \{\hat{c}_1, \dots, \hat{c}_n\}$. Each constraint $\hat{c}_k \in \vec{c}$ is a tuple $(key, \triangleright, value)$ itself with \triangleright an operator $\triangleright \in \{<, \leq, =, \geq, >\}$ applied to value. \square

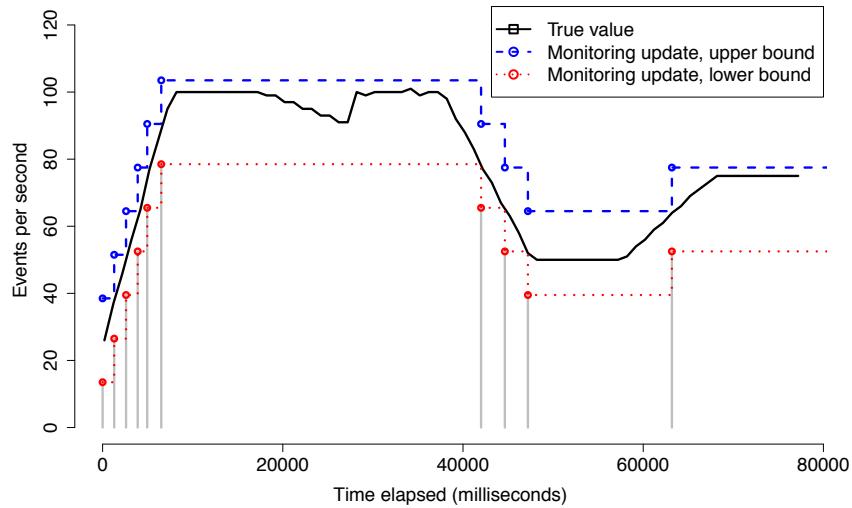
Aggregated feedback is delivered to all participants that have registered interest in a metric $m_a(e)$ about the system's population or state such as the examples defined in Equations (3.11) to (3.14). The granularity of metrics can be increased by a list of additional constraints \vec{c} . For example, $m_{publisherCount}(e)$ as defined in Equation (3.11) requested with $\vec{c} = \emptyset$ would return the total number of publishers providing notifications about events of type e without any constraints. Using $\vec{c} = \{('samplingRate', \geq, 40), ('trustworthiness', \geq, 'medium')\}$ instead keeps the subscriber informed only about the number of those publishers that provide notifications about events of type e at a rate of at least 40 notifications per second and have a level of trustworthiness of at least 'medium'.

As shown in Figure 3.27a, ASIA is not reporting a single value for each metric but an *interval* of values, defined by a lower and an upper bound. The size of the interval depends on the imprecision \hat{v} that has been chosen by the subscriber. Nevertheless, the interval always encloses the true value for $m_a(e)$ as shown in Figure 3.27b. Updates for $m_a(e)$ are sent to a participant only if the current state exceeds the boundaries of the interval that has been reported to that

participant before. Thus, participants do not have to process aggregation updates reflecting changes that are insignificant to them.



(a) Tuple representing aggregated feedback



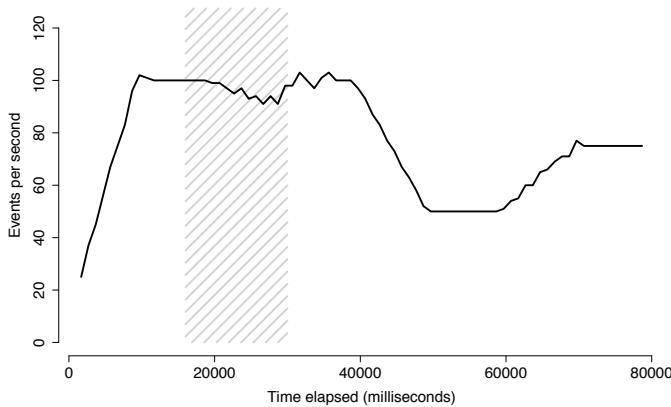
(b) Reported intervals always enclose v .

Figure 3.27.: Example for representing the sampling rate using ASIA aggregations.

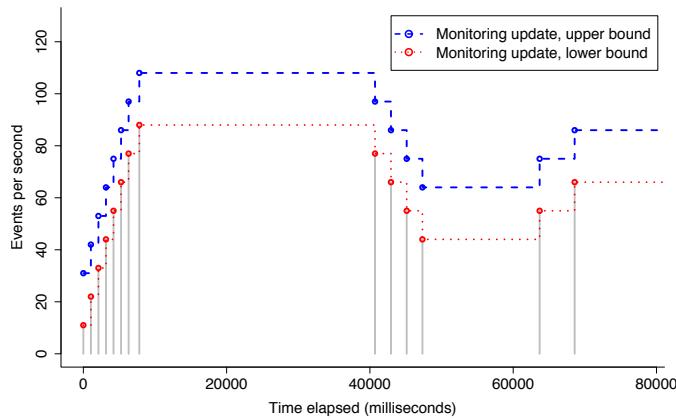
Example: Monitoring the Sampling Rate

We use an example to illustrate the impact of different levels of imprecision on the precision of the system state that is reported back to a participant. The sampling rate for a given type of notification changes dynamically over time as shown in Figure 3.28a. A subscriber wants to be updated about this but might not care about deviations of ± 20 notifications/second. At the same time, a publisher is also interested in updates about the sampling rate for notifications about the same type of event but does not care about deviations of ± 50 notifications/second. Consequently, the subscriber chooses an imprecision of $\hat{v}_s = 20$ while the publisher sets $\hat{v}_p = 50$.

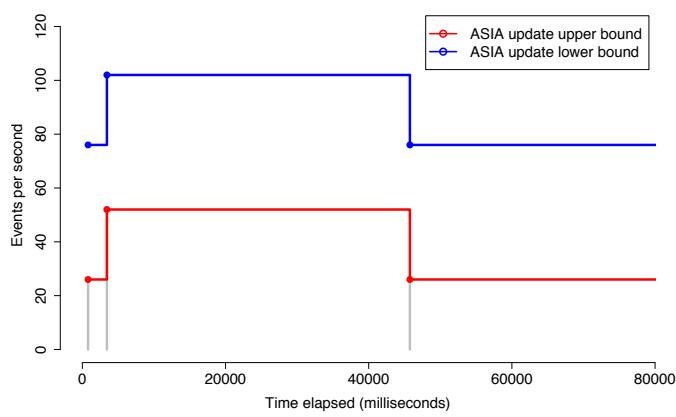
The aggregated feedback about the system state reported to both subscriber and publisher reflects the general rate distribution over time but hides changes that each participant has individually defined as being insignificant. Figure 3.28b illustrates the effect of $\hat{v}_s = 20$ on the results sent



(a) Example for a changing sampling rate



(b) Sampling rate of Figure 3.28a reported with $\hat{v} = 20$.



(c) Sampling rate of Figure 3.28a reported with $\hat{v} = 50$.

Figure 3.28.: Example for representing the sampling rate using ASIA aggregations.

to the subscriber representing the workload shown in Figure 3.28a: the temporarily declining sampling rate (grey area) is ignored as the fluctuation in that area is less than ± 20 events/second. The publisher is left with a very coarse grained representation of the system state as shown in Figure 3.28c as the chosen imprecision $\hat{v}_p = 50$ hides most of the dynamics.

Referring to the example shown in Figure 3.28a, the participant receives 14 updates when using an imprecision of 20 as shown in Figure 3.28b; choosing an imprecision of 50, however, would result in only 3 updates sent to the participant for the same workload as shown in Figure 3.28c. The received updates can be forwarded to analytical systems for further aggregation, displayed on dashboards for ITSM as shown in Figure 3.29, or used for self-adaptation [173, 177].

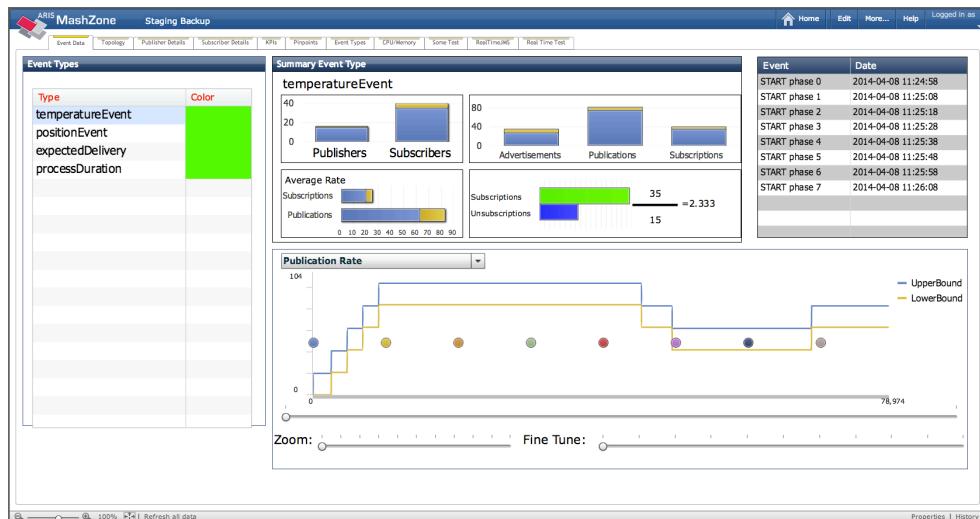


Figure 3.29.: Example for a dashboard to support runtime governance of EBS [177].

3.6 Summary

In this chapter, we have presented the core components of our model to express requirements about and support for QoI in an EBS: expectations, capabilities and feedback.

We have discussed the relationship between QoI and other related concepts addressing runtime quality. We have shown that QoI is a key concept for any data-driven task as it bridges the gap between objectively measureable characteristics of a system and subjective preferences about these characteristics. We have established a taxonomy to distinguish objective characteristics from subjective preferences and have derived a generic format to model characteristics as generic properties. Based on this generic representation, we have introduced the notions of expectations and capabilities.

Expectations enable subscribers to map their individual preferences to malleable sets of requirements about generic properties, rank these sets with utility values and declare trade-offs between them. The MOM can exploit the explicit trade-offs and rankings when trying to satisfy an expectation while defining requirements in a malleable way simplifies the definition of a requirement. We have introduced the fidelity metric to quantify the conformance between the data provided by the EBS and the preferences of a single subscriber. Allowing subscribers to weigh their requirements on their own raises questions about incentives and misuse, i.e., how to restrain subscribers from misusing the system by weighing all of their requirements with very high utility values to get them satisfied. These questions touch on general issues about incentives and a common

approach is to combine rankings with prices to be paid by the requester [381]. For this dissertation, however, incentive mechanisms and ways to prevent misuse are out of scope. For further discussion about pricing models and incentive mechanisms for networked systems, we refer to [107, 144, 286, 383, 382, 424].

Support for generic properties is formalized by the notion of capabilities we have introduced to capture the actual and potential support a publisher can offer for a generic property. The MOM uses capability profiles to decide if an expectation can be satisfied and assess the degree of adaptation required. This decision process will be discussed in detail in the following chapter.

Feedback is essential for self-adaptation at runtime to satisfy expectations or optimize the overall system utilization. We have introduced different types of feedback provided by the MOM at runtime as part of our approach. We have discussed how participants can adjust their behavior at runtime based on aggregated feedback about the system state or individual feedback about their expectations and capabilities.

4 Runtime Negotiation and Enforcement

We have introduced expectations, capabilities and feedback in the previous chapter. We have shown how subscribers can manage their individual Quality of Information (QoI) requirements at runtime using expectations; publishers use capabilities to expose their current state as well as their abilities to support QoI properties to the Message-oriented Middleware (MOM) at runtime. Participants can use the feedback provided as part of our approach to adapt at runtime.

In this chapter, we describe the algorithms and concepts used at runtime by the MOM to negotiate expectations with capabilities and decide on how to enforce which QoI requirements. As shown in Figure 4.1, runtime support for QoI requirements in our approach combines negotiation with adaptation at runtime following the Observe-Orient-Decide-Act (OODA) cycle¹ [70, 330].

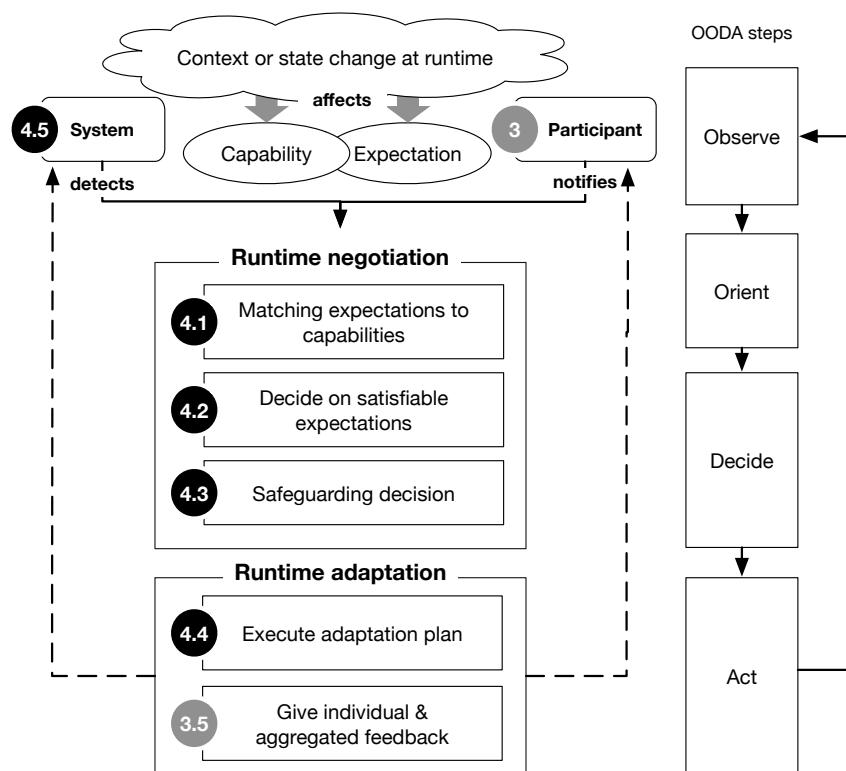


Figure 4.1.: Chapter overview: runtime support for QoI following the OODA cycle.

In the *observe step*, any changes that affect capabilities or expectations are detected – either using runtime monitoring in a self-aware system or actively announced by participants. In the *orient step*, the MOM has to identify the impact of the detected changes on the current state of the system while the *decide step* is about deciding on how to react; the *runtime negotiation* phase in our approach combines both steps. Finally, the reached decision is enforced in the *act step*; in our approach, this *runtime adaptation* phase consists of executing the adaptation plan and giving feedback to participants.

¹ Also referred to as monitor-analyze-plan-execute (MAPE) feedback loop [12].

The whole process of runtime negotiation and runtime adaptation is performed for a single expectation at a time. It can be triggered by a subscriber registering/updating an active expectation or by changes to a capability profile. A changed capability profile requires checking all expectations affected by it. The latter is the case if a publisher revokes/updates a capability profile or if the system state changes and affects a capability.

In Section 4.1, we present algorithms for matching expectations to capabilities in order to identify satisfied, satisfiable and unsatisfiable requirements. An expectation is *satisfied* if the system already delivers data with conforming QoI properties, or it is considered to be *satisfiable* if it would become satisfied after adapting the system. Otherwise, it is deemed *unsatisfiable*.

In Section 4.2, we focus on satisfiable expectations as adapting the system to satisfy them comes at additional costs. We discuss strategies to decide on whether to satisfy an expectation or decline it as unsatisfied.

In Section 4.3, we discuss algorithms to safeguard the set of possible adaptations resulting from the first two steps of the negotiation phase. Here, safeguarding refers to making sure that adapting the system to satisfy a dedicated expectation does not violate other requirements or constraints imposed by the MOM. The steps discussed in Sections 4.2 and 4.3 could be combined as they both address optimization and load-balancing aspects. We discuss them in separation to distinguish between deciding on satisfiable expectations and safeguarding the decision.

In Section 4.4, we show how decisions are enforced at runtime by triggering self-adaption of the MOM and advising publishers to adapt using individual feedback. We show examples for how requirements for the different categories of generic properties can be enforced by publishers and the MOM – either in isolation or by coordinated adaptation.

In Section 4.5, we present the concepts used in our approach to monitor the system state at runtime and provide participants with aggregated feedback about the population and dynamics of the Event-based System (EBS). Details on how participants inform the MOM about changes to their requirements or capabilities have already been discussed in Sections 3.3 and 3.4 while the types of individual and aggregated feedback have been covered in Section 3.5.

Finally, we discuss different types of conflicts that could arise at runtime in Section 4.6. We describe examples for each category of conflicts and how they are resolved in our approach.

4.1 Matching Expectations to Capabilities

Requirements negotiation in an EBS can be done automatically at runtime inside the MOM. The objective is to identify three sets of expectations: those that are already *satisfied* with the current state of the system and those that are currently unsatisfied but generally *satisfiable* by adapting the system. In some cases, however, a requirement cannot be satisfied due to limitations of the system or cost constraints. In this case, it has to be declined as *unsatisfiable*.

The first step of the runtime negotiation phase is about determining whether an expectation can be satisfied by the system at all. Thus, we have to match the set of requirements contained in an expectation against the capabilities currently available from publishers. As shown in Figure 4.2, this step can be broken down into three tasks.

First, the MOM has to identify those publishers that support all non-interdependent generic properties required by the expectation in a set-matching step. Second, for each of the associated capability profiles, we have to match the ranges of its capabilities against the required

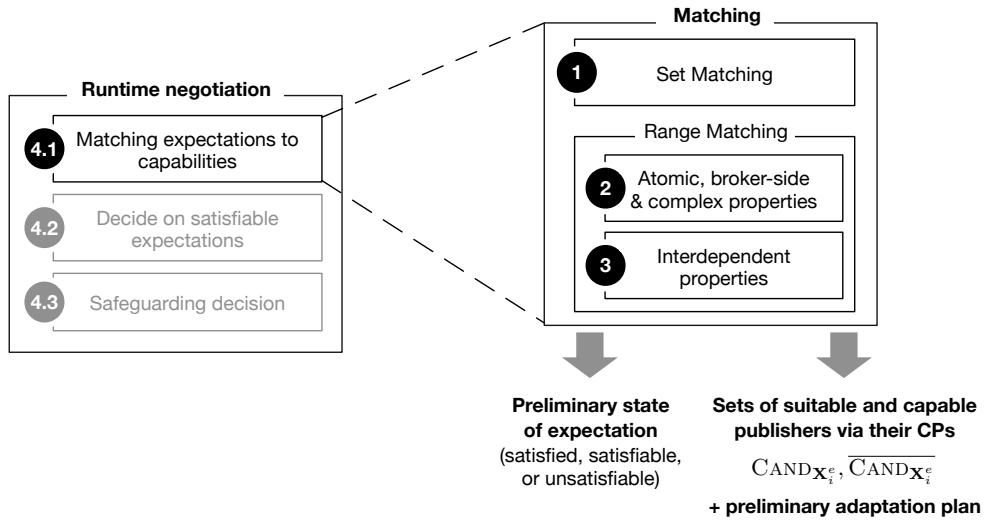


Figure 4.2.: Matching an expectation against the system state: set- and range-matching to identify the preliminary state of the expectation and those publishers that are able or capable to satisfy the expectation.

ranges defined in the expectation. For this task we have to check ranges that are defined over atomic, broker-side, or complex generic properties² first before matching the ranges of those requirements defined over interdependent generic properties.

This first step of the runtime negotiation phase results in a preliminary state of the expectation (satisfied, satisfiable, or unsatisfiable) together with two sets of capability profiles: one set of capability profiles that already satisfy the expectation ($CAND_{X_i^e}$) as well as one set of capability profiles associated with publishers that are capable in principle but would have to adapt ($\overline{CAND}_{X_i^e}$). We maintain both sets for each expectation X_i^e together with a preliminary adaptation plan.

4.1.1 Set-Matching to Find Suitable Capability Profiles

In the first step of the matching phase, the MOM has to find a set of publishers that are able to satisfy the expectation. As publishers are represented by their capability profiles, we have to find a set of suitable capability profiles. A capability profile CP_l^e is *suitable* if matching capabilities for all generic properties defined in the expectation can be found in CP_l^e . We call the resulting set of capability profiles we can match X_i^e against *nominees* ($NOM_{X_i^e}$). By definition, interdependent generic properties, such as *alternatives*, cannot be provided by a single publisher. Hence, they cannot be part of a single capability profile and are not considered during set matching.

The algorithm that determines the individual nominees for an expectation X_i^e based on a set of capability profiles $\{CP_1^e, \dots, CP_n^e\}$ is shown in Algorithm 2 in the appendix. For each CP_l^e that is registered for the same type of notification e as X_i^e , we check whether it supports at least³ all non-interdependent generic properties defined in X_i^e (line 3). This check is done using the mapping function $\Theta(p_k)$ (line 4) which returns a capability C_k^e that matches $p_k \in X_i^e$ in syntax and semantics as described on page 45. The algorithm returns $NOM_{X_i^e} = \{CP_l^e \in NOM_{X_i^e} \subseteq \{CP_1^e, \dots, CP_n^e\} \mid \forall p_k \in X_i^e \wedge \neg \text{isInterdependent}(p_k). \Theta(p_k) = C_k^e \in CP_l^e\}$.

² See Section 3.4.1 and table 3.6 for more details and examples on each type of generic property.

³ Please note that this includes capability profiles that offer support for a larger set of generic properties as defined in the expectation, i.e., $X_i^e \subset CP_l^e$

4.1.2 Range-Matching for Each Generic Property

Having identified a set of publishers that support at least all non-interdependent generic properties defined in X_i^e , we have to check whether the extent of the offered support suffices to satisfy the requirements. This is done by matching the range defined in each requirement against the range and current value of the matching capability in each capability profile $CP_l^e \in Nom_{X_i^e}$.

Figures 4.3 and 4.4 illustrate the different types of relationships between a capability and different requirements about generic properties with maximizing or minimizing improvement directions: first, the *current value* of the capability could *dominate* the lower bound (Figure 4.3 A,B,D) or upper bound (cf., Figure 4.4 G,H,J) of a requirement; it could even be *enclosed* by the upper and lower bounds defined in a requirement (cf., Figure 4.3 B and Figure 4.4 G).

In addition, the range defined in a requirement could *overlap* with the *range* defined in the capability as shown in Figure 4.3 (A,B,C) and Figure 4.4 (F,G,H). Alternatively, the ranges can be *disjoint* as shown in Figure 4.3 (D,E) and Figure 4.4 (I,J).

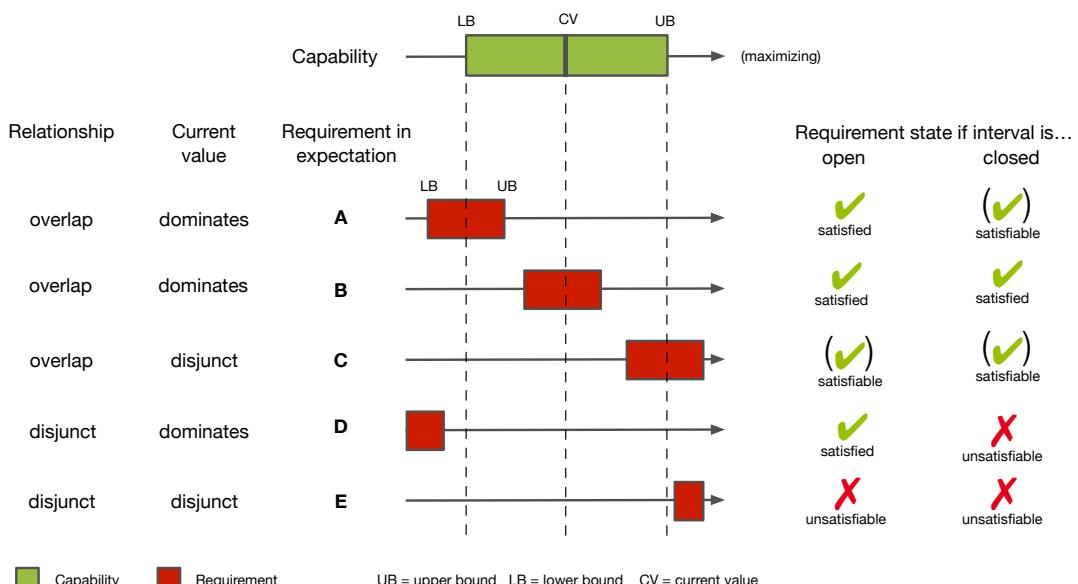


Figure 4.3.: Maximizing improvement direction, e.g., precision: relationship between a capability and a requirement with open/closed interval.

Determining the kind of relationship between a capability and a requirement is crucial to determine whether the requirement is satisfied or at least satisfiable.

The results differ depending on whether we assume requirements to be defined over open or closed intervals: requirements with closed intervals as shown in Figures 4.3 and 4.4 are only *satisfied* if the current value of the capability is enclosed by the lower and upper bounds defined in the requirement (B,G).

For requirements with open intervals, however, it is sufficient for the current value to *overfulfill* the lower bound of the requirement. For example, see Figure 4.3 (A,B,D) for a maximizing improvement direction, such as precision, and Figure 4.4 (G,H,I) for a minimizing improvement directions, such as latency. The difference between open and closed intervals becomes most apparent for the case where the ranges defined in the capability overfulfill both upper and lower

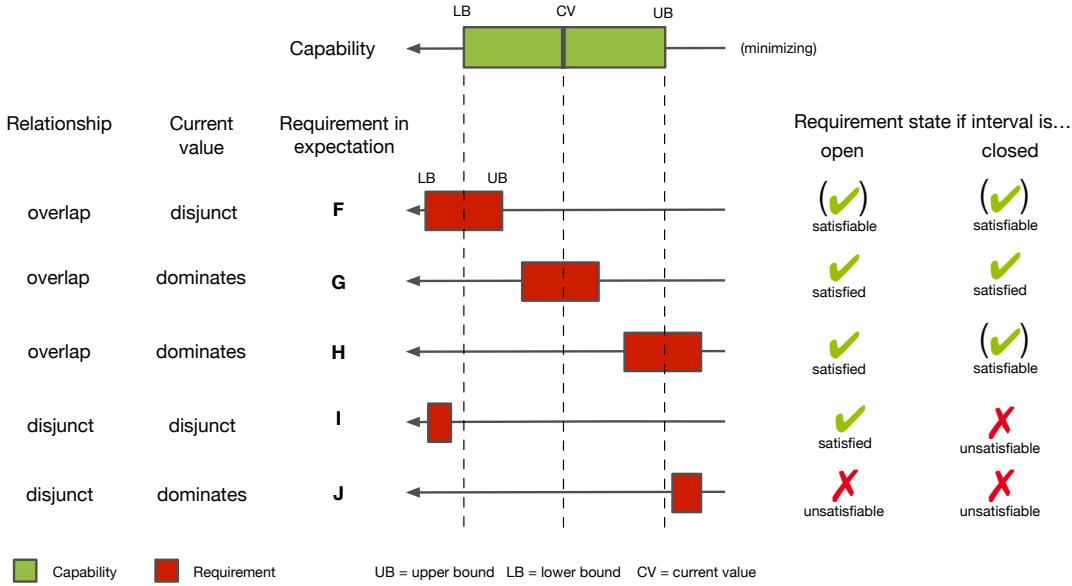


Figure 4.4.: Minimizing improvement direction, e.g., latency: relationship between a capability and requirement with open/closed interval.

bounds of the requirement (D,J): for open requirements, this means that the requirement is *satisfied* while it would be *unsatisfiable* for closed requirements as the current value of the capability could never be adapted to satisfy the requirement. Contrastingly, requirements C and F are not yet satisfied but satisfiable while E and J are unsatisfiable.

Definition 9 (Relationships capability and generic property). Generalizing the relationships between capabilities and properties, we define the following terms for the requirement about a generic property p_k of an expectation X_i^e and a matching capability C_k^e of a capability profile CP_j^e :

Satisfied requirement. The requirement about a maximizing generic property defined over an open interval is satisfied if a matching capability's current value dominates the lower bound of the property: $C_k^e.CV \geq p_k.LB$. Respectively, a requirement defined over a generic property with minimizing improvement direction is satisfied if the upper bound of the requirement is dominated by the capability's current value: $C_k^e.CV \leq p_k.UB$. A requirement defined over a closed interval is only satisfied if the capability's current value is enclosed by the lower and upper bounds of the requirement (i.e., $p_k.LB \leq C_k^e.CV \leq p_k.UB$).

Overfulfilled requirement. A requirement is overfulfilled if its complete range is dominated by the current value of the corresponding capability: $C_k^e.CV > p_k.UB$ for maximization and $C_k^e.CV < p_k.LB$ for minimizing improvement directions. While requirements defined over an open interval are satisfied by overfulfilling capabilities, requirements defined over closed intervals are unsatisfied. In some cases, the ranges do not even overlap: $C_k^e.LB > p_k.UB$ for maximization and $C_k^e.UB < p_k.LB$ for minimizing improvement directions.

Covered requirement. A requirement is covered if its range of accepted values overlaps with the range of a matching capability: $C_k^e.LB \leq p_k.UB \wedge C_k^e.UB \geq p_k.LB$.

The key steps of the range-matching algorithm we execute per generic property $p_k \in X_i^e$ are illustrated in Figure 4.5; the full algorithm is given in pseudocode in Algorithm 4 in the appendix.

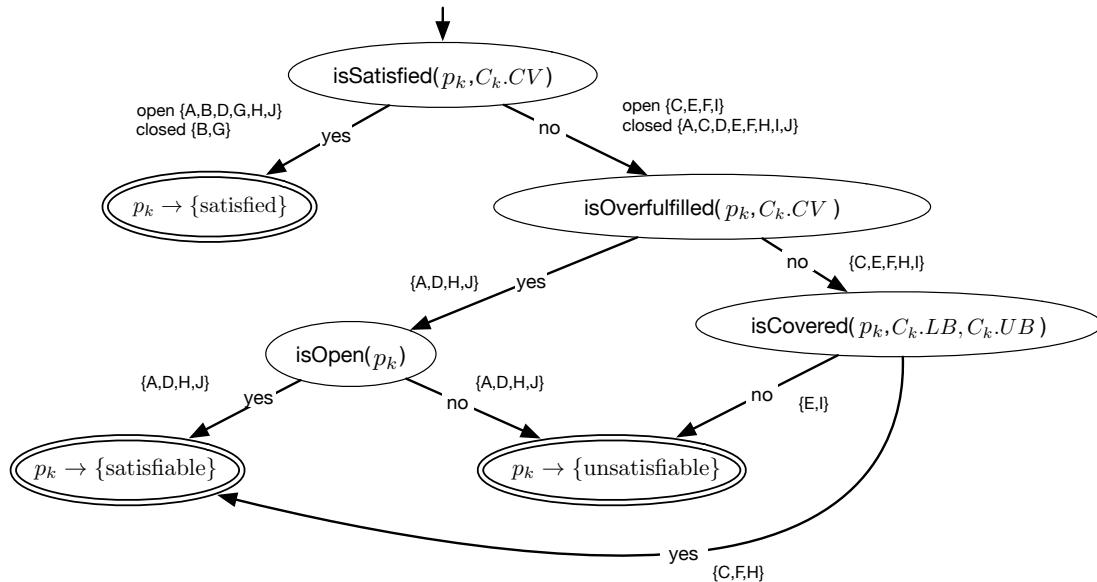


Figure 4.5.: Range-matching performed in Algorithm 4 and Algorithm 5 for each property to check if the requirement is satisfied or satisfiable; letters refer to the types of relationships shown in Figures 4.3 and 4.4.

First, we check if the requirement is satisfied by the current state of the capability. For a requirement defined over a closed interval, the current value of a matching C_k^e has to be within the interval described by $p_k.LB$ and $p_k.UB$ – regardless of whether the improvement direction is maximizing or minimizing; for open intervals, it is sufficient for the current value to be greater or equal $p_k.LB$ for maximizing properties (and smaller or equal than $p_k.UB$ for minimizing, respectively). In this case, we can add p_k to the list of generic properties already satisfied by this capability profile.

We first execute this algorithm for all requirements about non-interdependent generic properties defined in X_i^e (line 6) and each capability profile $CP_l^e \in \text{Nom}_{X_i^e}$ (line 3). The results are two sets of capability profiles: one that contains all capability profiles which already satisfy the expectation ($\text{CAND}_{X_i^e}$) and one that contains all that would satisfy it after adapting ($\overline{\text{CAND}}_{X_i^e}$).

In a subsequent step, we run the same checks shown in Figure 4.5 for each requirement in X_i^e that is defined over an interdependent generic property.

After checking all requirements about non-interdependent generic properties for all capability profiles in $\text{Nom}_{X_i^e}$, we proceed to check all remaining requirements about interdependent generic properties. For each interdependent generic property we have to execute the algorithm that checks whether a requirement is satisfied, satisfiable or unsatisfiable. The algorithm that has to be applied depends on the semantics of each interdependent generic property.

We use the example of *alternatives* (cf., Algorithm 5 in the appendix) to illustrate the general approach. While this process is the same sequence of checks as shown in Figure 4.5, we now have to use current values that are computed by the MOM based on the number of capability profiles in $\text{CAND}_{X_i^e}$ and $\overline{\text{CAND}}_{X_i^e}$. The requirement is *satisfied*, if there are enough publishers satisfying all other requirements about non-interdependent generic properties: $|\text{CAND}_{X_i^e}| \geq p_{alt}.LB$ (line 1). If the number of capability profiles already satisfying all requirements about non-interdependent

generic properties is not yet sufficient, we have to take into account those capability profiles that are capable of satisfying the remaining requirements with adaptation: $|C_{AND_{X_i^e}}| + |\overline{C}_{AND_{X_i^e}}| \geq p_{alt} \cdot LB$ (line 6). In case of the requirement being defined over a closed interval with a maximum number of alternatives, we also have to check if the number of publishers already satisfying all other requirements does not exceed this upper bound: $p_{alt} \cdot LB \leq |C_{AND_{X_i^e}}| \leq p_{alt} \cdot UB$. Otherwise, the requirement's state is set to *satisfiable*.

4.1.3 Determining the Preliminary State of an Expectation

Having performed the first part of the runtime negotiation phase, we can already determine a preliminary state of the expectation based on the results of the set and range-matching steps.

As long as there are no requirements about interdependent generic properties defined, an expectation X_i^e is considered to be *satisfied* if there is at least a single capability profile that is already satisfying X_i^e , i.e., $C_{AND_{X_i^e}} \neq \emptyset$ (cf., Algorithm 4 lines 26-29). If there are requirements about interdependent properties — such as alternatives — the number of capability profiles satisfying all other requirements of X_i^e has to be at least the minimum number required for this interdependent property (cf., Algorithm 5 lines 4-7) to render an expectation *satisfiable* or *satisfied*.

The state of an expectation X_i^e is the conjunction of each requirement's state as shown in Figure 4.6: an expectation is *satisfied* by a capability profile if and only if the requirements about *all* generic properties are already satisfied by the capability profile (A).

An expectation is *satisfiable* (B) by a capability profile if there is at least one requirement about a generic property that is not yet satisfied by this capability profile but could be satisfied (e.g., b) while all other requirements are already satisfied.

An expectation is *unsatisfiable* (C) by a capability profile if there is at least one requirement that cannot be satisfied by the capability profile (e.g., c) – even if the requirements about all other generic properties can be satisfied or are already satisfied.

Please note that these deductions are independent of each requirement being defined over open or closed intervals as this characteristic; the *isSatisfied* method shown in Figure 4.5 and algorithm 3 considers the type of interval.

Capability Profile	Expectation: contained Requirements			Expectation State	Initial adaptation plan
	a	b	c		
A				satisfied	→ routeMessagesFrom(A)
B		() satisfiable		satisfiable	→ all actions available for adapting B.b to satisfy requirement b
C				unsatisfied	→ nothing, as capability profile C does not satisfy the expectation

Figure 4.6.: State of an expectation (satisfied, satisfiable or unsatisfiable) depending on the capability profile that is matched.

The range-matching step populates an initial adaptation plan as shown on the right hand side of Figure 4.6 in addition to the sets of suitable and capable publishers (cf., Algorithm 4 lines 22+28).

In case of a satisfied expectation, we have to make sure that the subscriber associated with the respective expectation receives notifications from only those publishers that already provide notifications that conform to the expectation. Thus, we add a `routingAdaptation` action (cf., Section 3.2.2) to the adaptation plan for each publisher P_l that is associated with a capability profile $CP_l^e \in CAND_{X_i^e}$ (cf., Algorithm 4 line 28).

For every capability profile that could satisfy the expectation, we need to adapt the system first before we can acknowledge the expectation to be satisfied and route notifications to the subscriber. From the range-matching step described in the previous section we know which capabilities to decrease or increase for which capability profile. We use this knowledge here to select appropriate actions for each satisfiable generic property and add them to the preliminary adaptation plan (cf., Algorithm 4 lines 16+22).

For example, an expectation requires the generic property *sampling rate* provided by a publisher to be decreased to 40 notifications per second. This can be done at the publisher by adjusting the sampling rate (*publisherAdaptation*), or at the MOM by reducing the sampling rate delivered to the subscriber (*rateControllerAction*). Let us also assume that the adaptation cost for a *publisherAdaptation* action is 10 while the adaptation cost for *rateControllerAction* is 2. In this case, we add instances of both types of actions with their respective costs to the initial adaptation plan.

Thus, the preliminary adaptation plan can contain a mix of routing adaptations and other actions. This is the case whenever there is at least one capability profile that is already satisfying the expectation but others are only capable. If the expectation contains requirements about interdependable generic properties such as *alternatives*, we do adapt the routing only if there are enough satisfying capability profiles so that at least the minimum number of alternative publishers is satisfied. Otherwise, we first execute all adaptation actions that increase the number of suitable publishers before we adapt the routing and acknowledge the expectation to the subscriber as being satisfied.

4.2 Deciding on Satisfiable Expectations

The range-matching step discussed previously marks an expectation as *satisfiable* if the system could satisfy the contained requirements by adapting certain capabilities. Mechanisms for adapting the system have their own costs in addition to the costs for operating capabilities at a certain level. Thus, the MOM has to assess the expected costs and adapt if the expected benefits outweigh the calculated costs [79, 80, 381].

Whether a satisfiable expectation should be satisfied or not is an abstract decision problem to be solved by the MOM. While every decision problem can be decomposed into a decision tree with objectives, goals and attributes [249, 424], their nature and hierarchy depend on the preferences of the decision maker. Thus, our model allows to express such preferences from the perspective of the MOM that acts as the *decision maker*⁴. These preferences are orthogonal to the preferences articulated by subscribers and formalized in expectations. We follow the definitions of Keeney [249] in that *objectives* describe outcomes that can be achieved to a certain degree while *goals* describe outcomes that are achieved either completely or not.

Examples for objectives in the context of QoI in EBS:

- Maximize total utility.
- Maximize utility per subscriber.
- Maximize generated fidelity per subscriber.
- Maximize total number of subscribers with satisfied expectations.
- Minimize adaptation costs.
- Minimize operational costs.

Respectively, examples for goals are:

- Every subscriber has at least one satisfied expectation.
- No subscriber is put into disadvantage by an adaptation.
- No publisher is responsible for more than 40% of all satisfied expectations.

The preferences defined by the decision maker can consist of a single objective or of multiple objectives and goals that can be conflicting due to limitations of the system.

In case that the preferences of the decision maker require the MOM to optimize multiple objective functions at the same time, the decision problem becomes an optimization problem. The resulting optimization problem consists of a set of attributes (here: capabilities to adapt, utility generated by an expectation, etc.), a set of objective functions that formalize objectives and goals as well as a set of constraints [437]. In general, the optimization problems to solve are min-max or max-min problems, i.e., some target function has to be maximized subject to several auxiliary functions that need to be minimized or vice-versa; a feasible solution for such a problem does not violate any constraints [297].

By abstracting the decision about a satisfiable expectation as an optimization problem, we can rely on the huge body of work done for decades about optimization approaches in the area of Multi-Objective Optimization Problem (MOOP) and Multi-Criteria Decision Making (MCDM). However, the computational complexity of MOOPs is NP-hard if attributes require integer solutions [83, 161], e.g., the number of alternative publishers to be chosen. Thus, heuristics and

⁴ In practice, we assume these preferences to be defined by the party deploying and operating the MOM.

relaxing assumptions are usually used in practice to find approximate (weak pareto-optimal) solutions [36, 79, 80, 346, 381].

We summarize the general types of optimization approaches that build on preferences of the decision maker and refer the interested reader to [83, 133, 151, 270, 272, 283, 297, 437] for a more detailed presentation of the different approaches.

In general, three categories of approaches can be distinguished based on the point in time when the preferences of the decision maker are defined [83, 133, 270, 437]:

1. *A priori*: the preferences of the decision maker are known beforehand. In this case, the objectives and goals can be aggregated into a single objective function, e.g., using scalarization, ϵ -constraints, goal programming, lexicographical ordering, weighted global criteria, or weighted sums; if the preferences establish a meaningful hierarchy of objectives, multi-level programming can be used as well.
2. *A posteriori*: the preferences of the decision maker are not known beforehand. Thus, different alternative solutions are provided to the decision maker to choose from.
3. *Interactive*: the decision maker is included into the optimization process itself. At each step, preferences can be defined that establish a hierarchy of objectives and narrow down the solution space.

None of the above approaches is superior as such: the information available, the types of preferences as well as system limitations determine feasible approaches for a specific optimization problem [297].

Runtime negotiation in our model does not require a specific optimization approach to decide on satisfiable expectations. Our model allows any custom approach to be used based on the deployment scenario: *how* the decision is reached is encapsulated and transparent for all other steps of the runtime negotiation phase.

4.2.1 Decision Strategies Encapsulating the Decision Process

In our approach, a *decision strategy* encapsulates an individual decision tree of minor and major objectives or goals together with their attributes and optimization approach (optimal algorithms, prediction models, or heuristics). We assume that the party deploying or operating the MOM provides at least one decision strategy. In the remainder of this section we will discuss an example for a simple decision strategy to illustrate the general principle.

A decision strategy encapsulates the two steps shown in Figure 4.7: calculating costs and deciding on whether the expected benefits outweigh these costs. For a single satisfiable expectation, applying a decision strategy results in a final state (*satisfied* or *unsatisfied*) for that expectation as well as an updated adaptation plan.

Load-balancing approaches can be included when calculating the costs for adaptation, e.g., to include resource utilization as a weighting factor; or it can be included in the second step when deciding on whether the benefits outweigh the costs. For example, using the FIT score [169] to weigh the total costs for adapting a publisher can be used to avoid overloading publishers.

Including or excluding those considerations, however, depends on the preferences of the decision maker or the party defining a decision strategy.

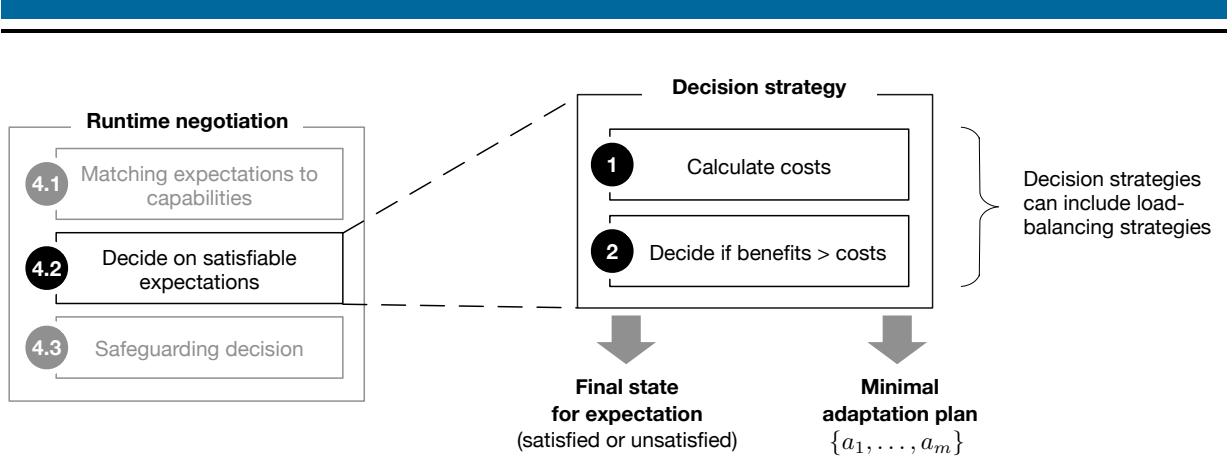


Figure 4.7.: Two-step approach to decide whether to satisfy a satisfiable expectation.

4.2.2 Example Strategy: First-Come, First-Served while Minimizing Costs

A valid decision strategy does not necessarily have to include aspects of multi-objective optimization. For example, it could be based on a First-come, First-served (FCFS) policy just as well: the MOM tries to satisfy expectations in the chronological order they are registered or updated. An expectation $X_{i,k}^e$ by subscriber S_i should only be declined if the cost for providing it exceeds the total cost of an already satisfied expectation $X'_{i,s}^e$ by the same subscriber.

Algorithm 1 shows a simple heuristic to implement this strategy for each subscriber S_i :

1. Select minimum adaptation plan to satisfy $X_{i,k}^e$ (line 1).
2. Calculate total costs expected for satisfying $X_{i,k}^e$ (line 4).
3. Calculate total costs for the currently satisfied $X'_{i,s}^e$ (line 3).
4. Satisfy $X_{i,k}^e$ if satisfying it is less expensive than operating $X'_{i,s}^e$ (line 6).

Algorithm 1: Decide if a satisfiable expectation should be satisfied with the example strategy.

```

Function decideOnSatisfiable( $X_{i,new}^e$ , ADAPTATIONPLAN $X_{i,new}^e$ ) is
    Result: minimal ADAPTATIONPLAN $X_i^e$ 
    ADAPTATIONPLAN $X_{i,new}^{min}$  ← getAdaptationPlanMinimalCosts( $X_i^e$ , ADAPTATIONPLAN $X_{i,new}^e$ )
    costsnew ← getTotalCosts(ADAPTATIONPLAN $X_{i,new}^e$ )
     $X_{i,current}^e$  ← getCurrentSatisfiedExpectation( $i$ )           // Get currently satisfied expectation
    costscurrent ← getTotalCosts( $X_{i,current}^e$ )
    state ← unsatisfied
    if costsnew < costscurrent then                                // Does it cost less on the long run?
        state ← satisfied
    else
        ADAPTATIONPLAN $X_{i,new}^e$  ← ∅
    return ADAPTATIONPLAN $X_{i,new}^e$ 

```

The total costs for satisfying an expectation in steps 2 and 3 include the costs for executing all necessary adaptations as well as the costs for a publisher to continuously provide notifications with certain capabilities. Thus, we do a rough break-even analysis in step 3 by calculating the operational costs for providing δ notifications that conform to $X_{i,k}^e$ or $X'_{i,s}^e$. In case that there is no other expectation satisfied for S_i so far, the costs for $X'_{i,s}^e$ are set to ∞ , automatically resulting in the decision to satisfy $X_{i,k}^e$.

4.3 Safeguarding the Decision

In the runtime negotiation phase, a single expectation is negotiated at a time. This negotiation results in a preliminary adaptation plan for the expectation. The adaptation plan contains all actions that have to be executed for a set of capability profiles to satisfy the expectation: changes to the routing table for those publishers that have capability profiles which already satisfy the expectation; other actions for those capability profiles that require adaptation. The set of actions is empty if the expectation is declined or cannot be satisfied even with adaptation as described in Sections 4.1 and 4.2.

The last step of the runtime negotiation phase takes care that executing actions does not violate constraints. Constraints stem from requirements defined in expectations as well as local or global constraints imposed by the MOM due to budgets or load-balancing considerations. We refer to this process as *safeguarding*.

Three examples illustrate such constraints:

- Advising one publisher to increase the sampling rate to satisfy the expectation currently under negotiation might violate requirements about sampling rate defined in already satisfied expectation.
- Advising several publishers to increase their sampling rate in order to satisfy a requirement about alternatives might saturate a network link or broker and violate requirements about latency and completeness [173].
- While the MOM can apply content aggregation or traffic shaping to satisfy requirements about the latency of notifications, effectively reducing the number of processed notifications can violate requirements about the sampling rate [138].

During the safeguarding process, we try to detect such violations of constraints and adjust the adaptation plan accordingly. We decline the expectation as unsatisfiable if severe violations cannot be avoided without emptying the adaptation plan. The feasibility and complexity of the whole process depends on the custom decision strategy that is used.

In general, simulating the effect of adaptations before actually executing them or predicting their impact on local or global constraints should be part of this process. However, this requires complex performance and system models for EBSs and Distributed Event-based Systems (DEBSs) to capture cause-effect relationships and make proper predictions. This actively pursued field of research is out of scope of this dissertation and we refer the interested reader to [72, 202, 258, 305, 311, 367, 388, 410] for further information.

General Safeguarding Approach

In the remainder of this section, we describe the general safeguarding approach and show where custom performance models and load-balancing approaches can be integrated. The sequence of operations in our safeguarding algorithm is shown in Figure 4.8.

Safeguarding starts with parsing the initial adaptation plan that has been compiled during the range-matching step described in Section 4.1.2 and the decision step in Section 4.2.

We rank all capability profiles affected by the adaptation plan according to a custom ranking function. Starting with the top-ranked capability profile, we iteratively check whether the actions defined for each capability profile would violate any global or local constraints.

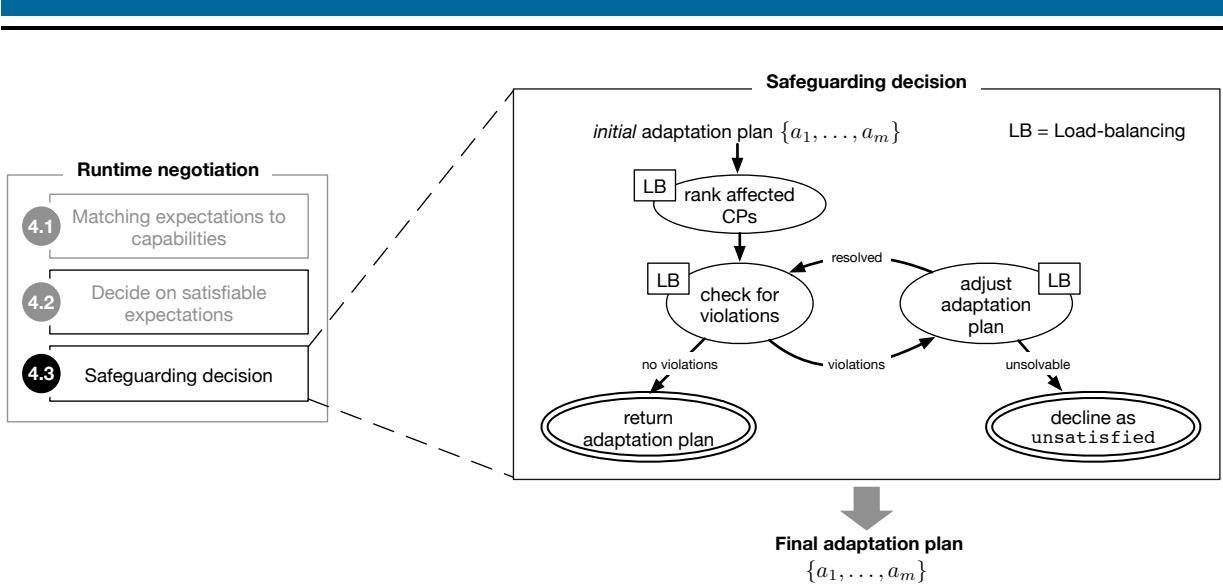


Figure 4.8.: Steps to safeguard the adaptation plan.

If we detect such violations, we try to adapt the adaptation plan. For example, we could try to use alternative actions as those currently defined or we could exclude the current capability profile from the list. Finally, we end with a safe adaptation plan that we proceed with as the final adaptation plan. The safeguarding algorithm rejects the expectation as *unsatisfied* if the violations cannot be resolved. In this case, the final adaptation plan would be empty.

Integrating Load-balancing

Custom algorithms for load-balancing can be integrated in all steps of the safeguarding process to distribute the load between different publishers or brokers. The ranking function allows for encapsulating load-balancing aspects first and foremost. For example, the capability profiles could be ranked based on their adaptation costs in ascending order to avoid expensive publishers to be chosen first. Alternatively, ranking could be based on the utilization of the publishers associated with each capability profile the action is targeting. Using the FIT-score [169], ranking could be done in ascending order to choose less critical publishers first (as a high FIT score indicates a high criticality) while ranking in descending order would start with more critical publishers. Another possibility would be to rank publishers based on their trustworthiness score in descending order. Alternatively, load-balancing can be included when checking for violations or it can be used during the purging step in a similar way as during the initial ranking step.

Please note the difference between load-balancing in this section and load-balancing as part of the decision mechanism discussed in Section 4.2. There, we sketched out how load-balancing considerations can be included into the decision process of deciding *whether* the system should adapt to enforce satisfiable requirements or not. By contrast, including load-balancing here aims at *how* the expectation is going to be satisfied.

Example: Detecting and Resolving Violations for Alternatives

We illustrate the safeguarding process by describing how we avoid violations of the *alternatives* requirement when assuming the simple decision strategy we describe in Section 4.2.2.

When defined over a closed interval, the alternatives requirement can be violated by routing notifications from too many publishers to a subscriber i . This can be avoided by checking the size of $|\text{CAND}_{X_i^e}|$ and $|\overline{\text{CAND}}_{X_i^e}|$: if the size of $\text{CAND}_{X_i^e}$ already exceeds the range of allowed values, we do not have to adapt any further capability profiles from this list. Rather, we have to limit the number of satisfying publishers that i receives notifications from. Conversely, if the sum of satisfying and capable publishers exceeds the upper bound of the requirement, we have to limit the number of capability profiles in $\overline{\text{CAND}}_{X_i^e}$ to adapt⁵.

In both cases, all required information is available in the preliminary adaptation plan: routing actions for satisfying capability profiles and other actions for capable capability profiles.

Thus, the basic algorithm we use for this is as follows:

1. Rank all capability profiles in the adaptation plan according to a ranking function
2. Iteratively check and resolve violations for requirements about generic properties other than alternatives
3. Keep only the top- k capability profiles, delete all other from the adaptation plan

Here, k defines the upper bound defined for the requirement about alternatives if defined over a closed interval. This way, we keep only the maximum number of capability profiles allowed by the subscriber.

⁵ Please note that for load-balancing reasons we could also decide to adapt capable publishers first and deactivate publishers that already satisfy the expectation.

4.4 Runtime Adaptation

The last phase of the runtime support process is to adapt the system and give individual feedback to subscribers. While system adaptation is limited to routing adjustments for *satisfied* expectations, approved *satisfiable* expectations require further adaptation. For the sake of clarity, we focus on runtime adaptation to satisfy an expectation $X_i^e \in \overline{\text{SAT}}$; adaptation to free up resources or optimize system utilization is part of future work.

Runtime adaptation can be realized in three ways by:

1. adapting the MOM transparently for publishers and subscribers;
2. advising publishers to adapt; or
3. coordinating the adaptation of both MOM and publishers.

In our model, all these different types of adaptations are uniformly represented by the generic *actions* we have introduced in Section 3.2.2.

4.4.1 Middleware Self-Adaptation

Some generic properties can be influenced at the MOM using self-adaptation as discussed in Section 3.2.2. While the MOM can influence some generic properties only in one direction (e.g., decrease), others can be influenced in both directions (i.e., increased or decreased). Adapting the MOM transparently for participants is necessary when publishers are not able to adapt or if adapting them would violate constraints. In this section, we present three mechanisms to illustrate how self-adaptation by the MOM can be used to enforce requirements.

Adapt Routing

Routing adaptation refers to changing entries in the routing tables of a broker so that particular participants are permitted or excluded from receiving notifications. Routing adaptation in the context of our approach is used to ensure that a subscriber receives notifications only from those publishers that have capability profiles conforming to its requirements. Notifications from other publishers are not routed to the subscriber even though they match that subscriber's subscription in terms of notification type or content. This contrasts routing adaptation in a typical EBS or DEBS where routing tables are populated based on matching advertisements and subscriptions.

Having access to the broker state allows us to build upon the existing routing tree built and maintained by the broker. Routing adaptation increases or decreases the number of alternative publishers a subscriber receives notifications from.

Decrease Sampling Rate Received by Subscribers

Reducing the sampling rate of notifications based on a leaky bucket algorithm, a token bucket algorithm, load shedding, or content aggregation is a widely used mechanism in networked systems [40, 141, 152, 153, 158, 240, 337, 405].

Such a `rateController` can be used by the MOM to reduce the sampling rate sr_{in} of notifications to a lower rate sr_{out} that another participant receives them with. The reduced outgoing rate can be fixed or dynamically changing, depending on why this mechanism is used. While a fixed sr_{out} can be used to satisfy the requirements about sampling rate defined by a receiving subscriber, a

flexible outgoing rate allows the MOM to adjust the outgoing traffic. For example, to stay within a bandwidth budget [173], or to reduce the total number of notifications to be processed by the receiving broker, i.e., to free up processing resources there [138].

In this dissertation, we use *rateControllers* with both fixed and flexible sr_{out} to reduce the sampling rate of notifications of type e .

Reduce Forwarding and Path Latency of Notifications

The end-to-end latency of notifications as discussed in Section 3.2.1 can be broken down into processing latency, forwarding latency and path latency [138]. Forwarding latency is added to the publication latency of notifications due to the MOM having to process the notification. The MOM can minimize the *forwarding latency*. In a DEBS, further mechanisms are available to minimize the additional *path latency* that is added by dispatching notifications between brokers. Eichholz [138] discusses and evaluates different mechanisms for an EBS and a DEBS.

All available mechanisms can be grouped into two categories: those that aim at *prioritizing* certain notifications and those that aim at *reducing* the overall number of notifications to be processed. For example, a single broker can influence the forwarding latency by adapting its internal processing using notification prioritization, aggregation, or traffic shaping. In a DEBS, assigning publishers or subscribers to different brokers in the topology (publisher placement and subscriber placement) can be used to influence the path latency [138].

In this dissertation, we focus on publication and forwarding latency. The publication latency is controlled to some degree by the publisher and cannot be transparently decreased by the MOM. Additional forwarding latency, however, can be transparently minimized using aggregation and load shedding as described and evaluated in detail in [138].

Using a *rateController* with flexible outgoing rates, the MOM tries to free up resources on the broker to faster process notifications with strict latency requirements. The heuristic is triggered whenever a watchdog detects a latency requirement being violated. The key steps are [138]:

1. Rank all processed types of notifications based on their quantity, volume and cumulative fidelity in descending order.
2. Select the type of notification \bar{e} that generates the least cumulative fidelity.
3. Rank all subscribers of \bar{e} based on their expectations' utility value in descending order.
4. Aggregate or drop notifications for those subscriber(s) of \bar{e} with the lowest utility value.

Note that these steps describe a simple heuristic that tries to free up resources in a lazy fashion whenever a violation is detected for as long as it is detected. The MOM tries to reduce the number of notifications that have to be processed by aggregating or dropping all notifications of a certain type destined for a specific set of subscribers. The heuristic tries to maximize the impact on resource savings while limiting the impact on subscribers and the generated fidelity.

4.4.2 Client Self-Adaptation Using Feedback

In our approach we assume that most publishers can adjust their support for certain generic properties dynamically at runtime by using self-adaptation and additional feedback from the system [2, 3, 35, 62, 79, 96, 99, 109, 261, 362, 415]. For those publishers that are unable to adapt on their own, we assume that wrappers can be deployed.

For example:

- A gmond monitoring agent within the Ganglia [298, 299] open-source monitoring system can adjust the sampling rate of metrics using a wrapper that restarts the agent with new configurations.
- A publisher being part of the FINCoS open-source benchmarking tool [304] can change its sampling frequency, the precision and accuracy of its publications as well as its publication latency at runtime without the need to restart.

In our approach, the MOM triggers runtime adaptation of a publisher by sending a dedicated adaptation advice. As described in Section 3.5.1 and definition 7, such an adaptation advice contains the list of capabilities to adjust together with the required target values. However, an adaptation advice does not specify *how* this adaptation has to be done by the publisher.

4.4.3 Coordinated Adaptation

Adapting the current value for some generic properties requires coordination between MOM and publishers. Coordinated adaptation can be realized *explicitly* as a sequence of actions or *implicitly* by adjusting the costs of capabilities in the runtime negotiation phase.

We illustrate this using an example of a centralized EBS with a single broker, a single publisher P_1 but two subscribers S_1 and S_2 with requirements about sampling rate defined over two closed intervals that are disjoint.

Without coordinated adaptation, the requirements of only one subscriber could be satisfied; by implicitly using coordinated adaptation, the requirements of both subscribers can be satisfied.

The initial situation is illustrated in Figure 4.9: publisher P_1 provides notifications with a current sampling rate of 50 notifications per second to two subscribers S_1 and S_2 . Without any adaptation, both subscribers S_1 and S_2 would receive notifications with this sampling rate that exceeds their requirements about the closed intervals $[10; 20]$ (S_1) and $[30; 40]$ (S_2). However, P_1 could adjust the sampling rate between 15 and 60 notifications per second using self-adaptation or the MOM could apply self-adaptation to throttle the sampling rates delivered to each subscriber using a `rateController`.

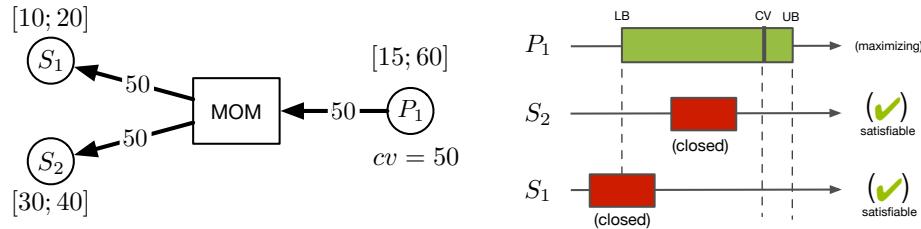


Figure 4.9.: Problem: only P_1 available but conflicting requirements about sampling rate.

Adapting only P_1 would not satisfy both subscribers as their ranges of accepted values are defined over closed intervals and do not overlap. Thus, advising P_1 to adjust to $\max S_1 = 20$ would leave S_2 unsatisfied while adjusting to $\min S_2 = 30$ would violate the requirement of S_1 .

Adapting the MOM would satisfy the requirements of both S_1 and S_2 as the sampling rate that P_1 sends with is currently greater than the required sampling rate. The scenario is shown in Figure 4.10: while P_1 would not adjust its current value and continue to send notifications with

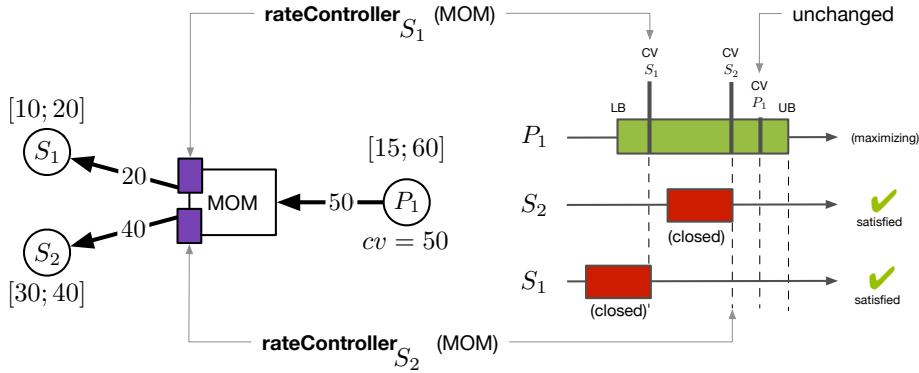


Figure 4.10.: Potential solution: adaptation of MOM only, traffic P_1 higher than necessary.

a sampling rate of 50 to the MOM, the sampling rates delivered to subscribers are curbed to the maximum values still accepted by each subscriber. While this solution satisfies both subscribers, not adjusting P_1 results in an overhead of 10 notifications per second that have to be dropped or aggregated by the MOM. This overhead wastes resources in terms of network traffic, CPU and energy for P_1 and MOM.

Alternatively, we could adapt both P_1 and the MOM using coordinated adaptation. This scenario is illustrated in Figure 4.11: while P_1 is advised to adjust its sampling rate to 40 to satisfy the requirement of S_2 which is defined over higher values than S_1 , the MOM *additionally* adjusts the sampling rate delivered to S_1 . In this case, the requirements of both subscribers are satisfied simultaneously while we do not have to drop surplus notifications at the MOM for S_2 . Please note that we adjust to the maximum values in this example as this is going maximize the generated fidelity (cf., Section 3.3.3).

The fact that we consider explicit costs in our model is the enabling factor for coordinated adaptation: at runtime publishers and the MOM can manipulate the operational costs of a capability (cf., Definition 4) and the adaptation costs of an action (cf., Definition 2).

Adjusting the adaptation costs for the different actions at runtime allows us to switch between the two solutions described above.

Forcing MOM-based adaptation is done by setting the adaptation costs for `adaptPublisher` to ∞ . This forces our negotiation algorithm to choose any other available action. The same applies if a publisher is not able to adapt, too expensive, or deemed too critical (e.g., high FIT score).

Forcing coordinated adaptation, on the other hand, is done by adjusting the operational costs of the capability sampling rate. We adjust the cost function for sampling rate once we advise the publisher to adapt for the first time. Depending on the improvement direction we set the operational costs to ∞ for all values lower (maximizing improvement direction) or greater (minimizing improvement direction) than the current value. Thus, we avoid that publishers adapt in a way that would violate already satisfied requirements about the sampling rate

In our example, setting the operational costs for the capability sampling rate to ∞ for any value $c < 40$ when satisfying S_2 prevents the algorithm to advise the publisher again to adapt to $c = 20$ when negotiating S_1 (as this would violate the already satisfied requirement of S_1). Please note that this does not keep us from reaching the same state when we negotiate S_1 first: upon adapting to satisfy S_1 , $costs_{adaptPublisher}$ are set to ∞ for any $c < 20$ while the original cost function remains for greater values. Thus, we can advise P_1 again to adapt to $c = 40$ when negotiating S_2 next.

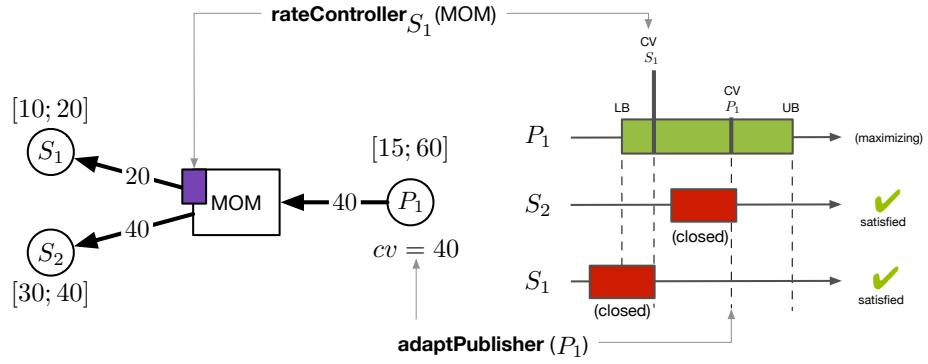


Figure 4.11.: Alternative solution: coordinated adaptation of MOM and P_1 .

While this iterative approach violates S_1 for some time, the system reacts immediately. As P_1 updates its capability as soon as the advised adaptation has finished, triggering a re-negotiation. Now, the negotiation algorithm notes that adapting P_1 is too expensive (∞) and searches for alternative actions to reduce the sampling rate again. At this point, any adaptation and operational costs defined for other actions are less expensive. In our example, this would result in using a **rateController**, while a different publisher could also be selected in a setup with multiple publishers.

Please note that we can *prevent* the system from using a **rateController** in the same way: setting the operational or adaptation costs for **rateController** to ∞ would prevent the system from using this action.

4.5 Monitoring the System State at Runtime

Every change in the system's context and state can trigger a reevaluation of affected expectations: a lifecycle change of an expectation, participants joining or leaving the system, or changes to a capability. While the participants themselves can actively announce some of these changes, the MOM has to be self-aware to detect all significant changes, including link-failures, network saturation, varying sampling rates, or crucial end-to-end latencies of notifications.

4.5.1 Detect and React to State Changes with Monitorlets and Watchdogs

In our approach, we separate the detection of significant changes from the reaction to them. We encapsulate the functionality to monitor specific aspects of the local or global system state in *monitorlets* while *watchdogs* encapsulate reactivity. Their design is the same for centralized or distributed setups and inspired by the concept of *eventlets* introduced by Appel et al. [18, 19, 20]. As shown in Figure 4.12, watchdogs and monitorlets are complementary. The local state refers to a single broker while the global state refers to a network of brokers in a DEBS, where we use a novel approach to monitor the global state of the system and implement monitorlets.

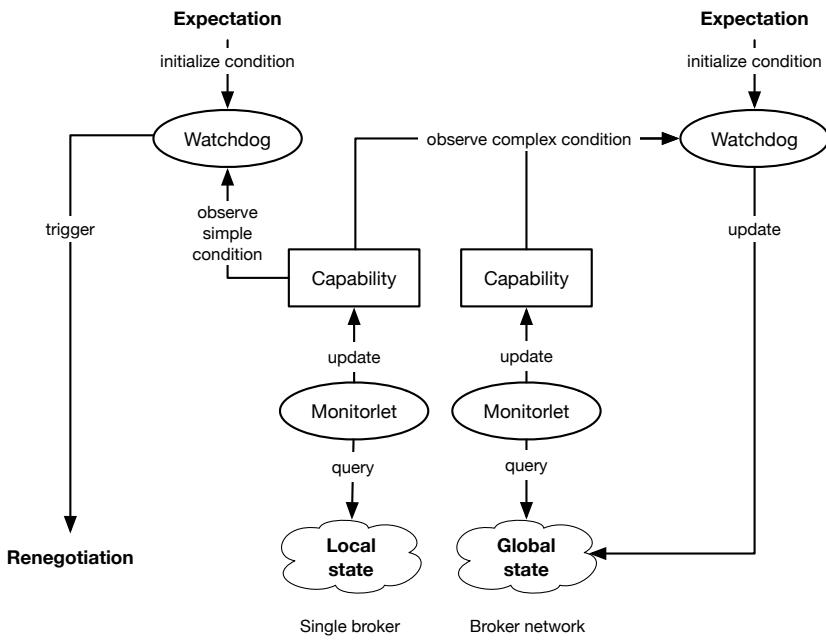


Figure 4.12.: Runtime monitoring using watchdogs and monitorlets.

A *watchdog* is bound to at least one expectation and observes the current value of one or more capabilities. Each watchdog is initialized with a simple or complex condition to check; it triggers a renegotiation of its associated expectation(s) or updates the global system state if that condition is met. Simple conditions refer to only a single capability, e.g., $C_k^e.CV < x \vee C_k^e.CV > y$, while complex conditions do also include multiple capabilities. The capabilities to observe and the condition to check are based on the associated expectation(s). The lifecycle of a watchdog is tied to the lifecycle of its initializing expectation(s) and a watchdog is destroyed by the MOM as soon as all of its initializing expectations are revoked. While each expectation is active, however, the conditions the watchdog has to observe can change with updates to the expectation.

For example, a watchdog for a *satisfiable* expectation triggers a negotiation of this expectation as soon as the current values of capabilities are *within* the range of allowed values; conversely, a

watchdog triggers the renegotiation of a *satisfied* expectation if it observes that the current value of a capability *exceeds or falls below* the range of allowed values defined for a requirement.

Monitorlets, in turn, act independently of expectations and do not trigger reactions. A monitorlet is responsible for updating the current state of a capability so that it can be observed by watchdogs and used by the MOM during the runtime negotiation phase. For this, monitorlets either query the local state of the broker or the global state of a DEBS. Examples are: the current forwarding and publication latency of notifications provided by a given publisher; the sampling rate of a single publisher for a given type of notification; the number of publishers currently providing notifications of a certain type; or the number of subscribers currently subscribed for a certain type of notification and set of requirements. Monitorlets are instantiated by the MOM at runtime when necessary. For example, they can be initiated upon pre-defined events by the MOM, e.g., subscribers or publishers joining the system for the first time. All monitorlets directly associated with capabilities of a publisher are destroyed when this publisher disconnects.

The way watchdogs work is illustrated in Figure 4.13 with an example containing one publisher and three expectations X_a^e, X_b^e , and X_c^e . A single publisher is providing notifications that fit the other requirements defined in these three expectations; its sampling rate is monitored by a dedicated monitorlet, which has been initialized by the publisher sending a new advertisement.

In our example, each expectation defines a requirement about the sampling rate over a closed interval of allowed values (shown on the right hand side of Figure 4.13). Thus, three watchdogs – w_a, w_b, w_c – are associated with these expectations and this publisher. Each watchdog observes the current value of the capability *sampling rate* of this publisher and triggers a re-evaluation of its associated expectation if the monitored sampling rate enters or leaves the range of allowed values defined in the associated expectation.

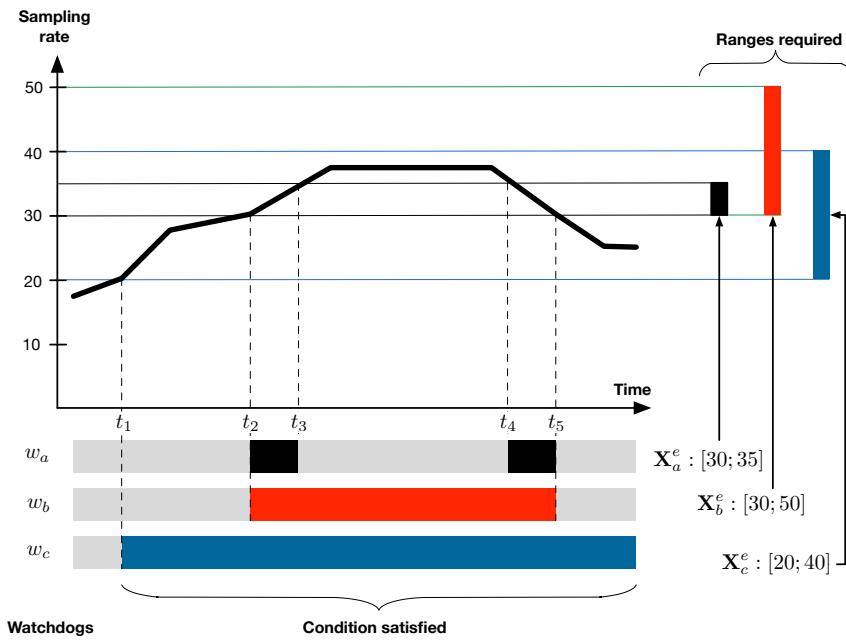


Figure 4.13.: Example for watchdogs detecting satisfied or unsatisfied conditions.

In our example, the sampling rate increases and decreases over time. At starting time, the sampling rate is too low to satisfy the requirements of any expectation, rendering them all *unsatisfied*. Thus, each watchdog is initialized to trigger a renegotiation if the sampling rate is equal or greater than the lower bound defined in each expectation.

The first watchdog to trigger a renegotiation of its expectation is w_c at t_1 . As the sampling rate requirement defined in X_c^e is now satisfied, w_c is changed to trigger a renegotiation of X_c^e as soon as the sampling rate becomes greater than the upper bound or lower than the lower bound. While this event is not detected for w_c it is for the other watchdogs. Watchdogs w_a and w_b trigger renegotiation at t_2 to satisfy X_a^e and X_b^e respectively. The necessity to change their triggering conditions becomes apparent at t_3 for w_a as the still increasing sampling rate exceeds the range of allowed values defined in X_a^e but not that defined in X_b^e (watched over by w_b). Now, the condition for w_a is changed again so that the watchdog can detect the sampling rate entering the range of allowed values again at t_4 . Both watchdogs w_a and w_b now have to trigger a renegotiation of their expectations again at t_5 as the sampling rate drops below the lower bounds defined in the associated expectations.

4.5.2 Monitoring the Global State of a Distributed Event-Based System

In a centralized EBS with a single broker, querying the state of this broker is sufficient for a monitorlet to gather all data that it needs to update a given capability.

In a distributed EBS (DEBS), computing the global state requires a monitorlet to collect the necessary metrics from all other brokers. Unfortunately, participants in an EBS are unaware of each other and anonymous except to their directly connected brokers. As illustrated in Figure 4.14, each broker that is part of the MOM has only local knowledge about its directly connected publishers, subscribers, and brokers. For example, broker B_k (black) is not aware of the total number of subscribers and publishers as these are connected to other brokers outside of its direct neighborhood (gray area).

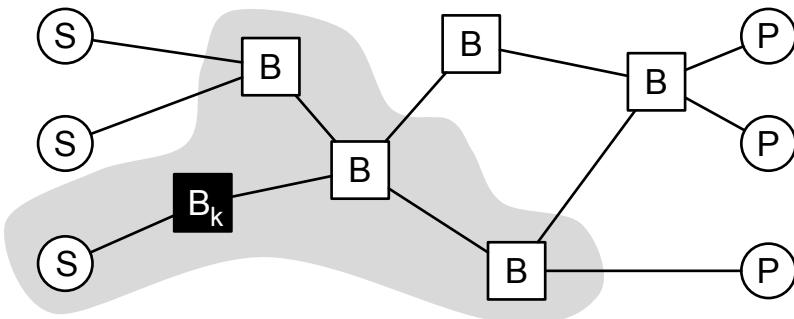


Figure 4.14.: Distributed decentralized EBS. Brokers (B) need only local knowledge (gray area) about directly connected brokers, publishers (P) and subscribers (S) for routing.

This design is beneficial for scalability but complicates the task of maintaining a global view on the state of the system, as local state information has to be synchronized between brokers. Furthermore, the dynamic nature of an EBS means that the aggregated data continuously changes over time and requires a large number of synchronization messages.

Overview: Application-specific Aggregation for Publish/Subscribe (ASIA)

We enable monitorlets and participants to be updated about the global state of the system using the concept of *application-specific integrated aggregation* (ASIA). In Section 3.5.2 we have already introduced the basics of ASIA. We have discussed how participants can express their interest in different runtime aspects of the population or dynamics of a DEBS and how they can specify the granularity as well as the precision of the metrics $m_a(e)$ contained in aggregated feedback.

In this chapter, we focus on how those metrics are computed in a distributed and decentralized network of brokers. We discuss how the MOM synchronizes the necessary state information between brokers and exploits the relaxations in data precision defined by participants.

ASIA dynamically *integrates* monitoring functionality into the broker network at runtime instead of adding a separate monitoring overlay. Using an approach that is inspired by Aspect-oriented Programming (AOP) [252], we augment the methods of a broker's Application Programming Interface (API) with *aggregators* that have access to the broker state and use the existing routing topology for transparently synchronizing state information between brokers. An aggregator encapsulates the computation of an *aggregation function* f_a that returns the requested metric $m_a(e)$ for notifications of type e . The global result for $m_a(e)$ is computed by iteratively aggregating the information exchanged between relevant aggregators [171, 172, 177].

In the terminology of AOP we define a set of joinpoints in the broker's business logic. These joinpoints can be advised to perform the aggregation of relevant information. Thus, our joinpoints can be used to support more general behavior change in brokers such as aggregating or filtering any kind of notification [173]. An advice is used to invoke an aggregator, e.g., for computing the number of subscribers currently active [172].

As shown in Figure 4.15, ASIA uses the existing routing overlay between participants to send additional metadata (red arrows) about aggregations to those participants that have expressed interest in them. In a typical EBS, the flow of information is unidirectional from publishers to subscribers: there is no feedback from the subscriber side of the system back to the publisher side. The subscriptions sent from subscribers to brokers are not forwarded to publishers but matched against advertisements within the broker network to establish a routing topology from the edge broker of the publisher to the edge brokers of each interested subscriber. At the same time, subscribers do not get any metadata on the population of publishers as publishers and subscribers are fully decoupled by the MOM [147, 208].

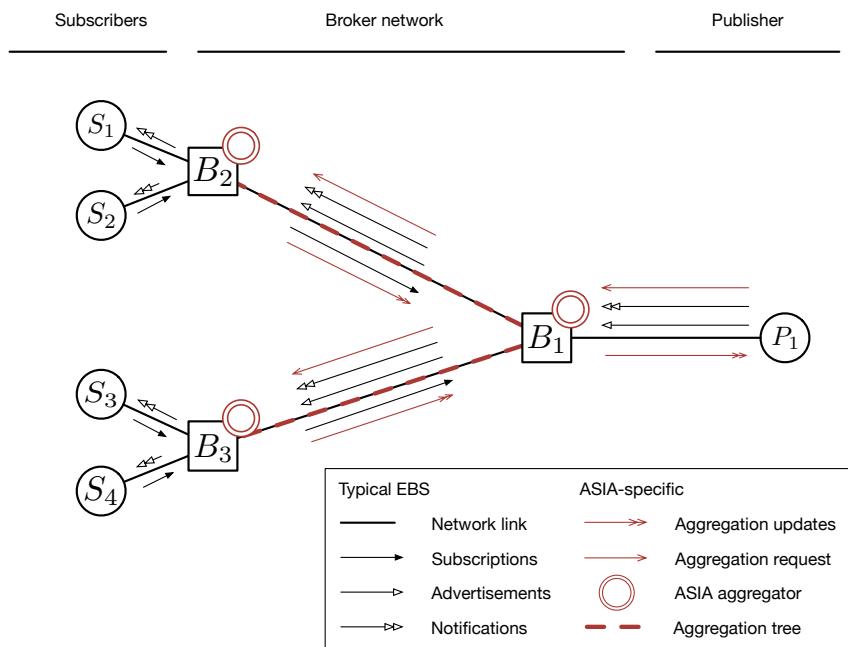


Figure 4.15.: Information flow in a DEBS (black) with additional ASIA metadata (red) for aggregated feedback about `subscriberCount`.

In ASIA, subscribers, publishers and monitorlets of brokers can request aggregated feedback. Using the extended API of its edge broker (i.e., B_1 for P_1 in Figure 4.15), an interested participant indicates its interest in updates about an aggregated metric $m_a(e)$. As already introduced in Definition 8, a participant requests updates for an aggregated metric by sending a tuple $(e, m_a(e), \hat{v}, \vec{c})$ to its edge broker containing the metric, the required precision as well as an optional set of constraints. With that it provides a callback function that is invoked by the edge broker when $m_a(e)$ changes to a significant degree.

Distributed Aggregation of State Information and Synchronization Between Brokers

In a DEBS, a participant that registers interest in an aggregated metric at its edge broker B_r causes the formation of a spanning tree rooted at B_r . This *aggregation tree* contains all other brokers B_i that can contribute relevant aggregation results for $m_a(e)$ to B_r . The tree is used to synchronize aggregation results between brokers and is based on the established routing topology. Thus, we do not have to maintain a separate overlay but can increase efficiency by piggybacking information onto notifications that are routed anyway rather than being sent separately.

Using the aggregation tree we can distinguish between *upstream* and *downstream* neighbors. From the perspective of a broker B_i , upstream neighbors are those brokers that B_i receives aggregation updates from; downstream neighbors, in turn, are those brokers, subscribers or publishers that B_i sends aggregation updates to.

Maintaining the aggregation tree is crucial as computing the global result for an aggregated metric $m_a(e)$ is done by iteratively combining the local knowledge of a broker B_i with the aggregated results for the same metric sent from its direct upstream neighbors in the aggregation tree.

Thus, each broker B_i maintains four values::

lc_i	the local value of $m_a(e)$ at broker i
tc_k^{rec}	the last result for $m_a(e)$ received from upstream broker k
tc_i^{new}	the local result for $m_a(e)$ at broker i
tc_r^{send}	the last result for $m_a(e)$ sent to downstream receiver r

Where tc_i^{new} is the combination of the local value and the updates received from all upstream brokers based on the semantics of the aggregation function. For a sum-based aggregation such as *subscriberCount*, the total count would be the sum of the directly connected subscribers and the received sum of subscribers being subscribed to other brokers as shown in Equation (4.1).

$$tc_i^{new} = lc_i + \sum_k tc_k^{rec} \quad (4.1)$$

A broker B_i sends an update to its downstream receivers if tc_i^{new} differs from tc_r^{send} . The process of calculating and updating this global aggregation result is triggered by each aggregator that detects a significant change in its broker's local state, e.g., a subscriber connecting or disconnecting from its edge broker. Please note that optional constraints \vec{c} could restrict the aggregator, e.g., to monitor only those publishers that provide a certain set of capabilities.

In this dissertation we focus on aggregators using additive functions f_a , including counting and rate measurements. Any associative and commutative function can be used within an aggregator, including multiplication, set operations, maximum/minimum or the arithmetic mean.

Figure 4.15 shows the aggregation tree (dashed red line) formed when P_1 requests the count of subscribers as aggregated feedback from B_1 . From the perspective of B_1 , the subscriber count is $4 (S_1, S_2, S_3, S_4)$. B_1 can determine this total number of subscribers by summing up the results received from its immediate children B_2 and B_3 .

Exploiting Relaxations of Data Precision

For precise results, *every* change at any broker causes updates to be sent up to the root of the aggregation tree, inducing overhead for dynamic situations with many changes. Therefore, we exploit the relaxations in the metrics' precision that each participant defined upon requesting aggregated feedback. This feature is similar to other scalable aggregation models [240]: a distance d sets the maximum imprecision that will be tolerated by the aggregator at a particular broker. An imprecision of 1 denotes very precise results while a higher imprecision indicates less precise results (cf., Figure 3.26).

When a participant requests updates for a metric with an imprecision \widehat{v}_r , its directly connected edge broker B_r splits this imprecision according to Equation (4.2) and forwards the request with the split imprecision \widehat{v}_k to each of its m upstream neighboring brokers. This process is repeated at each broker along the aggregation tree until all upstream edge brokers are reached.

$$\widehat{v}_k = \frac{\widehat{v}_r}{1 + m} \quad (4.2)$$

The local imprecision is defined per downstream receiver, aggregation, set constraints and type of notification. Thus, each broker has to maintain different local imprecisions. In case that a broker B_i has to forward multiple requests for the same aggregation, constraints and type of notification, it forwards only one request with the minimum imprecision of all locally known requests. However, B_i keeps the original requests so that it is able to update each of its own requesters based on their imprecision. This approach is similar to filter merging in a DEBS.

With the split imprecisions distributed along the aggregation tree, each broker B_i can use this information about relaxed precision to individually decide if an update has to be propagated.

As before, the update process is triggered if an aggregator detects a change in the local state of the broker or the broker receives an update from an upstream broker. However, by using imprecision, the exact values we have introduced to synchronize state information between brokers turn into intervals with upper and lower bounds as already described in Definition 8. The only exception is lc_i , as a broker B_i is supposed to determine the exact value from its local state.

Having updated its local value lc_i , each triggered broker B_i computes the interval of its current total count tc_i^{new} using Equations (4.3) and (4.4) as the weighed sum of its own local count lc_i and the aggregates tc_k^{rec} received from each upstream broker B_k .

$$tc_i^{new}.LB = lc_i + \sum_k^m tc_k^{rec}.LB + \left(\sum_k^m tc_k^{rec}.UB - tc_k^{rec}.LB \right) \cdot \frac{1}{2} - \frac{\hat{v}_r}{2} \quad (4.3)$$

$$tc_i^{new}.UB = lc_i + \sum_k^m tc_k^{rec}.UB + \frac{\hat{v}_r}{2} \quad (4.4)$$

For each receiving downstream participant r (broker, subscriber or publisher), B_i uses Equation (4.5) to check if the new total count still overlaps with the last update sent to r . If either of the conditions evaluates to true, an update has to be sent.

$$\underbrace{\left[tc_i^{new}.LB > tc_r^{sent}.UB - \delta \right]}_{\text{overshoot}} \vee \underbrace{\left[tc_i^{new}.UB < tc_r^{sent}.LB + \delta \right]}_{\text{undershoot}} \quad (4.5)$$

In Equation (4.5) we add a safety margin δ to trigger updates sooner while the two intervals still overlap; for quickly changing metrics, this avoids a temporary situation where the true value is not enclosed by the interval known to the participant anymore.

This way, updates are only propagated downstream if the current state of the subtree differs too much from the previously sent update. The level of imprecision, the branching factor of the aggregation tree and the safety margin δ determine the sensitivity of the updates.

Example ASIA Aggregation: Subscriber Counts

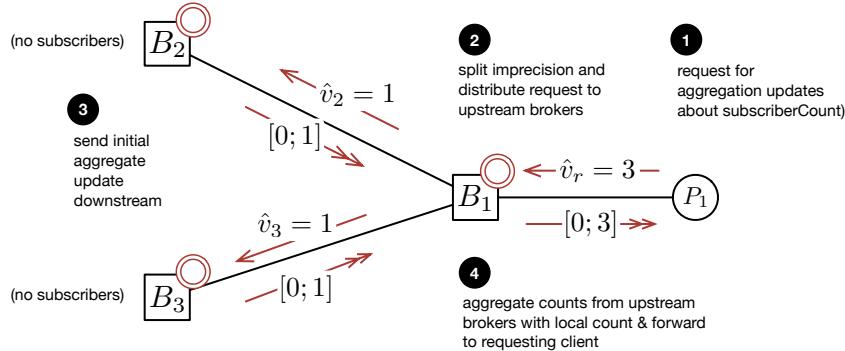
We illustrate the whole approach using a simple subscriber count aggregator implemented in ASIA. Figure 4.16 illustrates each step of our approach and its update algorithm. We assume that publisher P_1 requests aggregated feedback about the number of active subscribers in the system from its edge broker B_1 , allowing for an imprecision $\hat{v}_r = 3$. The aggregation tree is built accordingly by B_1 which splits the received imprecision evenly between itself and its two upstream neighbors B_2 and B_3 , resulting in $\hat{v}_1 = \hat{v}_2 = \hat{v}_3 = 1$

In an initial bootstrapping step, we distribute the split imprecision and collect the initial intervals from all affected upstream brokers as shown in Figure 4.16a. If the local count for a broker B_k is 0 it still returns an interval $[0; \hat{v}_k]$ to its next downstream neighbor.

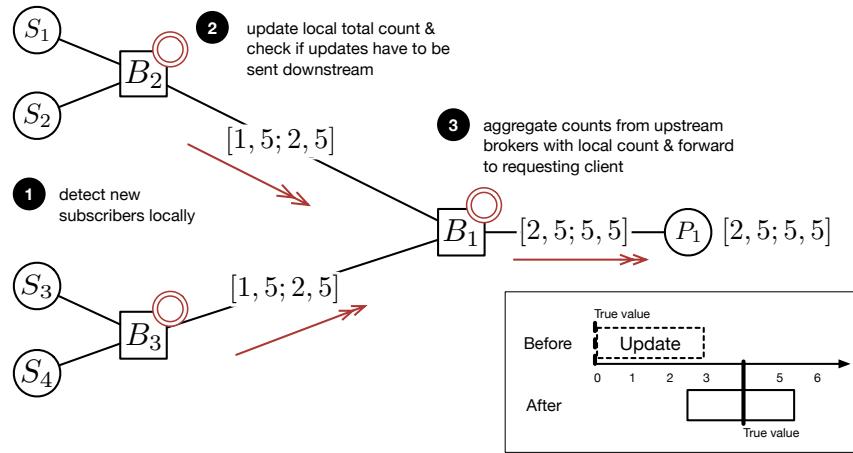
Next, we assume that four subscribers join the system, two at each edge broker. Figure 4.16b shows the three steps of the update algorithm: first, the edge brokers detect the new subscribers. Second, both B_2 and B_3 update their local values $lc_2 = 2, lc_3 = 2$ and check if they have to send an update to B_1 (i.e., $tc_2^{new}.UB = tc_3^{new}.UB = 2 > 1 = tc_1^{sent}.UB$). Third, at B_1 , these updates trigger a reevaluation of its total count, resulting in an update sent to P_1 as the last update is outdated.

Finally, we assume that a single subscriber disconnects from B_3 (cf., Figure 4.16c). B_3 detects this event, updates its local counts and sends an update to its downstream neighbor B_1 where no update is sent to P_1 as the last update is still precise enough.

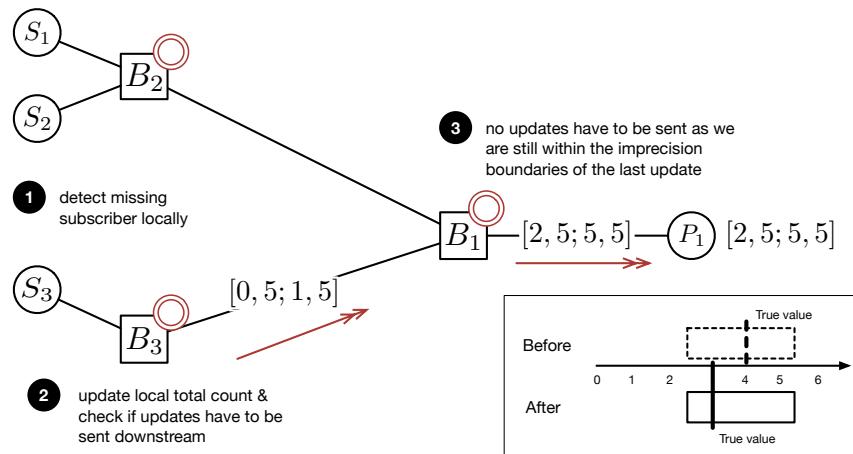
This example shows how ASIA detects changes to the population of a DEBS and notifies interested participants only if the detected change is significant for them. By exploiting P_1 's imprecision, one update message from B_1 to P_1 has been saved while the last aggregated feedback to P_1 is still valid as its interval encloses the true value (cf., bottom right corners in Figures 4.16b and 4.16c).



(a) Bootstrapping ASIA for `subscriberCount`.



(b) Updates for `subscriberCount` sent when four subscribers joins.



(c) No updates for `subscriberCount` sent when one subscriber leaves.

Figure 4.16.: Example using `subscriberCount`: bootstrapping and updating steps in ASIA.

4.6 Resolving Possible Conflicts at Runtime

During runtime negotiation and runtime adaptation, three types of conflicts can occur while negotiating and enforcing requirements about QoI in EBS: participants might use definitions for generic properties that conflict with definitions used by the MOM, a subscriber's requirements might conflict with the current system state, or adapting the system to satisfy requirements of one subscriber might violate the requirements of another subscriber.

Conflicting Definitions Used by Participants

In general we assume that all participants use the same definition for a generic property. This can be realized by querying the repository on each edge broker for the currently used definitions or by defining transformations as discussed in Section 3.2.2.

In some cases, however, some participants might still end up with their own definitions that could conflict with the definitions deployed at the MOM. We illustrate two possible types of conflicts using the property *delivery guarantees* as an example.

Subscriber-side conflict: a subscriber-side conflict is shown in Figure 4.17. In our example, the MOM defines *delivery guarantee* as: "best effort" < "at most once" < "at least once" < "exactly once". However, a subscriber wants to avoid a delivery guarantee of "at most once" and defines *delivery guarantees* as "best effort" < "at least once" < "at most once" < "exactly once". Consequently, the requirement about *delivery guarantees* in X_i^e is defined over a closed interval ["at most once";"exactly once"] as shown in the top left corner of Figure 4.17.

As shown in the bottom left part of Figure 4.17, mapping the subscriber's lower and upper bounds to the MOM's definition results in the label "at least once" being enclosed by "at most once" and "exactly once". Thus, adapting the system to provide "at most once" might be a valid choice from the perspective of the MOM but it would not satisfy the subscriber's preferences.

This conflict is solved in our approach by splitting up the original expectation X_i^e into two expectations $X_{i,1}^e$ and $X_{i,2}^e$ with disjunct intervals for *delivery guarantees* as shown in the top right corner of Figure 4.17; both expectations are defined over the same properties and ranges otherwise. In principle, this split can be done by the subscriber or by the MOM when receiving X_i^e .

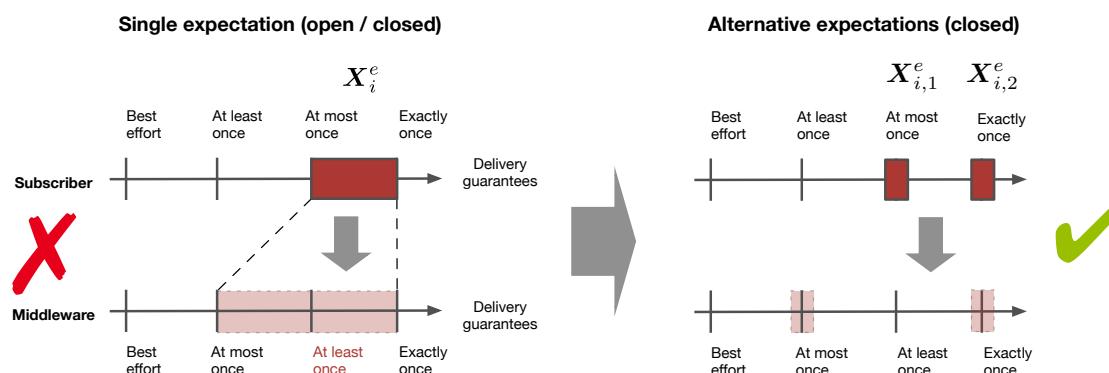


Figure 4.17.: Conflicting definitions used by subscribers and MOM are resolved by splitting up the original expectation into two equivalent expectations using a closed requirement.

Publisher-side conflict: the case we have discussed for subscribers could be applied to publishers as well as shown in Figure 4.18. Let us assume again that a publisher's definition of the generic property *delivery guarantees* differs from that of the MOM. The MOM ranks the labels for delivery guarantees as "best effort" < "at least once" < "at most once" < "exactly once" based on the execution costs arising for the current configuration.

One publisher, however, ranks these labels as "best effort" < "at most once" < "at least once" < "exactly once" as it is resource-constrained and checking whether a notification has already been sent to avoid duplicates is considered to be less expensive than sending multiple copies. In this case, the publisher defines its cost function for the capability *delivery guarantees* as shown on the right hand side of Figure 4.18 with "at least once" being more expensive than "at most once".

No adjustments have to be made to solve this conflict: if the MOM decides that it requires a publisher to adapt its capability to "at least once" to satisfy an expectation, it compares the costs predicted by different publishers. As the preference of this publisher is already reflected in its cost function, the costs for using this publisher would be higher than for using a different publisher.

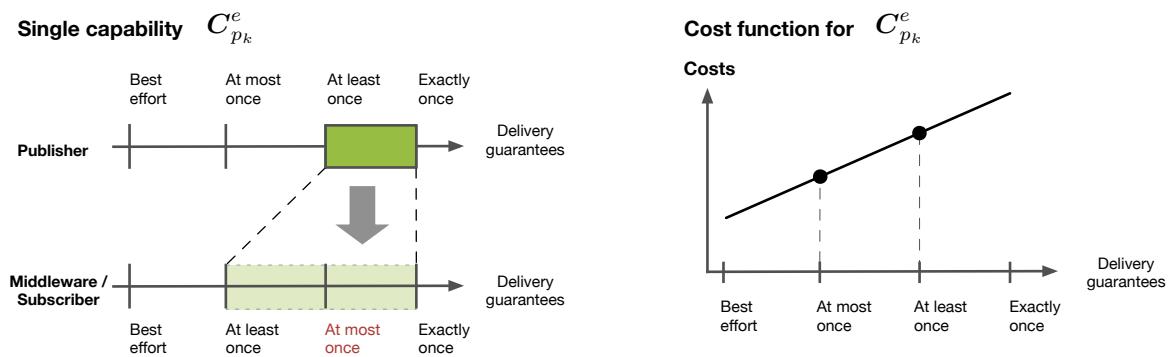


Figure 4.18.: Conflicting definitions used by publishers and MOM are directly taken care of by the capability's cost function that is defined by the publisher.

Conflicts Between Requirements and System State

Conflicts between the requirements of a subscriber and the system state are resolved by the algorithms we have introduced in Section 4.1. A conflict could be severe so that the expectation is *unsatisfied* or it could be solvable by deciding on an expectation that is *satisfiable*.

Conflicts Arising From Adapting the System

Adapt the system to satisfy an expectation could result in two types of conflicts: conflicts between the new system state and requirements defined in other expectations, or conflicts with global or local constraints imposed by the MOM, e.g., budget constraints or resource utilization.

In our approach, these conflicts are resolved during the safeguarding step as described in Section 4.3 or using coordinated adaptation as described in Section 4.4.3.

4.7 Summary

In this chapter, we have described the algorithms used in our approach to provide runtime support for QoI requirements in an EBS. Runtime support in our approach is based on *runtime negotiation* of expectations with capabilities, *runtime adaptation* of participants, and *runtime monitoring* of the population and dynamics of the system. We have described how these tasks are entwined with each other and how each is structured internally.

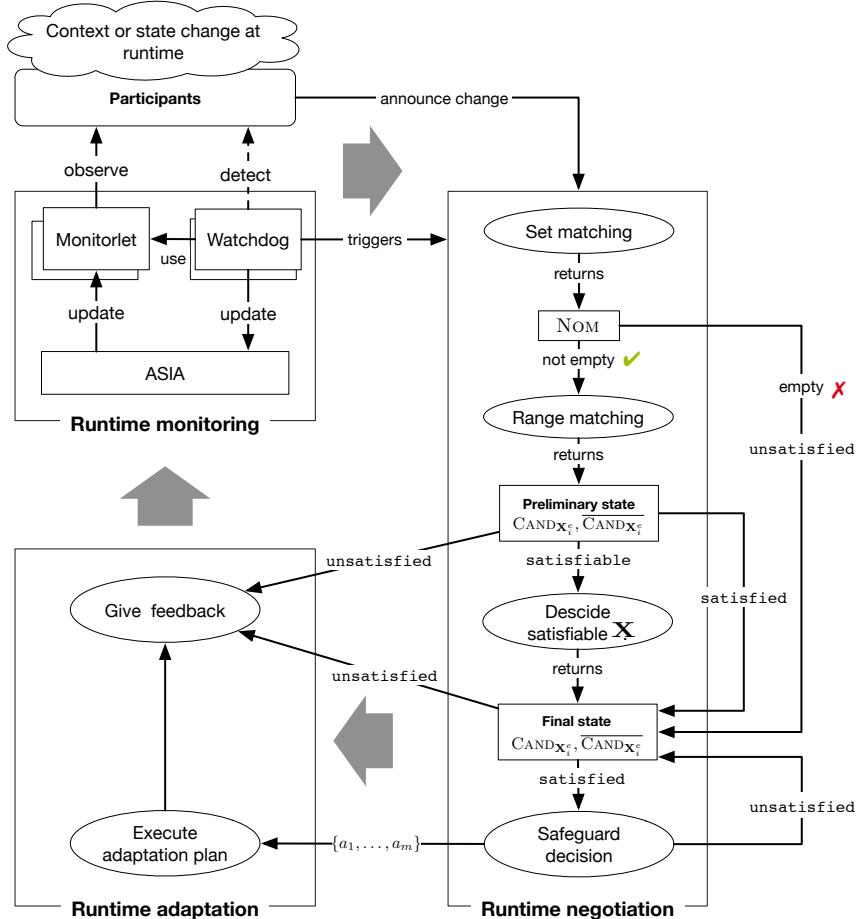


Figure 4.19.: Runtime support for QoI and its components summarizing Chapter 4.

Figure 4.19 summarizes the topics discussed in this chapter with their relationships and illustrates the cyclic characteristics of runtime support: runtime negotiation of QoI requirements is triggered by significant changes to participants at runtime. In our approach, these changes are either actively announced by participants or detected by watchdogs and monitorlets as part of runtime monitoring for centralized and decentralized EBSs. Significant changes refer to new, updated or revoked expectations as well as to new, updated or deteriorated capabilities.

Runtime negotiation is done for each expectation at a time. It starts with a set-matching step that identifies all publishers that would be able to satisfy all requirements expressed in an expectation X_i^e . If this set of nominees ($NOM_{X_i^e}$) is empty, we can already decline the expectation as *unsatisfied* due to missing publishers. Otherwise, we can proceed with checking the expectation against all nominees. In this range-matching step, we check if the current state of the system is already sufficient to satisfy the expectation or if the system has to adapt. This check compares the ranges of accepted values captured in the requirements with the current system state expressed

as the current values of matching capabilities. If the expectation cannot be satisfied by the current system state, the algorithms check whether the system could adapt to an extent that would enable it to satisfy the expectation. This is done by matching the ranges of accepted values with the ranges of values realizable with adaptation and represented by the ranges that matching capabilities are defined over. The results of this range-matching process are captured in a preliminary state of the expectation as well as two sets of capability profiles: one that denotes publishers already satisfying the requirements (C_{AND}) and one that identifies capability profiles that would have to be adapted first ($\overline{C}_{\text{AND}}$). In case that the system would not be able to satisfy the requirements even by applying adaptation, we again skip all following steps and decline the expectation as *unsatisfied*.

If the preliminary state of the expectation is set to *satisfiable*, the MOM has to decide whether the system should adapt in order to satisfy the expectation. We have shown how decision strategies can be used at this point to formalize different optimization goals with their hierarchy of objectives and attributes, e.g., number of satisfied subscribers, or adaptation costs.

Deciding on the final state of the expectation by safeguarding the decision completes the runtime negotiation part of our runtime support. The final state of the expectation is either *satisfied* or *unsatisfied* and determines how the system proceeds in the runtime adaptation part of our approach. If the expectation is *satisfied*, we proceed to safeguard the decision while we skip this step in case of the expectation being declined as *unsatisfied*. We have shown how safeguarding the made decision can be done algorithmically using the example of requirements about *alternatives*: during the safeguarding step, the MOM makes sure that the total number of publishers a subscriber is going to receive data from does not violate the maximum number of alternatives defined in an expectation. If we detect severe violations that cannot be solved by adjusting the adaptation plan, we have to discard the expectation as *unsatisfied* nonetheless.

The final adaptation plan is the result of the safeguarding step and is sequentially executed before the MOM gives individual feedback about the state of the expectation. In addition to the final states *satisfied* and *unsatisfied*, we notify the subscriber about an ongoing adaptation of the system by sending a temporary *pending* state which indicates that the expectation is going to be satisfied soon; as soon as the adaptation is complete and the expectation satisfied, the subscriber is notified about the expectation being *satisfied* (cf., Section 3.5.1).

Potential conflicts can arise at runtime during negotiation and adaptation. We have described the different types of conflicts and how they are algorithmically resolved in our approach.

The strength of our solution is its general applicability: all the principles and mechanisms described in this chapter do not rely on specific properties or software platforms. Thus, we can support arbitrary QoI properties on various platforms. The feasibility of runtime negotiation is limited only by the complexity of the custom decision strategy that is defined upon deployment — the more complex the decision making process becomes, the more dependencies and compensations would have to be considered during the final safeguarding step. In this chapter, we have presented straightforward heuristics as a starting point and show their feasibility in the following chapters that describe the implementation and evaluation of our prototypes.

5 Implementation

In this chapter, we present the design and prototypes of an architecture to implement the runtime support for Quality of Information (QoI) requirements in an Event-based System (EBS) or a Distributed Event-based System (DEBS) using expectations, capabilities and feedback.

Our architecture consists of extensions to the Message-oriented Middleware (MOM) as well as additional libraries and interfaces provided to publishers and subscribers. We distinguish between different levels of abstraction: generic, platform-specific, and application-specific. Most parts of our architecture are independent of the MOM platform as well as of the applications that use the MOM for communication. Some extensions to the MOM are platform-specific but independent of the application running on it. A few components, however, have to be platform- and application-specific to fit in with the semantics of an application.

The chapter is structured in three parts along these levels of abstraction as shown in Figure 5.1.

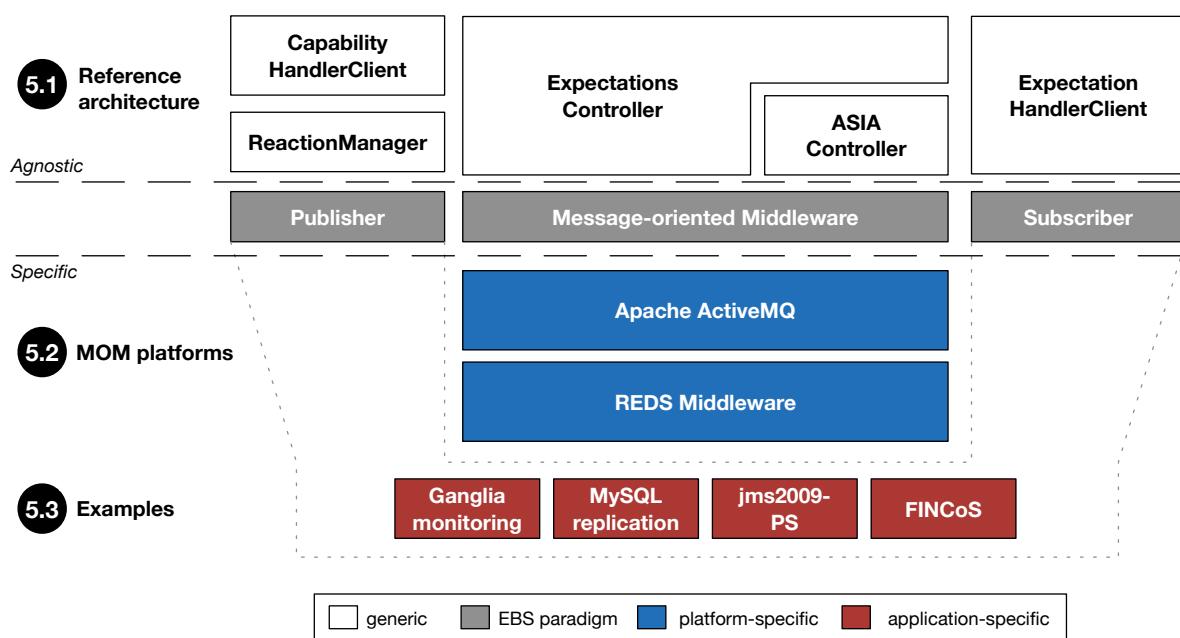


Figure 5.1.: Chapter overview: platform- and application-agnostic design, platform-specific prototypes (centralized, distributed), and example applications.

In Section 5.1, we describe the overall design of all components implementing our approach. The reference architecture described in this section is independent of the actual MOM platform being used or the application running on it. We describe platform-specific parts where necessary.

In Section 5.2, we describe the platform-specific components of our prototypes built for two open-source MOMs written in Java: Apache ActiveMQ and REDS. We describe the integration of our reference architecture into each platform and present the additional code necessary for participants to negotiate QoI requirements with the MOM.

In Section 5.3, we discuss four open-source applications we have extended to use expectations, capabilities and feedback: MySQL replication, Ganglia monitoring, FINCoS, and jms2009-PS.

5.1 Architecture and Design

Our architecture consists of an extension to each broker of the MOM as well as handlers provided to subscribers and publishers to deal with feedback and manage the lifecycle of expectations or capabilities. The remainder of this section is structured in three parts as shown in Figure 5.2.

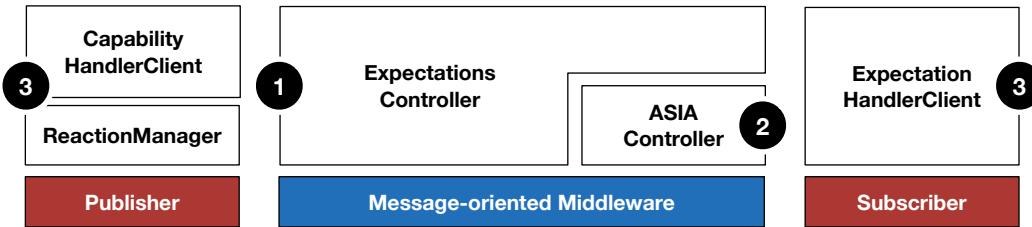


Figure 5.2.: Components discussed in this section: generic support for MOM and participants.

First, we focus on the extensions to the MOM. We start with describing the design of **ExpectationController**, which extends the MOM to support the runtime negotiation of expectations with capabilities. Then we describe the design of **ASIAController**, which implements the ASIA approach to provide aggregated feedback in a DEBS.

The handlers provided to subscribers and publishers are described in the third part of this section. We distinguish between passive client handlers that support participants in managing their expectations or capabilities and active reaction managers that enable self-adaptation of otherwise static publishers by enforcing adaptation advices sent by the **ExpectationController**.

The symbols used in all architectural diagrams are described in Figure 5.3.

EBS entities	Data access	Interface(s)	Component
→ Advertisement	→ Read	● Provided	Platform-specific
→ Subscription	↔ Read/Write	—○ Required	
→ Notification			Application-specific

Figure 5.3.: Legend of symbols used in all architectural diagrams.

5.1.1 Extending the Message-Oriented Middleware: **ExpectationController**

The core of our architecture is an extension to the MOM shown in Figure 5.2 called **ExpectationController**. It consists of platform-independent components to negotiate expectations with capabilities and a few platform-specific components to enable reactions, e.g., filtering notifications. MOM and participants communicate with platform-specific messages.

An **ExpectationController** consists of five key components:

- **ResourceMonitor**: monitors the system state and reports changes to the **Registry**.
- **Registry**: stores all necessary state information including expectations and capabilities as well as metadata about participants. Changes trigger runtime negotiation at the **Balancer**.
- **Balancer**: matches expectations to capabilities and decides on satisfiable expectations. Triggers the **ReactionCoordinator** upon completion to execute the adaptation plan.
- **MechanismsRepository** stores all applicable actions to manipulate generic properties.
- **ReactionCoordinator**: applies actions from the **MechanismsRepository** and coordinates their execution by adapting the MOM, advising publishers, and notifying subscribers.

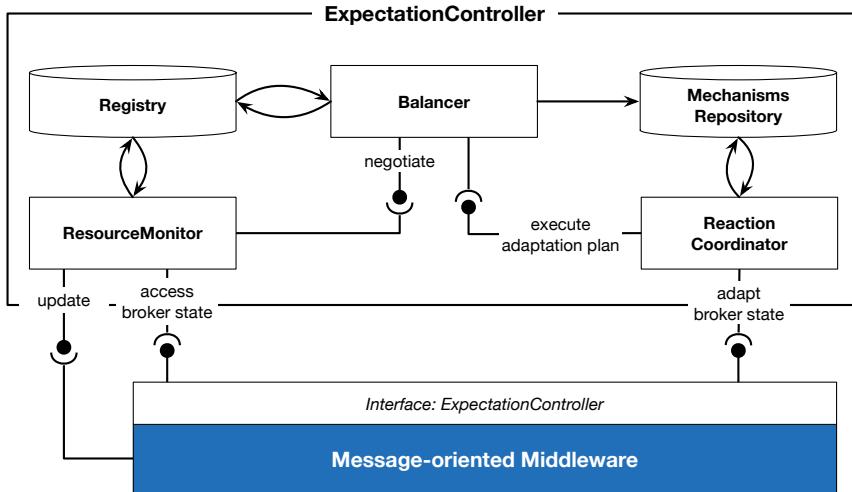


Figure 5.4.: Design of the MOM extension to enable runtime support for QoS.

The components encapsulated in the **ExpectationController** require access to the state of the MOM for monitoring the system and to apply reactions like filtering messages or routing adaptation [172]. The access is provided through an Application Programming Interface (API) by the MOM implementing the **ExpectationController** interface (cf., Figure A.1a in the appendix).

The methods to be provided by each broker implementing the **ExpectationController** interface can be categorized into three types: first, forward updates about the lifecycle of expectations and capabilities (e.g., register, update, revoke) to the **Registry** in a platform-independent format; second, enable the access between different components of the **ExpectationController**; third, allow for manipulating the routing tables to adapt the routing of notifications.

In the next sections, we are going to present the internal design of each key component in the **ExpectationController**. We will describe their generic, platform- and application-specific parts as well as their interfaces to each other in more detail.

ResourceMonitor: Monitoring the System State

The **ResourceMonitor** encapsulates the functionality to monitor the system state as described in Section 4.5, report changes to the **Registry**, and trigger the **Balancer** whenever necessary.

The **ResourceMonitor** consists of a set of active monitorlets and watchdogs as shown in Figure 5.5. The lifecycle of each monitorlet is managed by a **MonitorletsRepository** while a **WatchdogsRepository** does the same for watchdogs. Each repository stores a set of prototypes. A factory creates, updates or deletes instances of appropriate prototypes whenever necessary.

Each instance of a *monitorlet* keeps a specific capability in the **Registry** up-to-date by monitoring the state of the broker or the system as described in Section 4.5.1. Thus, monitorlet prototypes are assumed to be platform- or application-specific, e.g., using **AdvisoryMessages**¹ on ActiveMQ or ASIA aggregations as described in Section 4.5.2.

Each instance of a *watchdog* monitors a capability or a set of capabilities based on conditions and triggers a renegotiation of an expectation at the **Balancer** whenever its condition is satisfied

¹ <http://activemq.apache.org/advisory-message.html>

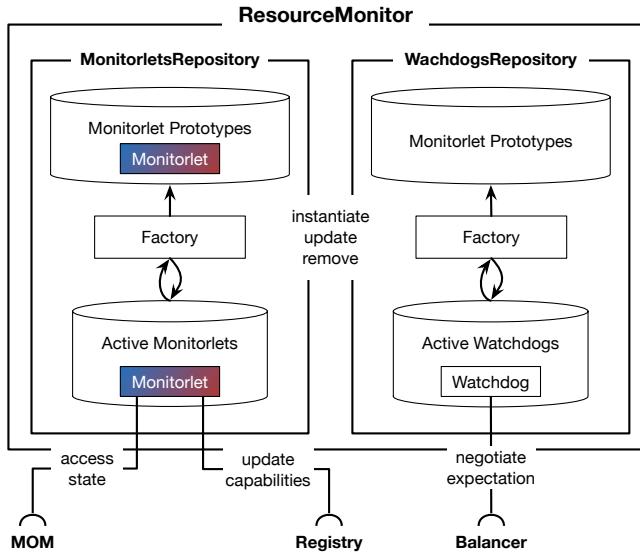


Figure 5.5.: Design of the ResourceMonitor.

as described in Section 4.5.1. As watchdogs only operate within the `ExpectationController`, prototypes for watchdogs do not have to be platform- or application-specific necessarily.

Monitorlets and watchdogs can trigger implicitly or explicitly: *implicitly*, they run in the background and constantly pull data; *explicitly*, they are called when some significant event takes place, e.g., an ASIA aggregation update is received.

The granularity of watchdogs can be single or cumulative. *Single* refers to each expectation maintaining its own watchdog instance; *cumulative* refers to all satisfied expectations for a certain type of notification defining a single watchdog while also all satisfiable expectations per type of notification share a single watchdog. Single watchdogs reduce the negotiation overhead as they allow for each expectation to be (re)negotiated only if its specific conditions are met or violated. However, depending on the number of active expectations and their size (in terms of the number of generic properties they are defined over), having multiple threads of watchdogs running in the background can increase the resource utilization overhead. Cumulative watchdogs reduce resource utilization as fewer watchdogs are to be maintained. However, the number of unnecessary renegotiations might be higher than for single watchdogs if the ranges defined for the different expectations are not close together.

In our prototypes, and the example described in Figure 4.13, we have used single watchdogs that trigger implicitly by running as daemon threads. Please note that monitorlets and watchdogs could be implemented using `eventlets` [18] in principle.

Registry: Manage All Necessary State Information

The **Registry** stores all necessary state information and metadata about the system in one place and independently of the MOM platform or application running on it. As shown in Figure 5.6, this information is not restricted to metadata about subscribers, publishers and neighboring brokers but also includes the definitions of generic properties, expectations and capabilities.

The storage, state and lifecycle management of expectations is encapsulated in an **ExpectationsRepository**. The **CapabilitiesRepository** is similarly designed for encapsulating the storage

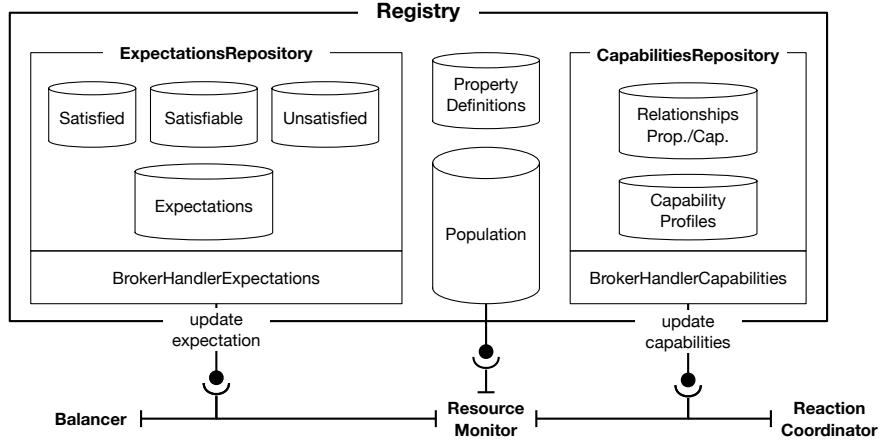


Figure 5.6.: Design of the broker component Registry.

and lifecycle management of capabilities together with the relationships/mappings/transformations between capabilities provided by publishers and generic properties required by subscribers (cf., Section 3.2.2 and fig. 3.8). Access to each repository is provided through broker handlers that offer an API to query and update expectations or capabilities.

The **Registry** can be accessed by every other component of the **ExpectationController**. In our reference architecture, the **ResourceMonitor** is the main contributor to the **Registry** as it monitors the system state and reports every change to the **Registry**. The state of expectations is also updated by the **Balancer** upon completing the runtime negotiation process as described in Sections 4.1 and 4.2. The state of capabilities, in turn, can also be changed by the **ReactionCoordinator** to enforce coordinated adaptation as described in Section 4.4.3.

Balancer: Decide on Expectations and Safeguard Decision

While **ResourceMonitor** and **Registry** provide up-to-date information about the system state, the **Balancer** encapsulates all functionality needed during the runtime negotiation phase: matching expectations to capabilities, deciding on satisfiable expectations, safeguarding the decision, and finally triggering **ReactionCoordinator** to pass on to the runtime adaptation phase.

The **Balancer** consists of three components as shown in Figure 5.7.

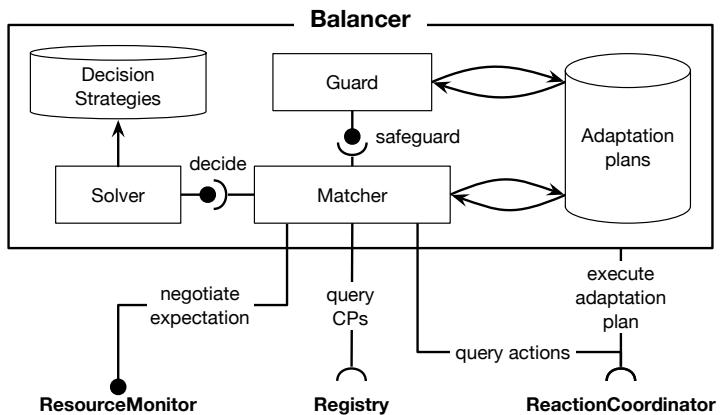


Figure 5.7.: Design of the Balancer which implements the runtime negotiation phase.

The **Matcher** is triggered by the **ResourceMonitor** to negotiate a dedicated expectation. For this, it queries the **Registry** for available capability profiles with a matching set of capabilities to perform the set and range matching steps of the runtime negotiation phase as described in detail in Section 4.1.

The **Solver** is triggered by the **Matcher** if the expectation is not yet satisfied but satisfiable. Using predefined decision strategies and custom approaches, the **Solver** decides whether to satisfy the expectation or decline it as unsatisfied (cf., Section 4.2). The result is reported back to the **Matcher**, which proceeds with safeguarding the decision by triggering the **Guard** component.

The **Guard** checks the preliminary adaptation plan for inconsistencies or violations of constraints as described in Section 4.3. It tries to solve violations or inconsistencies by adjusting the actions listed in the adaptation plan. If the **Guard** cannot resolve the violation of constraints, the expectation is declined as unsatisfied. The **Matcher** is informed about the result in any case, as it has to trigger the **ReactionCoordinator** to execute the adaptation plan.

ReactionCoordinator and MechanismsRepository: Runtime Adaptation

The **ReactionCoordinator** is responsible for executing the adaptation plan provided by the **Balancer**. For this, it relies on the **MechanismsRepository** as shown in Figure 5.8.

The adaptation plan contains a list of actions that have to be executed to adapt a set of capabilities. Each action is associated with at least one capability (cf., Section 3.2.2) in a mapping table. The **MechanismsRepository** provides generic prototypes of actions to the **Balancer** for runtime negotiation while it provides executable instances for the **ReactionCoordinator** to deploy and execute. The interface to implement by each prototype is shown in Figure A.2 in the appendix.

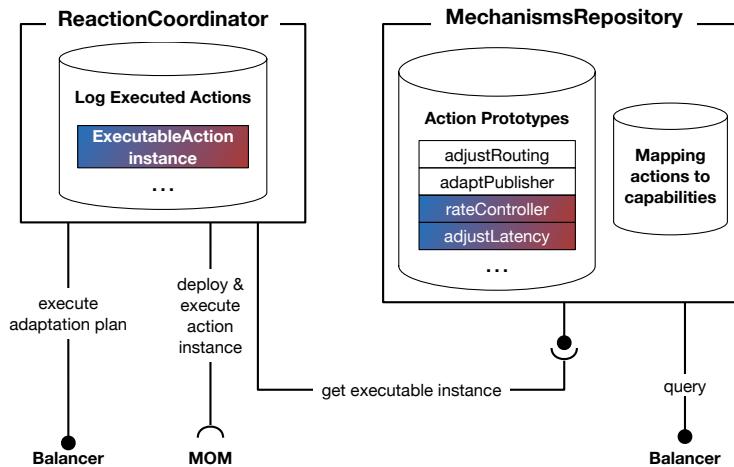


Figure 5.8.: Design of the ReactionCoordinator and attached MechanismsRepository.

Depending on the type of action, prototypes of actions can be agnostic or specific in regard to the MOM platform or application using the MOM as shown in Figure 5.8. Examples for generic actions are routing adaptation (`adjustRouting`) or publisher adaptation (`adaptPublisher`). The former action is generic as its execution is implemented by the MOM itself when implementing the `ExpectationController` interface; the latter action is generic as well as advising publishers to adapt is independent of the MOM platform or application (cf., Section 3.5.1).

Actions that require an adaptation of the MOM, however, have to be platform-specific at least. In some cases, they also have to be application-specific. For example, sampling rate reduction

(rateController) and forwarding latency minimization (adjustLatency) can be implemented to apply traffic shaping algorithms as described in Section 4.4.1. The way that notifications can be intercepted is platform-specific and has to be implemented in the prototype of the action. The rateController prototype must be application-specific if traffic shaping should be done by aggregating the content of multiple notifications instead of discarding notifications: the semantics of the notifications exchanged by the application over the MOM have to be known to the rateController to be able to aggregate the content correctly. The general design of a rateController is shown in Figure 5.9: a rateController instance is deployed for a dedicated receiver and intercepts the notifications of a given type before the broker dispatches them. Intercepted notifications are aggregated or decimated.

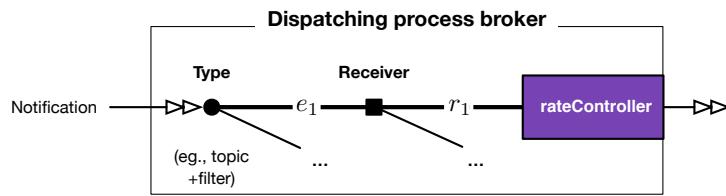


Figure 5.9.: Realizing the rateController per receiver and type.

The ReactionCoordinator sequentially executes the instances of the generic actions defined in the adaptation plan and keeps a log of all successfully executed actions for rollback operations.

The ReactionCoordinator also handles individual feedback to subscribers about the state of their expectations. As described in Section 3.5.1, this feedback can contain an acknowledgement about the expectation being already satisfied, a list of reasons why it is declined as unsatisfied, or a temporary state (*pending*), which indicates that the MOM is currently adapting to satisfy a satisfiable expectation.

Executing the adaptation plan and giving feedback to subscriber completes an iteration of the runtime support cycle for QoI requirements we have described in Chapter 4.

5.1.2 Decentralized Monitoring with ASIA

In addition to the individual feedback we provide about the expectations and capabilities of dedicated participants, we also provide aggregated feedback about the population and the system state in a DEBS using Application-specific Integrated Aggregation (ASIA).

The design of the ASIAController we use to support ASIA in our reference architecture is shown in Figure 5.10; it is a lightweight extension of the MOM compared to the ExpectationController discussed in the previous section.

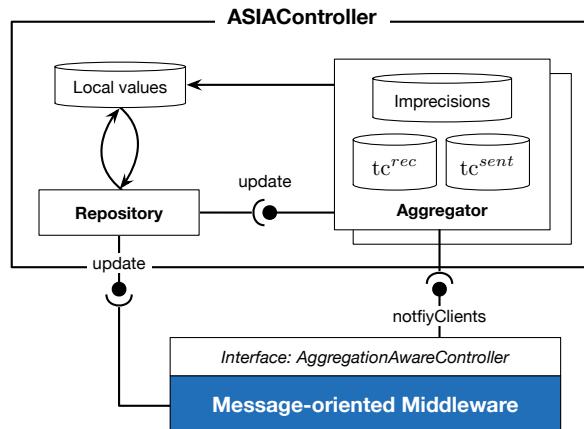


Figure 5.10.: Design of the broker component to enable runtime monitoring for DEBS with ASIA.

The components of an ASIAController are:

- A **Repository** that stores metadata about all connected participants and contains the routing tables of the adaptation tree as described in Section 4.5.2 to identify upstream and downstream participants.
- A hashtable of all local values that are of interest to aggregators and should be tracked by the **Repository**, e.g., the number of subscribers, the number of neighbors. Aggregators can add new entries if local values are not yet tracked.
- A list of **Aggregators** which encapsulate the aggregation functions of aggregated metrics that this broker instance can contribute to. Each aggregator manages the list of downstream receivers for updates about its aggregated metric together with the aggregated results for this metric received from upstream brokers and the last updates sent downstream.

The **Repository** updates all local values that change when being triggered by the broker. The **Repository** sequentially triggers all registered **Aggregators** if local values change; if it receives aggregated results sent from upstream brokers, only affected aggregators are triggered. Each **Aggregator** tracks the changes to its computed total values and updates all downstream participants based on their individual imprecisions as described in Section 4.5.2.

Requests for aggregated metrics from brokers upstream in the aggregation tree or from participants directly connected to this broker are received by the MOM and forwarded to the **Repository**. Based on these requests, new aggregators are deployed by the **Repository**.

The **notifyClients** method encapsulates the business logic for sending separate update notifications or piggybacking update information on existing notifications, advertisements or subscriptions. The broker also has to implement the business logic for detecting piggybacked updates.

The ASIAController interacts with the broker of a (distributed) MOM by the broker implementing the AggregationAwareController interface as shown in Figure A.1b in the appendix.

5.1.3 Libraries, Handlers and Editors Provided to Clients

We provide participants with libraries and handlers to deal with feedback by the middleware and use platform-agnostic APIs for managing the lifecycle of expectations and capabilities. The ExpectationHandlerClient provided to subscribers is shown in Figure 5.11. It allows subscribers to store, load, register, revoke, update, suspend, or resume expectations; requesting and receiving aggregated feedback is handled as well.

The CapabilityHandlerClient provided to publishers is quite similar to the ExpectationHandlerClient. It enables publishers to store, load, register, revoke or update capabilities and access their usage statistics (cf., Figures A.4b and A.5 in the appendix). Due to the similarity between both client handlers we show only the design and interface of the ExpectationHandlerClient in more detail; the architecture of the CapabilityHandlerClient can be found in the appendix.

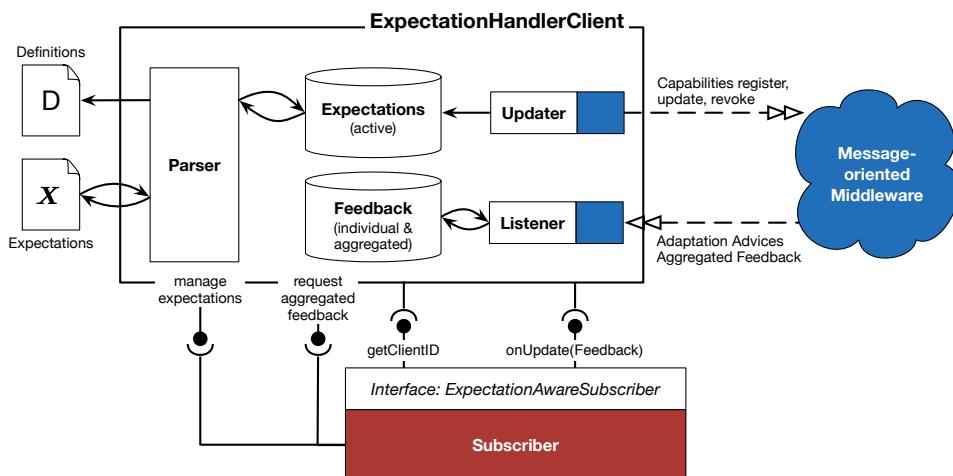


Figure 5.11.: Design of the ExpectationHandlerClient provided to subscribers.

The client libraries handle the loading of property definitions and predefined expectations or capability profiles into the participants and manage the platform-specific communication of lifecycle changes. Furthermore, they process the individual and aggregated feedback given to subscribers and publishers alike.

A client handler allows participants to register a callback method that is triggered by the client handler whenever new feedback arrives from the MOM. Subscribers can implement the `ExpectationAwareSubscriber` interface to be triggered for updates about their expectations or aggregated feedback requests; publishers can implement the `CapabilityAwarePublisher` interface to be triggered by adaptation advices or aggregated feedback. The client handlers can be used without providing a callback, though. In the appendix, Figures A.3a and A.3b show the interfaces for subscribers and publishers.

While most of the offered functionality is platform- and application-agnostic as it is encapsulated within the client handlers, exchanging information with the MOM requires platform-specific components to be added to each client handler. The `Updater` component inside a client handler needs to convert expectations or capabilities into the platform-specific format accepted by the MOM; the `Listener` component, in turn, needs to convert back individual and aggregated feedback

from the platform-specific message format. Thus, we provide an abstract class for each type of client handler that has to be extended for platform-specific ports. In the appendix, Figure A.4a shows the API offered by the abstract class for subscribers and Figure A.3b for publishers.

ReactionManagerClient: a Wrapper for Publishers

We assume that most publishers using our approach are able to implement the **CapabilityAwarePublisher** interface to receive and process adaptation advices from the MOM in order to adapt the generic properties of their publications.

Not all publishers, however, are able to do this without severe changes to their implementation. Using legacy applications in Cloud environments, for example, is a particular case [173]. Thus, we provide a wrapper for publishers, called **ReactionManagerClient**, that transparently adapts the information provided by the publisher without having to change its implementation.

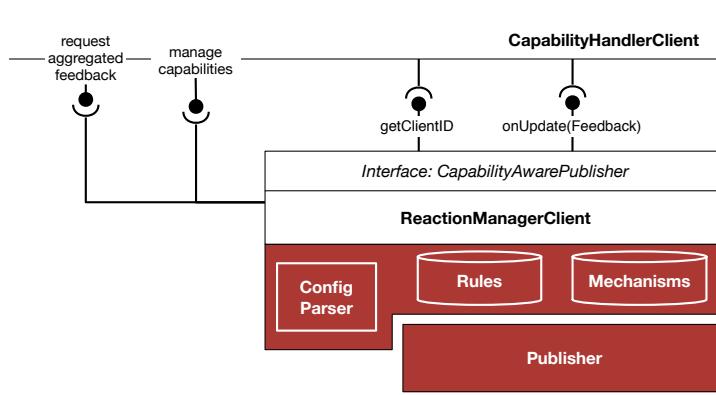


Figure 5.12.: Design of ReactionManager to enable adaptation of static publishers.

Each **ReactionManagerClient** consists of application-agnostic and application-specific parts as shown in Figure 5.12. The application-agnostic part of the **ReactionManagerClient** implements the **CapabilityAwarePublisher** interface and uses a **CapabilityHandlerClient** to handle feedback from the MOM. The application-specific part implements a set of rules (e.g., Event Condition Action rules (ECA) [401]), mechanisms, as well as a parser for the publisher's configuration. These components are used to determine the types of advertisements to send to the MOM as well as the capabilities provided by the publisher. Furthermore, they are required to interpret the received adaptation advises and adapt its associated publisher.

We apply this approach in our reactive McCAT middleware to support Ganglia monitoring and MySQL master-slave replication in public Cloud settings [173] (cf., Section 5.3).

Within our Ganglia scenario, for example, we have implemented a wrapper that changes the configuration of each gmond publisher on the fly before restarting it, realizing adaptation of the set of provided metrics, their granularity as well as the sampling rate within 26ms.

Graphical Editor to Define and Update Expectations and Capability Profiles

In addition to the client handlers we provide a platform-independent graphical editor written in Java to ease the management of expectations and capabilities. The editor imports definitions for generic properties as described in the Backus-Naur Form (BNF) of listing A.8a. Expectations and capabilities can be defined, altered or deleted using sliders and input fields. The form accepts only input that is conforming to the definition of the respective generic property. For example,

input fields for requirements or capabilities about the generic property *trustworthiness* defined over "low", "medium", "high" do not accept values such as "ultra" or "none". Figures 5.13 and 5.14 show screenshots of each type of editor. In a separate window of the editor, a star plot of the currently edited expectation or capability profile is kept up-to-date to give visual feedback to the user (cf., Section 3.3.1). The cost function of a capability is visualized as well.

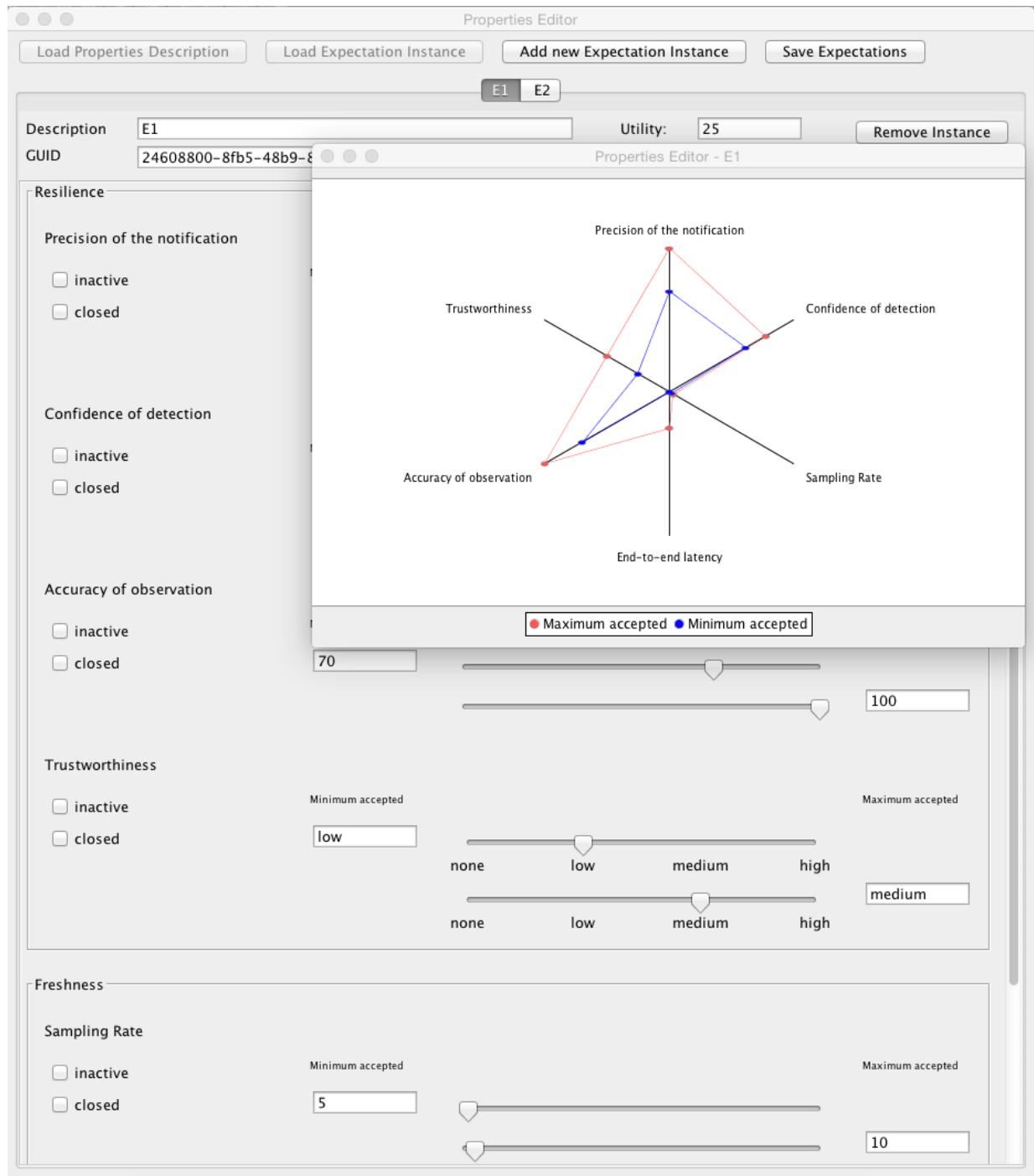


Figure 5.13.: Prototype of a graphical editor for expectations.

Expectations and capability profiles can be exported as Extensible Markup Language (XML) as shown in Listings 5.1 and 5.2. Definitions for generic properties are stored separately using a

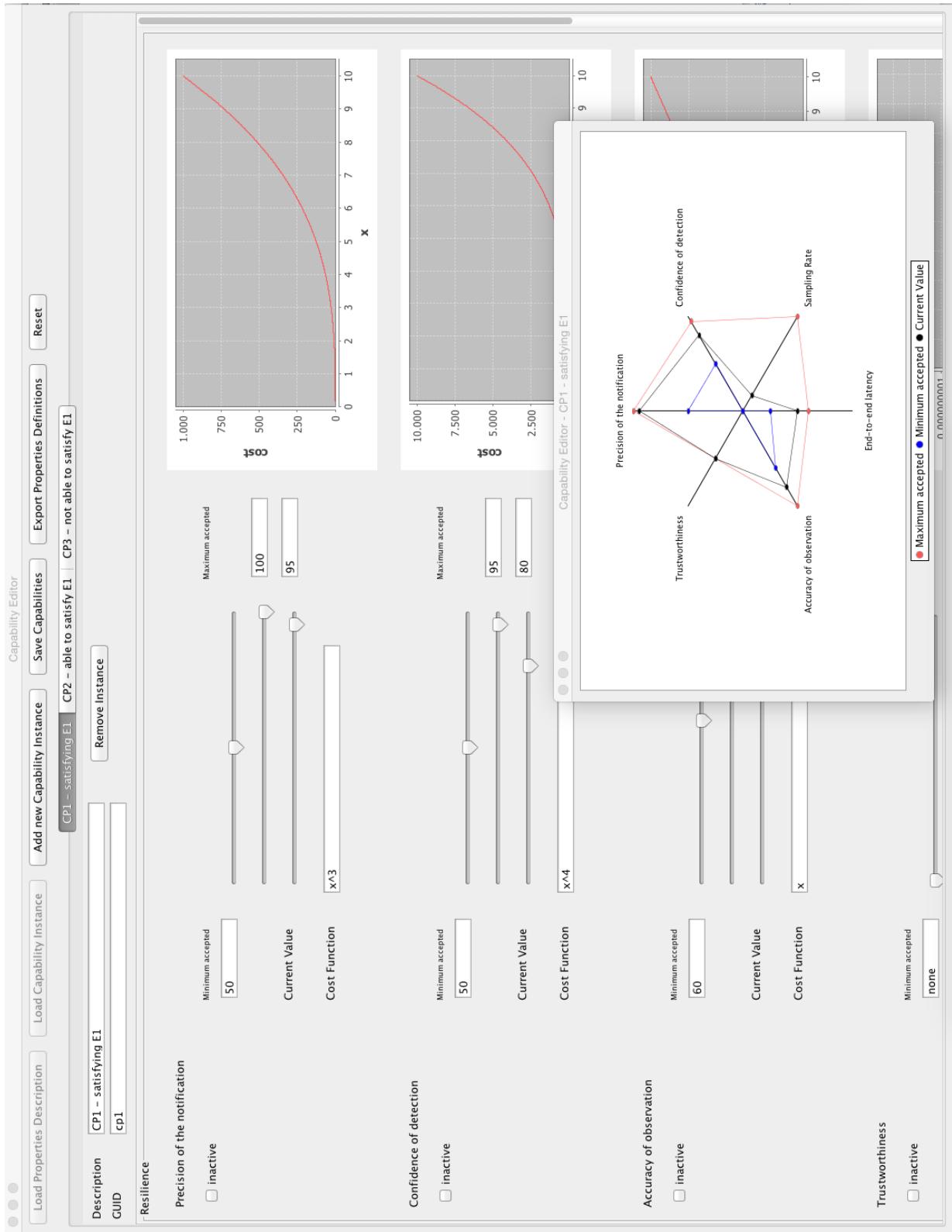


Figure 5.14.: Prototype of a graphical editor for capability profiles.

line-based syntax as shown in Figure A.8b. We chose these open formats for maximum portability but other formats can be easily integrated as well.

Listing 5.1: XML for an expectation.

```
<expectation utility="300" GUID="d496351b">
  <property abbrev="rate">
    <lower-bound>5</lower-bound>
    <upper-bound>10</upper-bound>
  </property>
  <property abbrev="confidence">
    <lower-bound>75</lower-bound>
    <upper-bound>95</upper-bound>
  </property>
</expectation>
```

Listing 5.2: XML for a capability profile.

```
<capabilityprofile GUID="5e5adf98" >
  <capability abbrev="confidence">
    <lower-bound>0</lower-bound>
    <upper-bound>80</upper-bound>
    <current-value>50</current-value>
    <costs>25+3x</costs>
  </capability>
  <capability abbrev="rate">
    <lower-bound>2</lower-bound>
    <upper-bound>60</upper-bound>
    <current-value>2</current-value>
    <costs>17x</costs>
  </capability>
</capabilityprofile>
```

5.1.4 Classes for Expectations, Capabilities and Generic Properties

The notion of expectations and capabilities based on generic properties is at the heart of our approach. While their semantics have been described in detail in Chapter 3, we want to discuss their implementation within our reference architecture in this section.

Expectations and capability profiles are handled as Java objects within the client handlers and the `ExpectationController`. For exchanging them between participants, these objects are either serialized (e.g., REDS) or converted into platform-specific message formats (e.g., `MapMessage` for Apache ActiveMQ). Figure 5.15 shows the methods and attributes most important for implementing the runtime negotiation process.

Each `Expectation` contains a hashtable of `XProperty` objects that embody the requirements about generic properties by defining upper and lower bounds on each generic property. The `XProperty` class provides the arithmetic functions required during the range matching step of the runtime negotiation phase.

Each `CapabilityProfile` contains a hashtable of `Capability` objects. Each `Capability` is an extended `XProperty` with additional cost function and current value of the same type of value that the generic property is defined over. In particular being able to define virtual current values valid only for specific expectations and adjusting the cost function are the key features used by `Balancer` and `ReactionCoordinator` during coordinated runtime adaptation (cf., Section 4.4.3).

The cost function of a capability is implemented using the `Calculable` class of the open-source Java library `exp4j`² that parses numeric functions described as Strings into executable objects.

Within each of these objects, we use the generic `Value` class to store values of generic properties. A `Value` is a wrapper for list-based or range-based values of arbitrary data types as described in Section 3.2.2; it also provides all methods to perform algebraic operations as shown in Figure 5.16. At the time of writing, `Value` supports `String`, `Integer`, `Double`, `Long`, and `Boolean`. List-based values can be defined as well by providing a list of range-based `Value` objects in ascending order.

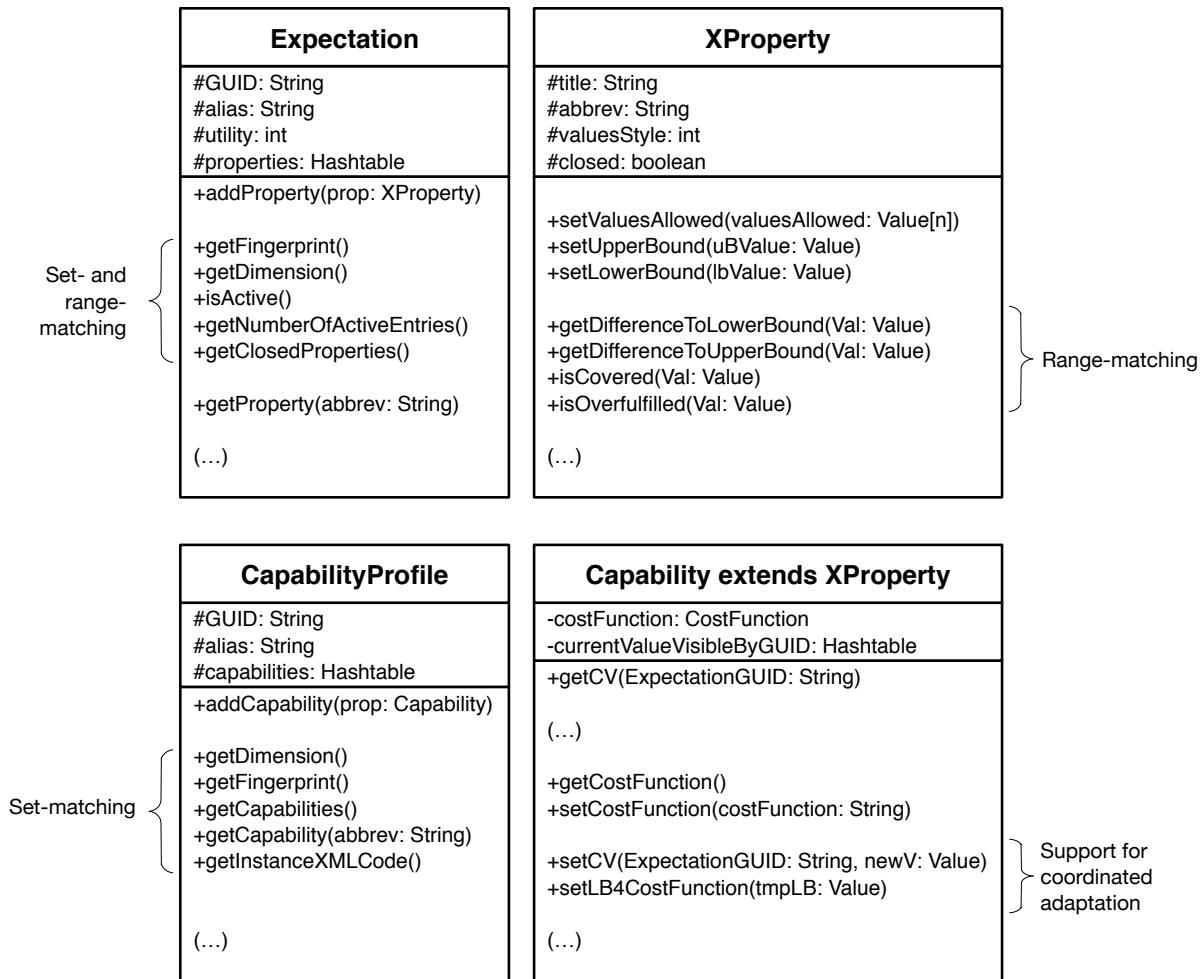


Figure 5.15.: Classes for expectations, capabilities, capability profiles and generic properties.

² <http://www.objecthunter.net/exp4j/>

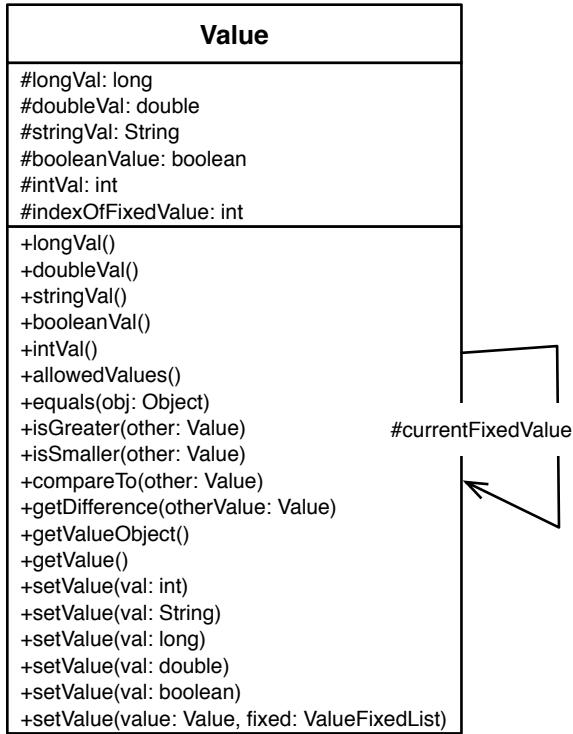


Figure 5.16.: Value: a wrapper for list-based and range-based values.

5.2 Platform-Specific Prototypes

We have implemented platform-specific components for the reference architecture described in the previous section to enable runtime support of QoI requirements on two open-source MOMs.

This section covers the support we have added for expectations, capabilities and feedback for self-adaptation within the Apache ActiveMQ messaging broker and the REDS MOM. Both platforms support a mixed type/content-based communication model and have been extended without affecting existing code.

We chose these two platforms for their different features: Apache ActiveMQ is representative of a centralized industrial-strength MOM focusing on high performance, while the distributed REDS MOM allows for exploiting routing strategies and broker topologies for adaptation in DEBS.

5.2.1 Centralized Implementation: Apache ActiveMQ

Apache ActiveMQ³ is an industry-strength open-source Java Message Service (JMS) MOM, which also supports other messaging protocols such as Advanced Message Queuing Protocol (AMQP), Simple Text Oriented Messaging Protocol (STOMP), and OpenWire [390]. Topics are used for many-to-many communication while queues implement point-to-point communication. Content-based routing is supported by adding attributes to notifications and defining filters on these attributes; explicit advertisement and unadvertisements, however, are not supported.

Using a plugin architecture, ActiveMQ allows to extend the MOM with custom functionality without the need to change existing code. A plugin has access to the state of the broker instance it is running on to intercept and modify any notification being processed by the broker. Furthermore,

³ <https://activemq.apache.org>

events indicating changes in the lifecycle of other participants (i.e., subscribers, publishers, or brokers) can be intercepted and modified as well. ActiveMQ is using an interceptor stack and container model to provide full access to its internal routing while isolating the business logic of a plugin and easing parallelization at the same time. Thus, multiple plugins can be deployed together without causing side-effects as illustrated in Figure 5.17 for **ExpectationController** plugin, **ASIAController** plugin and the **Latency** plugin described in [138].

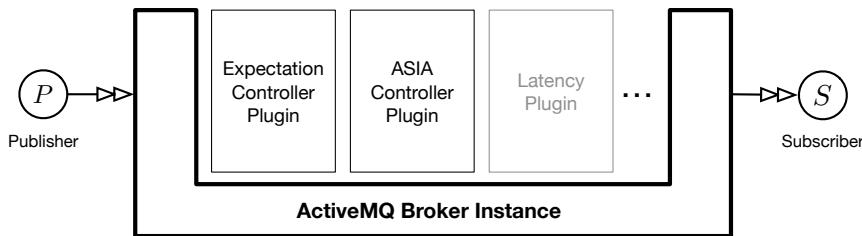


Figure 5.17.: Plugin-support on Apache ActiveMQ.

Due to this modular plugin architecture, we provide **ExpectationController** and **ASIAController** as two separate plugins on ActiveMQ.

Access to the broker state is offered by ActiveMQ through the API of the **BrokerPluginSupport** class that can be extended by each plugin. Selected parts of the API are shown in Figure A.6 in the appendix.

ActiveMQ is widely used as a centralized MOM. While clusters and networks of brokers can be supported for availability reasons, ActiveMQ itself does not support any higher-level routing strategies other than forwarding [302]: brokers subscribe on behalf of their local subscribers to all their neighbors while publications are flooded to all brokers. Within the business logic of ActiveMQ there is no distinction between edge and inner brokers as shown in Figure 5.18 for forwarding notifications. This is relevant for the interaction between the broker and our extensions **ExpectationController** and **ASIAController**.

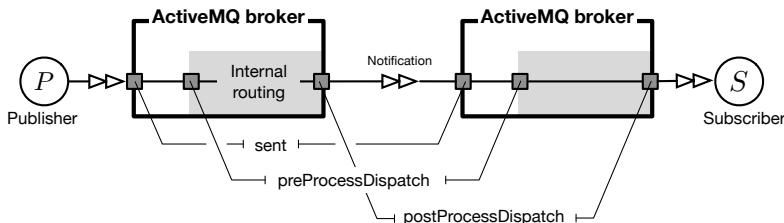


Figure 5.18.: Forwarding of notifications between multiple brokers using the same methods.

ExpectationController Plugin

We provide **ExpectationController** as a plugin on ActiveMQ by extending **BrokerPluginSupport** and implementing the **ExpectationController** interface. The integration of the **ExpectationController** with methods of ActiveMQ's API is shown in Figure 5.19.

The **ResourceMonitor** has to be triggered by changes of the broker state. Thus we add triggers to the methods of the ActiveMQ broker that are called whenever subscribers, publishers, or brokers join the system (`addConsumer`, `addPublisher`, `addBroker`) or leave (`removeConsumer`, etc.) as well as whenever topics or queues are added (`addDestination`) or removed.

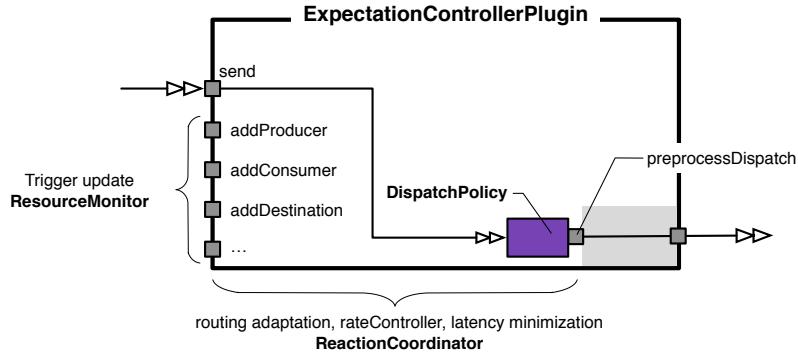


Figure 5.19.: ExpectationController: utilized joinpoints in ActiveMQ plugin API.

Notifications about expectations and capabilities are received and forwarded as MapMessages by a separate JMS subscriber that is subscribed on two topics provided on the broker: `expectations.control` and `capabilities.control`. The client handlers publish on these topics to register, update or revoke expectations and capabilities. The MOM publishes feedback on the topics `expectations.feedback` and `capabilities.feedback` that client handlers are subscribed to. We decided to use separate topics for the communication about expectations and capabilities to minimize the number of processed notifications we have to parse.

Runtime adaptation is realized by transparently changing the subscriptions of subscribers for routing adaptation and by using a customized `DispatchPolicy` to deploy `rateController` instances for those subscribers that require a reduction of the sampling rate.

On ActiveMQ, adjusting the `DispatchPolicy` is a powerful tool to transparently and efficiently intervene with the processing of notifications. At least one `DispatchPolicy` instance is assigned to a new destination. ActiveMQ hands over each notification that is processed for this destination to the `activeDispatchPolicy` together with a set of subscribers for this destination. The `DispatchPolicy` then decides for each subscriber if the notification should be dispatched.

We inject the `rateController` mechanism into the dispatching process of ActiveMQ by substituting the generic `DispatchPolicy` with the `RateControllerDispatchPolicy` shown in Listing A.1. This customized policy maintains a list of `rateControllers` registered for dedicated subscribers and types of notifications. If there is no `rateController` for a subscriber, the original notification is forwarded without alterations. The injection is done when a new destination is registered as shown in Listing A.2, lines 11-14. The listings are shown in the appendix.

ASIAController Plugin

We provide ASIAController as a plugin on ActiveMQ in the same way by extending `BrokerPluginSupport` and implementing the `ASIAController` interface.

As for the `ExpectationController` plugin, changes to the population known to the broker as well as incoming notifications trigger changes to the Repository. The Repository updates the local values for different aggregators and triggers the aggregators. Figure 5.20 shows which methods of ActiveMQ's API trigger the six aggregators we have implemented for ASIA.

Incoming aggregated results sent by upstream brokers on the aggregation tree are detected in the `sent` method of the plugin. They result in updates of their respective aggregates. The same method is used to forward updates about aggregated metrics downstream the aggregation tree.

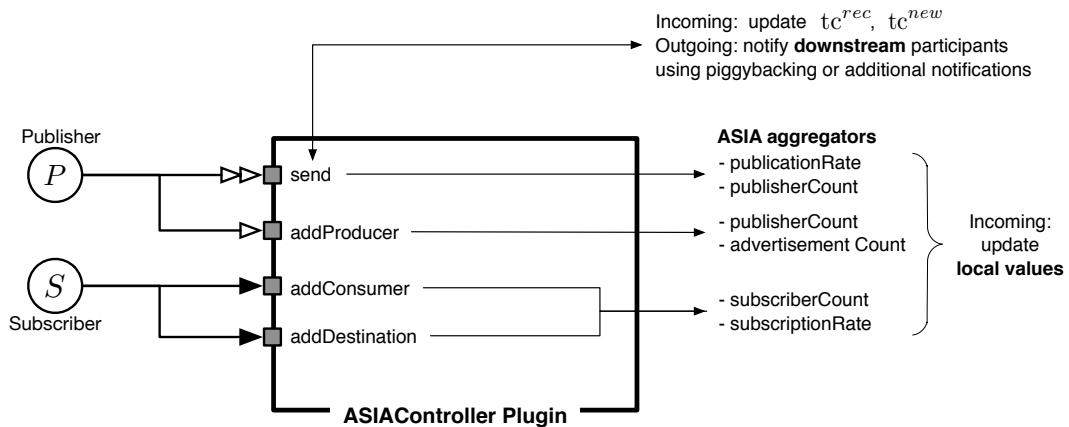


Figure 5.20.: ASIA aggregators: utilized joinpoints in ActiveMQ plugin API.

Aggregate update information is piggybacked onto existing JMS messages by adding attributes with a common prefix that describe the aggregated metric, its upper and lower bounds as well as the imprecision used to compute the interval. These additional attributes are stripped at the receiving broker. Subscribers and publishers are notified with separate MapMessages.

Extending Publishers and Subscribers

In addition to platform-specific implementations for brokers and client handlers, developers need to extend the code of existing publishers and subscribers to

1. access `ExpectationHandlerClient` and `CapabilityHandlerClient`;
2. define expectations and capabilities and register them at the MOM; and
3. react to feedback sent from the MOM.

Reacting to feedback is based on the semantics of the application and thus requires application-specific extensions that can differ in extent and complexity. Accessing the client handlers and managing the lifecycle of expectations and capabilities, on the contrary, is not.

Listing 5.3 shows a minimalistic example of a JMS subscriber that uses expectations to manage its QoI requirements. We omit most of the additional boilerplate code used to manage the lifecycle of the subscriber and the JMS connection itself. We assume that the subscriber subscribes to a predefined topic with a given filter expression; the approach is the same for queues.

The additional code necessary to integrate our `ExpectationHandlerClient` into the business logic of a subscriber consists of general bootstrap code to initialize the client handler (line 15 - 16) and load a list of predefined expectations saved in a separate file. Alternatively, the subscriber can define expectations on the fly after initializing the client handler. For every expectation to be activated for a given JMS topic and filter expression, the subscriber needs only a single line of code (line 17). The subscriber identifies itself at the MOM using the `clientID` attribute of the JMS connection. This identifier is valid for all subscriptions of the connection.

We want to point out the ratio between the *eight* lines of code necessary to establish a connection to a typical JMS MOM (lines 21-28) and the additional *three* lines of code necessary to include expectations (lines 15-17).

In our example, every update about the state of a registered and active expectation triggers the subscriber to print a list of all the feedback received so far using the method provided by the client handler (line 35).

Listing 5.3: Minimalistic JMS subscriber using expectations.

```
1 public class SubscriberExpectationAware implements ExpectationAwareSubscriber,
2     MessageListener {
3
4     private Connection connection;
5     private Session session;
6     private Topic topic;
7     private MessageConsumer subscriber;
8     private ExpectationHandlerClientActiveMQ eH;
9
10    (...)

11    public SubscriberExpectationAware(String url, String clientID, String topicName, String
12        filter, String xID, String configFile) throws JMSException {
13
14        this.setupJMS(url,clientID,topicName,filter); //establish JMS connection
15
16        eH = new ExpectationHandlerClientActiveMQ(this, configFile, true);
17        eH.addLocalExpectationsAuto(); //load all expectations in this file
18        eH.registerExpectation(xID, new EventType(topic,filter));
19    }
20
21    private void setupJMS(String url, String clientID,filter) {
22        ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(url);
23        connection = factory.createConnection();
24        connection.setClientID(clientID);
25        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
26        topic = session.createTopic(topicName)
27        subscriber = session.createConsumer(topic,filter)
28        subscriber.setMessageListener(this);
29        connection.start();
30    }
31
32    (...)

33    @Override //handle feedback
34    public void onUpdate(FeedbackExpectation news) {
35        System.out.println(this.expectationHandler.summarizeFeedback());
36    }
37
38    (...)
```

Listing 5.4 shows a minimalistic example of a JMS publisher that uses capabilities to announce its support for QoI properties. As for the subscriber example, we omit most of the additional boilerplate code used to manage the lifecycle of the publisher itself. Again we assume that the publisher publishes to a single predefined topic; the approach is the same for queues or multiple topics/queues.

As in our subscriber example, we need only two additional lines of code to bootstrap the client handler and load a list of predefined capabilities (line 15 + 16) while a single line of code is required to register a capability profile for a given JMS topic (line 17).

In our example, receiving an adaptation advice from the MOM (line 32) triggers publisher-specific code for adapting all capabilities required to change. Having completed the adaptation, the publisher updates the current value for the respective capability and notifies the MOM using the client handler (line 37).

Listing 5.4: Minimalistic JMS publisher using capabilities.

```
1 public class PublisherCapabilityAware implements CapabilityAwarePublisher {
2
3     private Connection connection;
4     private Session session;
5     private Topic topic;
6     private MessageProducer publisher
7     private CapabilityHandlerClientActiveMQ cH;
8
9     (...)

10
11    public PublisherCapabilityAware(String url, String clientID, String topicName, String
12        filter, String cID, String configFile) throws JMSEException {
13
14        this.setupJMS(url,clientID,topicName);
15
16        cH = new CapabilityHandlerClientActiveMQ(this, configFile, true);
17        cH.addLocalCapabilitiesAuto();
18        cH.registerCapabilityProfile(cID, new EventType(topic));
19
20    private void setupJMS(String url, String clientID) {
21        ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(url);
22        connection = factory.createConnection();
23        connection.setClientID(clientID);
24        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
25        topic = session.createTopic(topicName)
26        publisher = session.createProducer(topic,filter)
27        publisher.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
28        connection.start();
29    }
30
31    @Override
32    public void onUpdate(AdaptationAdvice news) {
33        for (RequirementForCovering req : news.getRequirements().values()) {
34
35            (...)

36
37            this.capabilityHandler.updateCapabilityCurrentValue(news.getAssociatedGUID(), req.
38                getAbbrev(), req.getRequiredValue());
39        }
40
41    (...)
```

5.2.2 Distributed Implementation: REDS

Our second set of platform-specific extensions provides support for expectations, capabilities and feedback in the distributed REconfigurable Dispatching System (REDS)⁴. REDS is an academic MOM written in Java which aims at providing a lightweight distributed and decentralized architecture that is easy to extend for custom approaches [126].

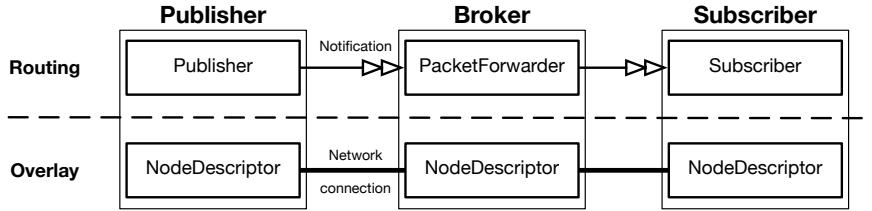


Figure 5.21.: REDS general architecture: routing and overlay layers.

The general architecture of REDS consists of an *overlay* layer and a *routing* layer as shown in Figure 5.21. On the overlay layer, network connections between participants are established and managed by *NodeDescriptors* that form a topology. Different transport protocols can be used to establish the overlay, e.g., TCP or UDP. On this layer, participants do not have different roles but each *NodeDescriptors* can send and receive notifications that consist of a subject and any *Serializable* Java object as payload.

On the routing layer, *PacketForwarder* classes establish a routing direction from publishers to subscribers based on the connections that are maintained in the overlay layer. Routing tables for storing and matching advertisements and subscriptions are maintained by instances of the *PacketForwarder* class that acts as broker. Any notification, advertisement or subscription processed by a *PacketForwarder* can be modified by it.

The broker API becomes accessible to *ExpectationController* and *ASIAController* by extending *PacketForwarder*. The methods are shown in Figure A.7 in the appendix. As on ActiveMQ, our extensions can intercept or generate subscriptions, advertisements, and notifications.

In contrast to ActiveMQ, however, REDS brokers use different methods based on whether they act as edge brokers or inner brokers in a DEBS during the delivery of notifications. The interlock of the different methods is illustrated in Figure 5.22 for the delivery of a single notification from a publisher (left) to a subscriber (right) via three brokers: while all three brokers provide the same methods, different methods are used based on their role. As shown in the annotations to Figure A.7, we can utilize this knowledge when updating ASIA aggregators.

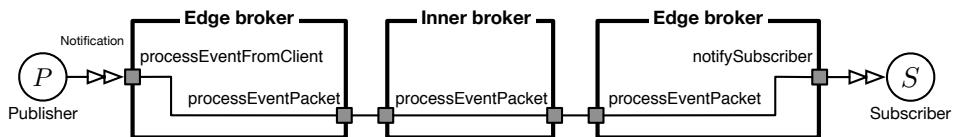


Figure 5.22.: Difference between edge and inner brokers in REDS.

⁴ <http://zeus.ws.dei.polimi.it/reds/>

ExpectationController and ASIAController combined

As REDS does not support interceptor chains with plugins as ActiveMQ does, we have integrated the functionality of both `ExpectationController` and `ASIAController` into a single extended `PacketForwarder` class by implementing both the `ExpectationController` interface and the `ASIAController` interface.

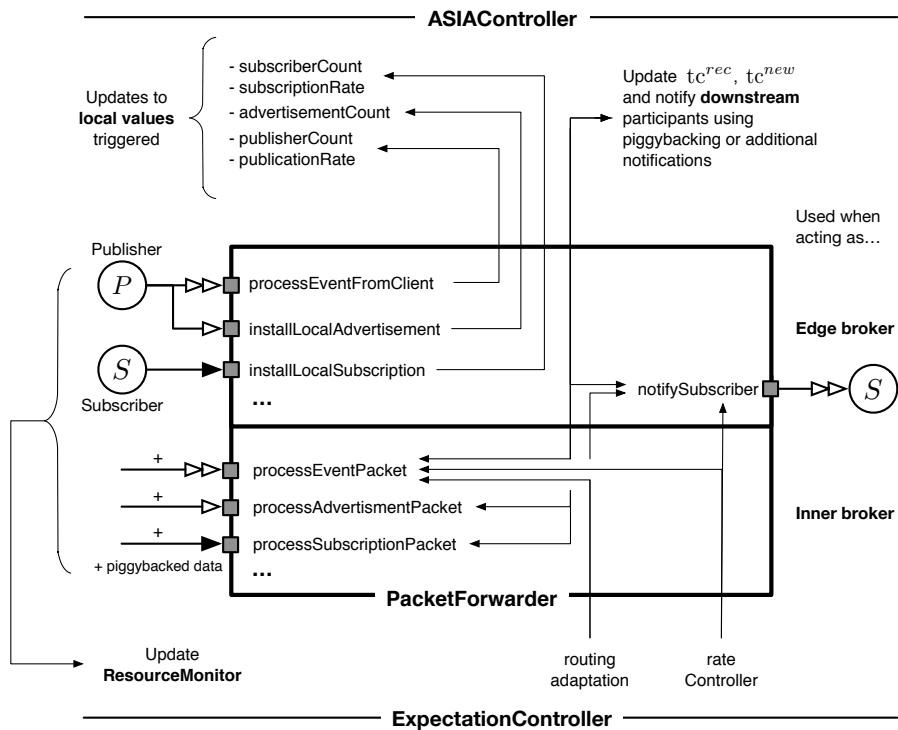


Figure 5.23.: Joinpoints of a REDS broker and their use for ASIA and QoL requirements.

Figure 5.23 shows how our extended `PacketForwarder` interacts with `ExpectationController` and `ASIAController`. When acting as an edge broker (upper half of Figure 5.23), all methods triggered by participants such as connecting, disconnecting, advertising, subscribing or publishing triggers the Repository of the `ASIAController` to update the local values and update registered aggregators. When acting as an inner broker, in turn, any forwarded notification, advertisement or subscription with piggybacked update information can lead to total counts being updated at the `ASIAController`. Any updates for aggregated ASIA metrics are forwarded downstream the aggregation tree using `notifySubscriber` or hooking into the `processEvent` method.

The `ResourceMonitor` of the `ExpectationController` is triggered by any change to the state of the broker – regardless of whether the broker acts as edge or inner broker. The methods `processEventPacket` and `notifySubscriber` are essential for runtime adaptation in terms of routing adaptation or rate reduction. When acting as an inner broker, rate reduction is activated in `processEventPacket` to reduce the number of notifications being processed to a neighboring broker while `notifySubscriber` is used to deliver the notifications to a dedicated subscriber at the required sampling rate.

Extending Publishers and Subscribers

On REDS, the code extensions necessary for participants to access their client handlers and manage the lifecycle of their expectations or capabilities is the same as on ActiveMQ.

Listing 5.5 shows an example of a subscriber to the REDS MOM that uses expectations to manage its QoI requirements.

The additional code necessary to give the subscriber access to the `ExpectationHandlerClient` and load predefined expectations (line 15 - 18) is the same as on ActiveMQ, as is the code to register a locally loaded expectation with a subscription (line 31).

Listing 5.5: Example for a REDS subscriber using expectations.

```
1
2 public class GangliaSubscriber extends Subscriber implements Runnable,
3   ExpectationAwareSubscriber {
4
5
6   ExpectationHandlerClientREDs expectationHandler;
7
8   (...)

9   public GangliaSubscriber(GenericRouter router, Overlay overlay, Monitor monitor,
10     GangliaInterceptor myController) {
11     super(router, overlay, monitor);
12
13   (...)

14   //Register ExpectationsHandlerClient
15   expectationHandler = new ExpectationHandlerClientREDs(this, "./reds-
16     expectationHandlerClient-local.conf", true, myController.asiaPortToListenTo+55);
17
18   //Load expectations as defined in the config file
19   loadedExpectations = expectationHandler.addLocalExpectationsAuto();
20
21 }
22
23 (...)

24 private void subscribeToAvailableMetrics(GangliaClusterData gcd) {
25
26   (...)

27   for (Subscription s : subs) {
28     sendSubscription(s, System.nanoTime(), SubPktType.SUB);
29     logger.finest("Send subscription for " + s.getType() + " with " + s.getConstraints().
30       size() + " constraints");
31
32     this.expectationHandler.registerExpectation(this.expectationsForTypes.get(s.toString())
33       , new EventType(s.getType().toString()));
34
35   (...)

36 }
37
38 }
```

Listing 5.6 shows code excerpts for a REDS publisher as part of our McCAT extension to the Ganglia monitoring system (cf., Section 5.3).

As in our subscriber example, we need only two additional lines of code to bootstrap the client handler and load a list of predefined capabilities (line 15 + 16) while a single line of code is required to register a capability profile for a given Ganglia metric that is being advertised by this publisher (line 16).

Listing 5.6: McCAT publisher using expectations on a REDS platform.

```
1  public class GangliaPublisher extends Publisher implements Runnable, CapabilityAwarePublisher
2  {
3
4      (...)

5
6      private CapabilityHandlerClientREDs capabilityHandler;
7
8      public GangliaPublisher(GenericRouter router, Overlay overlay, Monitor monitor,
9          GangliaInterceptor controller) {
10         super(router, overlay, monitor);
11
12         (...)

13         this.controller = controller;

14
15         capabilityHandler = new CapabilityHandlerClientREDs(this, "./reds-
16             capabilityHandlerClient-local.conf", true, controller.asiaPortToListenTo+55);
17         capabilityHandler.addLocalCapabilitiesAuto();

18     }
19
20     (...)

21
22     /* Method sends an advertisement for a specific Ganglia metric */
23     public void advertiseAvailableMetric(GangliaMetricData metric) {
24
25         (...)

26
27         Advertisement adv = new Advertisement(type, controller.currentGangliaConfiguration.
28             gmondData.getApplicableAggregations().get(metric.getMetricName()), controller.
29             currentGangliaConfiguration.gmondData.getAdditionalParametersForMetric(metric.
30                 getMetricName()).get("attributeToAggregate"), host, cluster);

31
32         clientAdvertisedTypes.add(type);
33         sendAdvertisement(adv, System.nanoTime());

34
35         //Register corresponding CapabilityProfile if any has been defined in gmond-config using
36         //CPGUID option
37         if (controller.currentGangliaConfiguration.gmondData.getAdditionalParametersForMetric(
38             metric.getMetricName()) != null) {
39             String cpGUID = controller.currentGangliaConfiguration.gmondData.
40                 getAdditionalParametersForMetric(metric.getMetricName()).get("CPGUID");
41             this.capabilityHandler.registerCapabilityProfile(cpGUID, new EventType(type.toString()))
42         );
43     }
44 }
```

5.3 Example Applications

We use our platform-specific prototypes to support QoI at runtime within different distributed open-source applications written in Java. In the remainder of this chapter, we illustrate the integration of our approach into the reactive McCAT middleware to enhance Ganglia monitoring and MySQL master-slave replication in public Cloud environments. Furthermore, we describe the extensions made to the benchmarking applications FINCoS and jms2009-PS.

McCAT: Enhance MySQL Replication and Ganglia in Public Clouds

Multi-cloud Cost-Aware Transport (McCAT) is an approach we propose to balance the network usage of distributed legacy applications with QoI requirements of a tenant in multi-Cloud deployments without having to change the application's implementation [173].

Many distributed applications that are used in public Cloud deployments were originally developed for use in environments where network traffic is sufficiently available and free of charge. Prominent examples are monitoring systems (e.g., Ganglia), data-parallel processing frameworks (e.g., Hadoop), and replicated databases (e.g., MySQL) deployed in clusters.

In today's public Clouds, network traffic has become a major cost factor for data-intensive applications: the Cloud provider charges a tenant for incoming and outgoing network traffic when operating the tenant's applications in one or multiple data centers. Consequently, the tenant needs to have means to balance the network usage of its applications with the QoI of the exchanged information, e.g., in terms of latency, completeness, or precision.

We propose McCAT to make distributed applications network- and cost-aware while enabling tenants to specify bandwidth budgets and QoI requirements. Instead of changing the implementation of the application, McCAT *transparently intercepts* data exchanged between components of the distributed application using application-specific interceptors as shown in Figure 5.24.

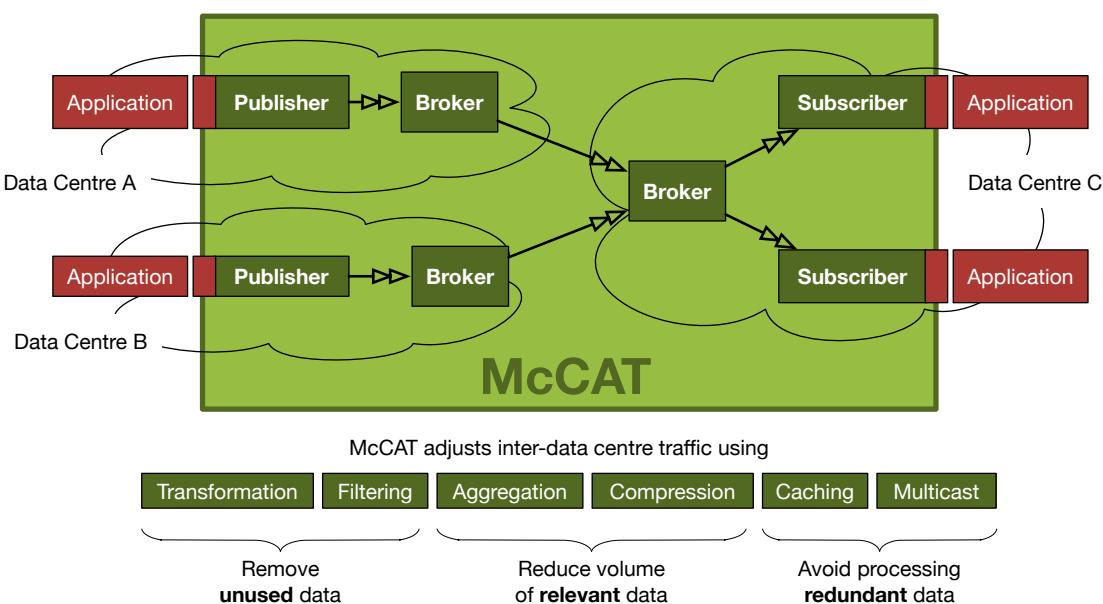


Figure 5.24.: McCAT: make distributed legacy applications network- and cost-aware.

McCAT tries to balance the network usage of the application with the QoI requirements of the tenant by *removing* unused data, *reducing* the volume of relevant data and *avoiding* to send redundant data. These objectives are achieved by McCAT dynamically:

1. *transforming* meaningful data from application-specific formats into generic notifications;
2. *filtering* out data that is not required by any receiving component;
3. *compressing* and *aggregating* relevant data before sending;
4. *caching* data and processing it with application-level multicast.

For example, in a Ganglia system deployed across multiple data centers, network traffic between data centers can be significantly reduced by aggregating multiple updates of monitored metrics before transmission if their sampling frequency is higher than the received rate required at the moment (e.g., sampling frequency of one update every five seconds vs. required granularity of five minutes' average), or by filtering metrics that are not being displayed to user.

Similarly, a master-slave replication deployment of MySQL across multiple Amazon EC2 regions can save bandwidth by multicasting the same replication update to multiple slaves, processing updates for selected tables only, or by delaying updates for a given amount of time to clear out redundant updates: if multiple updates to the same row of a table occur within a given amount of time, only the last update is processed.

Architecture Overview

McCAT is realized as a distributed application consisting of McCAT *interceptors* that act as publishers or subscribers in a DEBS and communicate via McCAT *brokers* as shown in Figure 5.24. Expectations are used to capture the QoI requirements of the tenant while publishers use capabilities. Adaptation advices sent by McCAT brokers adjust the publication behavior.

McCAT Interceptors: in McCAT, there are two types of interceptors: senders and receivers. They have different tasks but are quite similar in their design as shown in Figure 5.25.

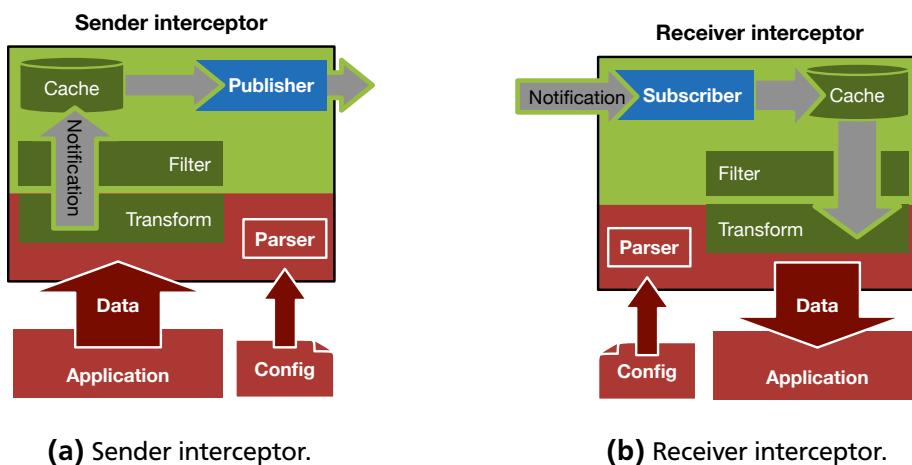


Figure 5.25.: Design of the two McCAT interceptor types.

Sender interceptors, as shown in Figure 5.25a, act as publishers in McCAT. Each interceptor uses application-specific components to mimic a receiving component (e.g., a MySQL slave) and intercept data from a sending component of the application. The McCAT interceptor then extracts and transforms the meaningful data items from the application-specific wire-format, such

as the binary format of MySQL replication updates or the XML schema of Ganglia monitoring updates, into generic notifications used by McCAT. These notifications are filtered, to remove currently irrelevant data, and cached, to purge redundant data.

A platform-specific publisher component sends notifications to all subscribers via McCAT brokers. Each publisher subscribes to ASIA aggregations about the number of subscribers interested in the data it publishes; notifications are published only if there is at least one matching subscriber active. Adaptation advices received from the system adjust the behavior of the interceptor, either by using a `ReactionManagerClient` to adapt the sending component of the application (cf., Section 5.1.3), or by adapting the publisher within the interceptor.

Sender interceptors use an application-specific parser and a set of rules to analyze the configuration of the sending component. They extract the information necessary to intercept the sending component, identify relevant data items, and generate appropriate advertisements with capabilities. This way, McCAT can leverage application-specific knowledge while respecting application semantics when filtering and caching.

Receiver interceptors, as shown in Figure 5.25b, act as subscribers in McCAT. Each receiver interceptor is associated with a receiving component of the application and mimics a sending component (e.g., a MySQL master). Internally, a platform-specific subscriber receives notifications from McCAT sender interceptors that it caches and filters. Before processing the update to the receiving component of the application, the interceptor transforms the data contained in the notification back into the application-specific wire-format.

Like sender interceptors, receiver interceptors parse the configuration of their associated components for clues about the data these components expect to receive. Based on this knowledge and the tenant's specification, subscriptions and expectations are generated and registered.

McCAT Brokers: realize the communication between sending and receiving components of the application. Each broker is extended by an `ExpectationController` to negotiate expectations with capabilities as well as an `ASIAController` to provide aggregated feedback to subscribers. Brokers are network- and cost-aware, i.e., they know which connections the cloud provider charges for and what the current utilization of these connections is. They apply lossy and lossless aggregation mechanisms to the notifications they process to reduce traffic whenever it becomes necessary to remain within the bandwidth budget set by the tenant.

Prototypes

We provide two prototypes of McCAT based on the ASIA extension of REDS [172]. They provide support for the Ganglia⁵ monitoring system [298] and MySQL master-slave replication⁶.

McCAT for Ganglia: the prototype of McCAT intercepts the communication between `gmond` and `gmetad` agents. In a Ganglia system, `gmond` agents monitor the state of the host (virtual) machine they are deployed on using different metrics. They update interested `gmetad` agents on remote machines with a configurable update frequency when multicast is available. In environments, such as public Clouds, where multicast is not available, `gmetad` agents query `gmond` agents at a given frequency following a multi-step protocol (request, acknowledgement, response, acknowledgement) over a given TCP socket. `Gmond` agents send an application-specific XML reply. Each interaction is stateless, i.e., `gmond` agents always send the full set of metrics they have measured even if the state of certain metrics has not changed since the last update.

⁵ <http://ganglia.sourceforge.net/>

⁶ <https://dev.mysql.com/doc/refman/5.1/en/ha-vm-aws-deploy.html>

In McCAT4Ganglia, we provide two kinds of interceptors: `gmondInterceptor` (sender interceptor) and `gmetadInterceptor` (receiver interceptor). They rely on existing Ganglia configuration files. We exploit the syntax of these configurations to store additional information for McCAT, such as explicit subscriptions (i.e., subscribe to a set of metrics, all metrics of a given host or a given cluster only), bandwidth budget, expectations to use, capability profiles to build on, or the types of aggregations to use for specific metrics. The information is added in a format that is ignored by an out-of-the-box Ganglia deployment but recognized by the application-specific parsers of McCAT. Listings 5.7 and 5.8 show examples for a gmond as well as a gmetad configuration file.

Listing 5.7: Extended gmond configuration for McCAT to use aggregation and capability profiles.

```

1 # Lines starting with # are ignored by Ganglia. McCat ignores only those lines
2 # where # is followed by at least one space. All other lines starting with # are
3 # considered to be additional configuration parameters.
4
5 (...)

6
7 collection_group {
8
9   # sampling frequency in seconds for all metrics in this group
10  collect_every = 5
11
12 metric {
13   name = "cpu_num"
14   title = "CPU Count"
15   #aggr = AVG
16   #attributeToAggregate = value
17   #CPGUID=cp1
18 }
19
20 metric {
21   name = "cpu_speed"
22   title = "CPU Speed"
23   #aggregation = AVG
24   #attributeToAggregate = value
25   #CPGUID=cp1
26 }
27
28 (...)

29 }
```

Listing 5.8: Extended gmetad configuration for McCAT to use subscriptions and expectations.

```

1 # Sebastian Frischbier:
2 # We make use of the syntax here to code additional information for McCAT into
3 # a valid gmond configuration file that can be used without McCAT as well.
4
5 # The first hostname is the host actually queried by gmetad.
6 # Only if this host fails to reply, the other hosts following that are queried
7 # For using the McCAT4Ganglia adapter, this very first host should be localhost.
8 # Otherwise, the adapter will show a warning message.
9 # You can check the correct settings from the adapter's output on the terminal
10 # Unused configurations can be marked with a # at the start of each line
11
12 data_source "ASIA1" 1 localhost:8649
13 # data_source "ASIA2" 1 localhost:8888 ec2-54-200-192-151.us-west-2.compute.amazonaws.com
```

```

14
15 # Experimental syntax for fine-granular subscriptions (metric & host level) => sub:c=[  

16   cluster]:h=[host]:m=[metric]:expectation=[GUID]
17 # if c,h,m or f are not explicitly set, wildcard * is assumed
18 # if no explicit subscriptions are used or --normalsyntax is set, the gmetadInterceptor for
19   this gmetad subscribes to all available metrics from all hosts and clusters
20
21 # Example: subscription with expectation: cpu_idle metrics from all hosts of ASIA1 that
22   conform to expectation ec1
23
24 RRAs "RRA:LAST:0.0:1:100000" "RRA:AVERAGE:0.5:1:5856" "RRA:AVERAGE:0.5:4:20160" "RRA:AVERAGE
25   :0.5:40:52704"
26 gridname "ASIADC1"
27 all_trusted on
28 setuid_username ganglia
29 xml_port 8657
30 case_sensitive_hostnames 0

```

McCAT for MySQL Master-Slave Statement-based Replication: the prototype of McCAT intercepts the communication between a MySQL replication *master* database and multiple MySQL replication *slave* databases. In MySQL, slave databases can register at a master to be informed about any change to rows in a specific database, e.g., INSERT, UPDATE, DELETE, DROP. Whenever such an update happens at the master, all slaves are notified by the master and receive a binary *BinLogEvent* containing the query causing the replication update; this statement-based replication is the default configuration for MySQL. The master and all slaves log each query to a BinLog record that can be used for rollback operations or to update slaves with outdated states.

In McCAT4MySQL, we provide two kinds of interceptors: `mysqlSlaveInterceptor` (receiver interceptor) and `mysqlMasterInterceptor` (sender interceptor). The interceptors parse the configurations of their database instances to learn the TCP ports used for replication as well as the current state of the BinLog record.

The `masterInterceptor` registers at the MySQL master on behalf of the replication slaves. BinLog events received from the master are buffered here until a sequence of interrelated events is complete and then forwarded by the publisher.

At the edge broker, BinLog events referring to the same table in a replicated database are cached. The size of the cache in terms of the number of BinLog events to add is determined by McCAT. Redundant replication actions currently kept in the cache are eliminated, e.g., only the latest update of subsequent updates for the same tuple is kept. Varying the size of the cache enables McCAT to trade-off the latency and consistency of updates with traffic savings: the larger the cache, the higher the probability for redundant updates to be observed and eliminated. The latency of updates, however, increases with an increasing cache size while the consistency of the slave databases is deteriorating. The bounds defined by the tenant for these generic properties define effective upper bounds for the cache size.

JMS-2009-PS: Benchmark for JMS

The jms2009-PS [368, 369] benchmark extends the official Standard Performance Evaluation Corporation (SPEC) benchmark⁷ SPECjms2007 for JMS. The workload of the SPECjms2007 benchmark is based upon a Supply Chain Management (SCM) scenario where different types and instances of participants (i.e., different distribution centers, supermarkets, headquarters etc.) interact with each other using a MOM. The benchmark is described in detail in Section 6.3.

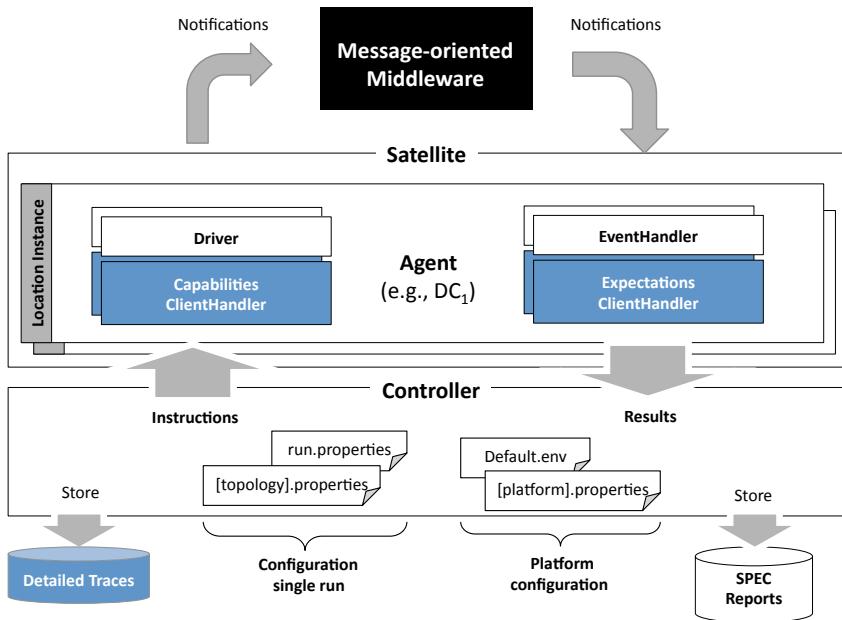


Figure 5.26.: jms2009-PS architecture and custom extensions (blue).

Figure 5.26 shows the design of SPECjms2007 and jms2009-PS: an agent encapsulates all publishers (drivers) and subscribers (event handlers) that represent a physical location (e.g., DC_1) in the scenario. A satellite manages all agents that represent a specific type of participant (i.e., headquarters, distribution centers etc.). A controller coordinates all satellites. The controller manages the lifecycle of its satellites and analyzes the results of a benchmark run.

Each driver and event handler is already designed as a dedicated JMS producer or JMS consumer in SPECjms2007. Hence, we have extended them as described in Section 5.2.1: each driver maintains its capabilities using a dedicated instance of `CapabilityHandlerClient` while subscribers do the same for expectations using their dedicated instance of `ExpectationHandlerClient`. This way, the additional load introduced by expectations and capabilities is tied to the benchmark's workload and scales with it.

FINCoS Benchmarking Tool

FINCoS [305] is an open-source benchmarking tool recommended by the SPEC Research Group⁸ to evaluate Complex Event Processing (CEP) engines and EBS infrastructures in terms of performance and correctness. FINCoS allows for defining and running complex custom workloads against different platforms without having to individually tailor or convert the same workload

⁷ <http://spec.org/jms2007/>

⁸ <https://research.spec.org/tools/overview/fincos.html>

for each targeted platform. In addition, FINCoS takes care of measuring and logging Key Performance Indicators (KPIs) such as latency or throughput of the notifications processed by the system under test. FINCoS is written in Java and designed as a distributed application where agents act as subscribers or publishers and thus simulate a heterogeneous population of an EBS.

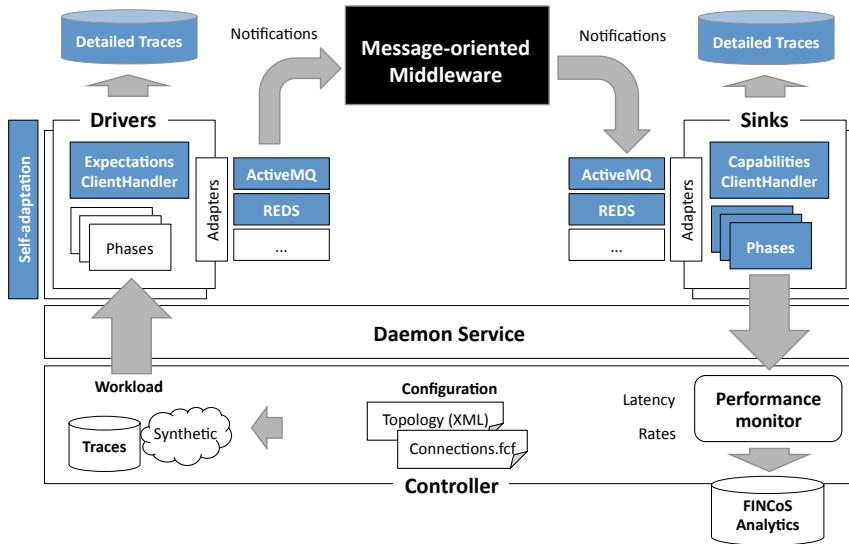


Figure 5.27.: Architecture and custom extensions (blue) of the FINCoS benchmarking tool.

As shown in Figure 5.27, load-generators (drivers) act as publishers in an EBS and send customized workloads to the system under test. Agents (sinks) act as subscribers based on predefined subscription patterns and receive the processed notifications. A FINCoS controller coordinates the lifecycle of all agents that act as subscribers or publishers. Publishers and subscriber in FINCoS can be distributed across different hosts for better performance. Hence, a daemon service is running on each host to connect subscribers, publishers and controller across hosts using Remote Method Invocation (RMI). A performance monitoring tool (perfmon) provides analytics such as the deviation of throughput and latency of the notifications received by subscribers over the course of the experiment.

FINCoS uses adapters to connect to CEP engines or EBS infrastructures implemented on different platforms. Each agent requires a custom adapter to convert a platform-independent notification into a platform-specific message and send it to the system under test (and to receive as well as decode it back). We have implemented adapters for drivers and sinks to interact with ActiveMQ and REDS based on the interfaces provided by FINCoS to custom adapters. We implemented a separate adapter for ActiveMQ as the provided adapter for the JMS caused issues with dynamic topics on ActiveMQ.

We have customized and extended FINCoS to handle expectations, capabilities as well as the feedback provided by a MOM supporting expectations and ASIA. Subscribers can now associate one or more expectations with a subscription and can change that behavior over the duration of an experiment with or without changing their subscriptions. Publishers can associate capabilities with their advertisements, change them at runtime, and – first and foremost – they can now adapt their publication behavior at runtime based on individual feedback given by the MOM.

The behavior of each publisher or subscriber can be individually defined in FINCoS in a platform-independent way. The behavior of publishers is centered around the notion of a phase. In

FINCoS, a phase describes a duration over which a publisher publishes one or more types of notifications at a given sampling rate. The sampling rate can be constant, increasing or decreasing during a phase. The content of each notification can be fully customized in terms of attributes and values with randomized or predefined values for each attribute. Alternatively, each publisher can replay datasets that contain recorded traces of notifications. Per publisher, multiple phases can be defined with individual durations, which allow the configuration of complex dynamics and workload patterns.

The behavior of subscribers is rather static in comparison: subscribers can define content-based subscriptions but cannot change them at runtime, as there is currently only a single phase configurable for subscribers. Thus, we have implemented phases for subscribers as this enables us to simulate requirements that change at runtime. Subscribers can now change their subscriptions, their aggregation requests for aggregated feedback or their expectations during the course of an experiment: as for publishers, multiple phases can be defined for each subscriber. Subscriptions and expectations are registered, revoked or updated at the start of each phase.

Each instance of a subscriber or publisher maintains its own local repository of expectations (or capabilities, respectively). In addition, we have also implemented a support for aggregated feedback about the dynamics and population of the EBS at runtime. Both subscribers and publishers can request updates about aggregated metrics such as *publisherCount*, *subscriberCount*, or *publicationRate*. For each request, an individual imprecision can be set. Aggregation request are registered, updated or revoked at the start of each phase by both subscribers and publishers.

5.4 Summary

In this chapter, we have focused on the implementation of our approach to support QoI requirements in EBS using the proposed concept of expectations, capabilities and feedback.

We have described the generic architecture of our extensions to MOM and participants in an EBS or a DEBS, including the libraries and tools provided to subscribers and publishers. Most parts of our reference architecture are independent of the platform or the application using it. We have presented `ExpectationController` and `ASIAController` to extend the MOM and enable it to provide aggregated feedback and handle QoI requirements.

We have presented prototypes of `ExpectationController` and `ASIAController` implemented on the open-source MOMs ActiveMQ and REDS. These two platforms are examples of centralized and distributed MOMs to highlight different features. While our prototypes for ActiveMQ are realized as separate plugins and can be combined with other off-the-shelf plugins, we have realized our prototype for REDS as an extension to the broker class used by REDS. Modifications to existing code of the MOM have not been necessary in both cases.

Finally, we have shown how publishers and subscribers have to be modified to enable runtime support for QoI requirements using expectations and capabilities. Using the client handlers that are part of our reference architecture, the modifications necessary to manage the lifecycle of expectations and capabilities are negligible compared to the code necessary to manage the lifecycle of advertisements, subscriptions and connections. We have discussed the modifications of four different applications: jms2009-PS, FINCoS, as well as the Ganglia and MySQL replication deployment of the reactive McCAT middleware.

6 Evaluation

In this chapter, we evaluate the concept of expectations, capabilities and feedback to support Quality of Information (QoI) in Event-based Systems (EBSs) at runtime. We first focus on benefits of our approach in terms of resource savings for participants, improved QoI, and superior expressiveness of our model. Second, we gauge the execution costs for the Message-oriented Middleware (MOM) when negotiating expectations and capabilities while scaling the population, workload and dynamics of an EBS. Third, we quantitatively evaluate the scalability of our monitoring approach in terms of QoI and performance.

As shown in Figure 6.1, each part of this chapter focuses on a different aspect of our approach using appropriate Key Performance Indicators (KPIs) and tools to evaluate each aspect. We use CPU utilization, network traffic, memory usage, end-to-end latency, and throughput as KPIs to quantify the benefits as well as the execution costs of our approach. We use fidelity as introduced in Section 3.3.3 to quantify the level of conformance between the QoI requirements of subscribers and the data provided by the EBS. We have extended the industry-strength benchmark *jms2009-PS* [368] and the open-source benchmarking tool *FINCoS* [305] to gauge our prototypes implemented in Java and deployed on a distributed setup using off-the-shelf Cloud technology.

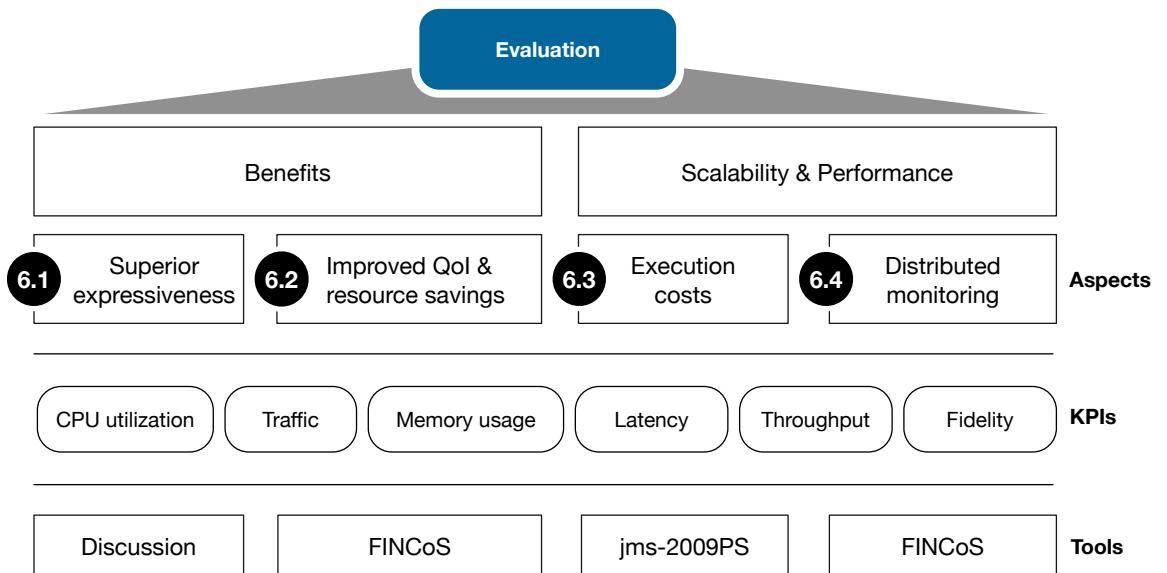


Figure 6.1.: Quantitative evaluation focuses on benefits and execution costs.

We chose FINCoS and jms2009-PS to maximize the comparability and repeatability of our experiments. The benchmark jms2009-PS uses a well-established workload and scaling strategy. This enables us to reliably gauge the overhead for the MOM introduced by our prototype compared to a typical EBS. With jms2009-PS, however, we cannot evaluate the reactive features and benefits of our approach as the benchmark penalizes runtime adaptation, such as adjusting participants or filtering notifications. Thus, FINCoS is used to complementary evaluate the effects of adaptation and feedback on the resource utilization and satisfaction of participants. FINCoS allows for defining complex workloads and individual behavior of participants in a platform-independent

way. With FINCoS, we can emulate scenarios that are suited to compare the features of our approach with the features used by related approaches.

In Section 6.1, we show that our approach is more expressive than related approaches that support QoI explicitly and implicitly. We present related approaches such as Adamant, IndiQoS, Harmony, INCOME, or encoded types and discuss their limitations.

In Section 6.2, we quantify the benefits of our approach in terms of increased QoI and resource savings for subscribers, MOM, and publishers. Using an experimental setup motivated by the Internet of Things (IoT), we compare the results obtained with an EBS applying our approach with those obtained in a typical EBS without any support for QoI, a typical EBS using encoded types and finally, a QoI-aware EBS that does not support self-adaptation. We show that our approach outperforms those approaches in terms of generated fidelity as well as reduced network traffic and CPU utilization.

In Section 6.3, we use jms2009-PS to gauge the execution costs arising from negotiating expectations and capabilities as well as handling feedback to participants in the MOM. We measure overhead regarding CPU utilization and memory usage of the MOM as well as the impact on end-to-end latency and throughput of the processed workload. We show that our approach scales with the workload and is able to provide support for QoI requirements even for the maximum population or throughput that is still manageable for a bare MOM.

In Section 6.4, we evaluate the scalability of our approach for monitoring the population and dynamics of a Distributed Event-based System (DEBS) at runtime. We show that our monitoring approach provides this information more efficiently than an aggregation system deployed as a separate application on top of a DEBS as our approach does not significantly impact the performance of the underlying MOM in terms of throughput and end-to-end latency. We show that our approach exploits relaxations of individual precision requirements defined by participants to significantly reduce the number of updates exchanged in the system while conforming to individual precision requirements at any time, regardless of the number of brokers making up the MOM.

6.1 Expressivity of Expectations and Capabilities

In this section, we discuss the expressiveness of our approach to model and enforce requirements about QoI properties. We show that our concept is superior to related approaches in its expressiveness and the supported set of properties. While related approaches support different fixed sets of properties, our approach supports not only the superset of these properties but also interdependent properties not supported by any other approach; furthermore, our concept can be easily extended to support arbitrary properties. Regarding the complexity of requirements, we show that our concept does not only provide the same expressiveness than related approaches but that it allows for more complex preferences to be defined and enforced. We discuss the limitations of related approaches before we compare them against our concept.

6.1.1 Related Approaches and their Expressiveness

IndiQoS, Adamant/DDS, and Harmony are approaches closely related to our work as they are general purpose MOMs based on the Publish/Subscribe (PS) paradigm with explicit support for quality-related properties; INCOME/QoCIM is an example for a domain-specific approach in the area of Ambient Intelligence (AmI) and Human Computer Interaction (HCI). Encoded-Types can be used alternatively in any EBS that does not offer explicit support for quality-related

properties. We review these related approaches, address their general expressivity regarding requirements and capabilities and discuss their limitations. We summarize our findings regarding the supported set of properties in Table 6.1 and the general expressivity in Table 6.2.

IndiQoS

IndiQoS [23, 22, 85] allows subscribers to express requirements about quality-related properties of notifications they subscribe to. A *QoS specification* is part of the subscription and contains a list of conditions that describe target-values for each quality-related property. These target-values are minimum or maximum values depending on the pre-defined semantics of each property. All conditions of a QoS specification have to be satisfied by the system.

IndiQoS supports only requirements about the maximum latency as well as the minimum periodicity and precision of notifications. The periodicity property *period* defined in IndiQoS can be mapped to the *sampling rate* property used in our model, as it defines the number of notifications published by one publisher during a given time interval [23]. Latency is determined, manipulated, and matched against the requirements defined in a subscriber's QoS specification by the MOM at runtime. Publishers expose their current support for period/sampling rate as a *QoS profile* that is part of an advertisement. Listing 6.1 shows an example containing an advertisement with a QoS profile and a subscription with a QoS specification in IndiQoS; the code examples are taken from [23] and [22].

Please note that in IndiQoS, *precision* is not modeled as a quality-related property explicitly enforced by IndiQoS but is rather considered to be part of the content of a notification [22]. Thus, the IndiQoS MOM can filter out notifications that contain a matching attribute *precision* but IndiQoS is not able to enforce requirements about precision by adjusting the currently provided precision either by adapting publishers or adjusting the precision at the MOM. Furthermore, the publicly available documentation of IndiQoS does not disclose whether requirements about period/sampling rate are enforced in the MOM.

Listing 6.1: Examples for expressing requirements and support in IndiQoS [23].

```

1 // Advertisement of a publisher with additional QoS profile
2 Publisher p = new Publisher of Temp
3         withProfile(
4             room  = "lab1",
5             temperature = any,
6             precision = 0.01)
7         withQoSProfile
8             Periodic (period=10)
9
10
11 // Subscription with additional QoS requirements
12 Subscription s = subscribe Temp
13         where (temperature > 60)
14         withQoS (
15             (Periodic(period < 1))
16             and (latency < 10 )
17         )

```

Adamant/DDS

Adamant [212, 213, 211, 210] is based on the Data Distribution Service (DDS) [334] and extends DDS to be self-adaptive based on requirements about low-level Quality of Service (QoS) properties of the MOM such as loss, jitter, etc. In Adamant, like in DDS, DataWriters are associated with publishers and DataReaders are associated with subscribers. DataWriters and DataReaders describe their current state or their requirements regarding quality-related properties in *QoS policies* that are assigned to a given type of notification. Each QoS policy contains a list of independent constraints on predefined properties and the current value of each property that it is provided with by a DataWriter. Latency and *deadline* are exposed by Adamant as properties relevant to QoI. The deadline property can be mapped to the *sampling rate* property used in our model as it describes the rate at which periodic data should be refreshed by the DataWriter [216].

Listing 6.2 shows examples for QoS policies expressed by a DataWriter (top) and a DataReader (bottom) using the Distributed QoS Modeling Language (DQML) introduced in [214, 215, 216]. Please note that the equality sign in these policies implies a lower bound, i.e., *period* = 100 implies that notifications should be sent *at least* every 100 milliseconds. This corresponds to a sampling rate of *at least* 10 notifications per second [214].

Listing 6.2: Examples for QoS policies in Adamant DDS expressed using DQML as given in [216].

```
1 <DQML>
2   <DataWriter name="DataWriter1">
3     <deadline>period="50" />
4   </DataWriter>
5 </DQML>
6
7 <DQML>
8   <DataReader name="DataReader1">
9     <deadline>period="100" />
10  </DataReader>
11 </DQML>
```

Harmony

Harmony [135, 253, 428] focuses on providing support for end-to-end latency. Subscribers can specify a maximum latency for a type of notification. Harmony monitors and adapts a network of brokers at runtime to enforce these requirements. At the time of writing, no further details on syntax or semantics of these requirements were publicly available. We refer the interested reader to [138] for a more detailed discussion of the general features of Harmony.

INCOME/QoCIM

INCOME [296] uses QoCIM [295] and targets applications in the domain of Quality of Context (QoC). QoCIM provides a domain-specific meta model for describing properties of contextual data that is similar to our notion of generic properties. Subscribers can add XPath queries over these properties to their subscriptions about types of context data. The PS middleware INCOME briefly discussed in [296] filters out context data provided by publishers (called producers) that does not match these extended subscriptions issued by subscribers (called consumers). Broker-side and interdependent properties such as latency or alternatives cannot be supported due to

limitations of the model. Listing 6.3 shows an example as given in [296] how to express the constraint that the precision of the pollution detection sensor #45 has to be at least 40.

In contrast to other approaches like IndiQoS or Adamant, INCOME does not apply any kind of adaptation to enforce requirements – unsuitable data is merely discarded. While QoCIM would allow for modeling other properties such latency, requirements about these properties cannot be enforced in INCOME as this would require a notion of broker-side or interdependent properties and ways to enforce them in the MOM or at publishers.

Subscribers cannot define trade-offs between requirements about different properties or rank multiple sets of requirements. As in [248, 346], subscribers are not supposed to define more than a single threshold per property.

Publishers cannot specify their costs for providing properties at a given value and cannot provide additional information about their adaptation spectrum other than the current values they provide for a given property.

Listing 6.3: Example given in [296] for a constraint on the precision of a pollution detection sensor to be > 40. Constraints are expressed using XPath as part of subscriptions in INCOME.

```

1 //QoC property
2 <qocindicator id="10" name ="PrecisionQoCIndicator">
3 ...
4 </qocindicator>
5
6
7 // Context-based constraint
8 if(xpath.evaluate("//observable[uri='#pollution' and
9           entity[uri='bordeaux://thiers ave./sensors/45/']]",
10          doc.XPathConstants.NODESET).length == 0)
11    {return false;}
12
13 // QoC-criterion constraint
14 if(xpath.evaluate("//qocindicator[@id='10' and
15           qoccriterion [@id='[10.1]']/qocmetricdefinition[@id='10.1']]",
16          doc.XPathConstants.NODESET).length == 0)
17    {return false;}
18
19 // QoC-value constraint
20 if(xpath.evaluate("//qocindicator[@id='10' and
21           qoccriterion[@id='[10.1]']/qocmetricdefinition[@id='10.1']]
22           and qocmetricvalue[@value>='40']]",
23          doc.XPathConstants.NODESET).length == 0)
24    {return false;}
```

EncodedTypes

EncodedTypes provide implicit support for QoI [7, 85]. They can be used in any EBS as they encode the quality-related properties of notifications into the name of an event type. All notifications published on such an encoded topic adhere to these properties and current values. For example, notifications published on the encoded topic `CpuUsage_accuracy90_precision95` are notifications about `CpuUsage` with 90% accuracy and 95% precision. Publishers announce their current support for properties by advertising encoded types that reflect their current state while subscribers announce their current requirements by subscribing to encoded types that map

to their current requirements. All encoded types expressed in subscriptions and advertisements are registered at the MOM and can be looked up by subscribers and publishers to determine which types are available. Publishers then publish on all encoded topics that cover their current capabilities while subscribers subscribe to all encoded types covering their requirements.

This approach, however, is limited in expressiveness and efficiency, as it does not support broker-side and interdependent properties. This is due to a lack of feedback between participants.

Regarding expressiveness, the set of supported properties is limited to Quality of Device (QoD) properties and those QoI properties determined by publishers such as drift, accuracy, or precision. Important broker-side and interdependent properties such as latency or alternatives cannot be expressed or enforced. Subscribers cannot encode their requirements about a minimum or maximum number of alternative publishers they want to receive data from. Similarly, neither lower nor upper bounds for received rates can be expressed or enforced. Both properties would require a coordination at least between different publishers that want to publish on the same encoded topic to avoid over- or underprovisioning. Such a coordination, however, is not possible in a typical EBS where participants do not know about the number and status of other participants [171, 172, 177].

Assuming a typical EBS where the MOM is routing notifications solely based on their types and content matching the filters defined by subscribers, subscribers have no way of indicating trade-offs or ranges of accepted values to publishers while publishers have no way of signaling ranges of values realizable with adaptation. Furthermore, subscribers cannot announce utility while publishers cannot announce costs.

Regarding efficiency, the number of encoded types that have to be maintained in parallel quickly becomes unmanageable due to a lack of feedback between participants [85]. Subscribers and publishers do not know which encoded topics are actively used at the moment, i.e., which encoded topics do have active subscribers and publishers. Thus, subscribers and publishers have to maintain all encoded types that match their requirements and capabilities. Without feedback, subscribers would have to perform repeated lookups to see if publishers have registered new encoded types, check whether these new types map to their requirements and subscribe to them if necessary. Publishers have to publish the same notification on all encoded topics that match their capabilities, as they do not know which types are subscribed to. For example, a newly connected subscriber will choose all the topics that match its own requirements as it does not know which types are actively published on.

Even when assuming a scenario where each publisher does not use its own values but the range of current values is segmented by equidistant intervals and inclusive semantics (e.g., CpuUsage_accuracy90_precision95 denotes all notifications with *at least* 90% accuracy and *at least* 95% precision), the number of encoded topics would still be huge as the set size of encoded types $|ET|$ is given in Equation (6.1).

$$|ET| = \text{interval size}^{\text{number of properties}} \quad (6.1)$$

For example, splitting the full range of values for each property into equidistant intervals with length 10 (e.g., precision10, precision20, precision30) per property and a set of 3 properties, this would result in $|ET| = 10^3 = 1000$ encoded topics that would have to be maintained in a worst-case scenario. The set of encoded types a publisher is publishing on at runtime, however, depends

on the current capabilities of publishers as we have discussed in Section 3.4. For example, a publisher has been publishing notifications with at least 90% accuracy on `CpuUsage_accuracy90` but at one point in time had been forced to reduce its accuracy to 80% to save energy. Consequently, the same publisher stops publishing on `CpuUsage_accuracy90` but starts to publish on `CpuUsage_accuracy80`. Once switching back to an accuracy of 90% it has to continue to publish on `CpuUsage_accuracy80` in addition to `CpuUsage_accuracy90`.

6.1.2 Summarizing the Limitations of Related Approaches

Summing up the results of our analysis with regard to expressivity in Tables 6.1 and 6.2 shows the limitations of related approaches compared to our work.

Table 6.1.: Quality-related properties shown in Figure 3.6 that are explicitly supported by related approaches discussed in Section 6.1.1.

Approach	Accuracy	Alternatives	Latency	Precision	Sampling Rate
IndiQoS	□	□	■	■ ¹	■ ²
Adamant/DDS	□	□	■	□	■ ²
Harmony	□	□	■	□	□
INCOME/QoCIM	□	□	□	■	■ ³
EncodedTypes	■	□	□	■	■
Expectations	■	■	■	■	■

¹ Precision is not modeled as a quality-related property as part of the QoS specification / profile in [23] but as content of the notification. Thus, it cannot be actively enforced.

² The property *period* corresponds to the generic property *sampling rate*.

³ The property *temporalResolution* corresponds to the generic property *sampling rate*.

IndiQoS, Adamant, DDS, Harmony, INCOME, and EncodedTypes all allow subscribers to specify requirements about quality-related properties. The set of quality-related properties explicitly supported by each approach, however, is very limited as shown in Table 6.1.

In all approaches, subscribers are allowed to model requirements only over a fixed set of properties that the MOM tries to enforce at runtime by either filtering out unsuitable notifications (INCOME, IndiQoS for properties other than latency) or actively adapting itself (IndiQoS, Adamant, DDS, Harmony). None of these approaches, however, actively enforces requirements about

Table 6.2.: Features necessary for expressing QoL requirements and their support in related approaches discussed in Section 6.1.1.

Approach	Alternative Sets ¹	Utility ²	Trade-offs	Adaptation Spectrum ³	Costs
IndiQoS	□	□	■	□	■ ⁴
Adamant/DDS	□	□	□	□	□
Harmony	□	□	□	□	■ ⁴
INCOME/QoCIM	■ ⁵	□	■ ⁵	□	□
EncodedTypes	□	□	□	□	□
Expectations	■	■	■	■	■

¹ For the same subscription, different sets of interdependent requirements are supported.

² Subscribers can rank and weight sets of requirements to reflect different utility values.

³ Publishers can expose their spectrum of adaptation.

⁴ Costs are considered by the MOM during adaptation but cannot be influenced by publishers.

⁵ Can be expressed using XPath but it is unclear how enforcing them is implemented by the approach.

atomic properties determined and manipulated only by publishers — such as accuracy — or about complex and interdependent properties — such as alternatives.

Furthermore, subscribers are restricted in the degree to which they can express their preferences. As we have discussed in Section 3.3, preferences of a subscriber can map to requirements about different sets of interdependent QoI properties; these sets can differ in their composition and the value they generate for the subscriber if satisfied. As shown in Table 6.2, subscribers cannot express such complex preferences in the discussed approaches as they cannot express the importance of their requirements to the MOM, cannot define independent sets of interdependent requirements and the trade-offs between the requirements of one set.

Conversely, the EBS is limited in supporting requirements about properties as publishers are not able to express their spectrum of values they could realize by (self-)adaptation or their costs for providing those properties.

6.1.3 Expressiveness of Expectations and Capabilities

The concept of expectations, capabilities, and feedback proposed in this dissertation offers a higher degree of expressiveness than all the related approaches discussed above: it does not only incorporate them completely but also advances the scope and degree of complexity that QoI requirements can be expressed and supported with in EBS.

The concept of expectations and capabilities can be used to model requirements in the same way as in related approaches. Modeling the properties supported by related approaches as generic properties, we can define requirements about them using expectations. A minimum target value for a generic property p_k without an explicit upper bound as shown in Listing 6.1 or Listing 6.2 can be modeled by setting the lower bound $p_k.\text{LB}$ to the desired target value while the upper bound $p_k.\text{UB}$ is set to the maximum value defined for this generic property. Conversely, maximum target values without an explicit lower bound can be modeled by setting the lower bound $p_k.\text{LB}$ to the minimum value defined for this generic property p_k while the upper bound is set to the desired target value. An example for an expectation containing both types of requirements is \mathbf{X}_3^e we have used to illustrate fidelity in Section 3.3.3 and shown in Figure 3.16a: for latency, we would accept all values lower than the defined upper bound of 250 milliseconds; for accuracy and precision in turn, we have defined minimum values but would also accept all other values up to the defined maxima. As IndiQoS, Adamant, and INCOME do not allow publishers to describe their spectrum of adaptation, we can imitate these restrictions for capabilities by setting $C_k^e.\text{LB} = C_k^e.\text{CV} = C_k^e.\text{UB}$ for a capability C_k^e defined for p_k .

The concept of expectations and capabilities allows for expressing more complex preferences. Defining several expectations with each expectation being defined only about a single generic property, we can model independent requirements. Defining an expectation about a set of generic properties in turn models a set of interdependent requirements where all requirements have to be satisfied simultaneously by the MOM as discussed in Section 3.3.1.

We can enforce requirements about more properties than all other approaches as we include publishers when deciding on adaptation; we can specify the current and potential support of a publisher for a given generic property and use this knowledge at the MOM at runtime to decide on suitable adaptations. In the same way, we can model and enforce requirements about arbitrary generic properties that only publishers can determine and manipulate — such as accuracy — as well as for complex and interdependent properties — such as alternatives. We demonstrate this in the next sections.

6.2 Benefits Regarding Data Quality and Resource Savings

In this section, we focus on the benefits of our approach in terms of higher QoI for subscribers as well as resource savings for subscribers, the MOM and even for publishers. We show in particular that it is necessary to adapt publishers at runtime to support QoI requirements of subscribers — merely discarding notifications or adapting the MOM is insufficient.

6.2.1 Heterogeneity Scenario: Dealing with Unsuitable Data

We use the following scenario motivated by the IoT that generalizes examples used in work about collective sensing tasks to show the feasibility and benefits of our approach: we assume a situation where multiple publishers all provide position data about a vehicle by publishing notifications about pos. The content of each notification is a set of double values describing the vehicle's position. Although all publishers provide the same type of data about the same entity, we assume that publishers can be heterogeneous due to their design, configuration or their current context and state at runtime. Especially in ambient intelligence environments with a multitude of sensors, multiple publishers with varying capabilities can publish the same type of information. As we have already discussed in Section 3.4, some publishers can publish with a higher sampling rate than others, have better connectivity (which results in lower latency) or they can offer data with higher accuracy due to better sensors, positioning techniques, or fusion algorithms. In our setup, this heterogeneity leads to different support for the generic properties *accuracy*, *latency*, *precision*, and *sampling rate*.

In our example, a subscriber is interested in the current position of a vehicle and can subscribe to notifications about pos with or without registering additional expectations about quality-related properties. The subscriber is resource-constrained and thus interested in limiting the maximum number of notifications it has to process per second. At the same time, it requires at least 10 publishers to provide at least seven notifications per second each. Inaccurate or outdated data as well as information that is not confirmed by enough data sources would lead to wrong conclusions about the position and speed of the vehicle.

For our evaluation, we assume that a subscriber has an expectation \mathbf{X}_5^{pos} about the quality-related properties *accuracy*, *alternatives*, *latency*, *precision*, and *sampling rate* as shown in Figure 6.3a and Table 6.3.

Table 6.3.: Expectation \mathbf{X}_5^{pos} and its reduced form \mathbf{X}_6^{pos} used in the next sections.

Expectation	Accuracy		Alternatives		Latency		Precision		Sampling Rate		Utility
	LB	UB	LB	UB	LB	UB	LB	UB	LB	UB	
\mathbf{X}_5^{pos}	70	100	10	20	0	250	85	100	7	20	25
\mathbf{X}_6^{pos}	70	100	—	—	0	250	85	100	7	20	25

Note: \mathbf{X}_i^e denotes expectation #i as a set of QoI requirements about notifications of type e .

Limiting the maximum number of notifications to be received by subscribers is realized by defining lower and upper bounds for both alternatives and sampling rate (cf., Equation (6.2)).

$$\text{receivedRate} = \left[p_{alt} \cdot \text{LB} \times p_{sampRate} \cdot lb \quad ; \quad p_{alt} \cdot \text{UB} \times p_{sampRate} \cdot ub \right] \quad (6.2)$$

We group available publishers into two categories: suitable and unsuitable publishers. Suitable publishers provide notifications with current values for precision, accuracy and latency that conform to the requirements of the subscriber; notifications are published with a sampling rate that also conforms to the requirements of the subscriber. Unsuitable but capable publishers, on the other hand, provide notifications that do not conform with the requirements of subscribers. Furthermore, these notifications are provided with a higher sampling rate than the maximum rate requested by the subscriber. For the sake of simplicity, suitable publishers register CP_4^{pos} with randomly varying current values that all satisfy the respective requirements by subscribers while unsuitable publishers register variations of CP_5^{pos} (cf., Table 6.4).

Table 6.4.: Capability profiles used by suitable and unsuitable publishers.

Capability Profile	Accuracy			Latency			Precision			Sampling Rate		
	LB	CV	UB	LB	CV	UB	LB	CV	UB	LB	CV	UB
CP_4^{pos} (suit.)	70	100	100	100	100	240	50	95	100	0	10	60
CP_5^{pos} (unsuit.)	50	60	80	200	400	400	40	60	95	0	40	60

Note: The generic property *alternatives* is not modeled as it cannot be satisfied a single publisher but only by a set of publishers.

Please note that the generic property *alternatives* cannot be part of the capability profile of a single publisher as it can be satisfied only by a set of publishers. Thus, we use X_6^{pos} as a reduced version of X_5^{pos} to make the visual comparison with each capability profile easier. The star plot of X_6^{pos} in Figure 6.3b shows X_5^{pos} without the requirements about *alternatives*.

As we can see from overlaying the star plots for each capability profile with X_6^{pos} in Figures 6.3e and 6.3f, publishers providing CP_4^{pos} do already satisfy X_6^{pos} while publishers characterized by CP_5^{pos} do not. However, the potential support described by CP_5^{pos} shows that X_6^{pos} could be satisfied by adapting the current value for each generic property. Thus, all publishers that are characterized by CP_4^{pos} are treated as suitable publishers when evaluating the requirements for *alternatives* while publishers characterized by CP_5^{pos} would have to adapt first.

We run several test runs with the same number of active publishers and subscribers. With each single run we increase the percentage of publishers that provide the correct type of information but with insufficient quality-related properties as illustrated in Figure 6.2. We denote this increasing percentage of publishers providing unsuitable data as the *degree of heterogeneity*.

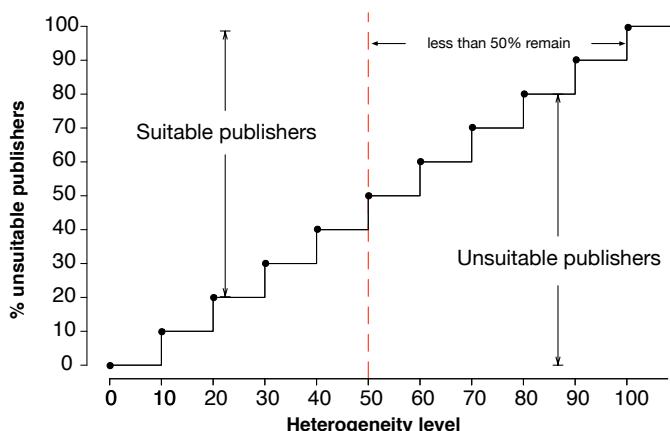


Figure 6.2.: Heterogeneity: increasing number of publishers providing data with insufficient QoI.

The starting point for our setup is a heterogeneity of 0%: all publishers offer suitable data. For this Best-Case workload (BC), no notifications from surplus publishers have to be processed or discarded. The results measured for this workload represent an upper bound for QoI of the notifications received by subscribers; at the same time they represent a lower bound for the resource utilization of publishers, subscribers and brokers as neither publishers nor subscribers or brokers have to process notifications that are not in conformance with X_5^{pos} .

We gradually increase the percentage of insufficient publishers until all available publishers provide the correct type of data but with insufficient quality-related properties at a heterogeneity level of 100%. The workload generated by publishers scales with the level of heterogeneity as unsuitable publishers publish more notifications than suitable publishers.

For each single run we compute the following KPIs:

- accuracy, latency, and precision as defined in Section 3.3.3;
- sampling rate as the number of notifications received by the subscriber per second from a single publisher;
- alternatives as the number of publishers that data has been received from by the subscriber at every point in time; and
- received rate as the total number of notifications received by the subscriber every second.

We measure and analyze the results received by subscribers when using a typical EBS to process the workload generated at each level of heterogeneity. For better comprehensibility, the graphs in Figure 6.4 visualizing the data received by subscribers contain red dashed lines that denote the upper and lower bound defined for each property. The percentage of received notifications that conform to the requirements is visualized in Figure 6.5.

Alternatives. As shown by the solid black line in Figure 6.4a, subscribers keep receiving notifications from all publishers, regardless of the fact that the overall number of publishers providing suitable data is decreasing with increasing heterogeneity (indicated by the dotted black line in Figure 6.4a). This behavior is to be expected as a typical EBS is routing notifications solely based on their content/type; both suitable and unsuitable publishers, however, provide notifications with matching content/type.

Total number of notifications and received rates. The total number of notifications processed as well as the rates that these notifications are received with by subscribers is increasing as shown in Figures 6.4b and 6.4c. This is due to unsuitable publishers publishing at a higher rate than suitable publishers.

Latency, accuracy and precision. Unsuitable publishers provide notifications with a higher latency and higher inaccuracy. With an increasing number of unsuitable publishers, this results in an overall deterioration of latency, accuracy and precision as shown in Figures 6.4d to 6.4f.

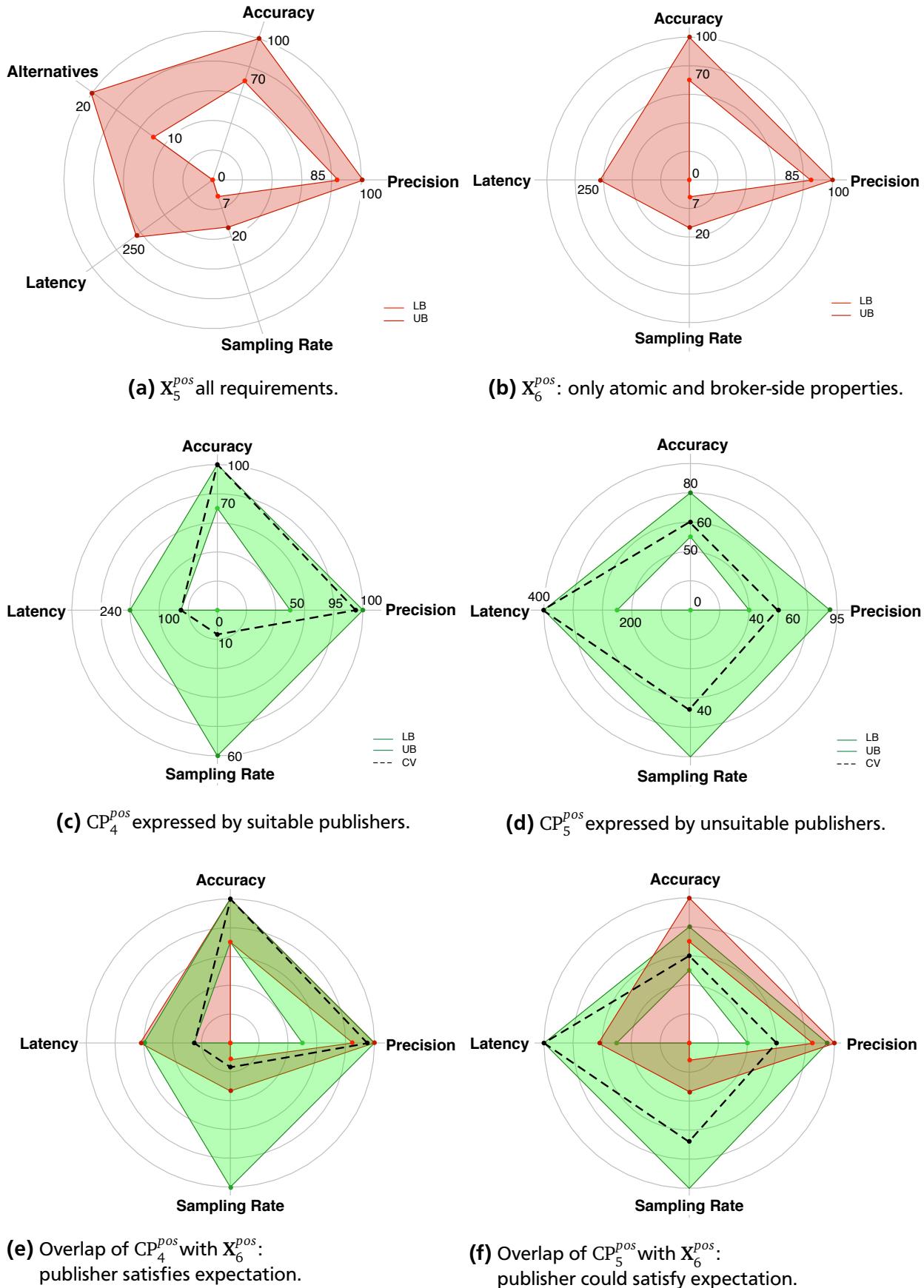
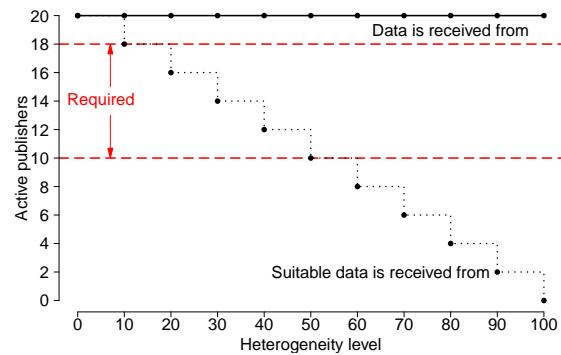
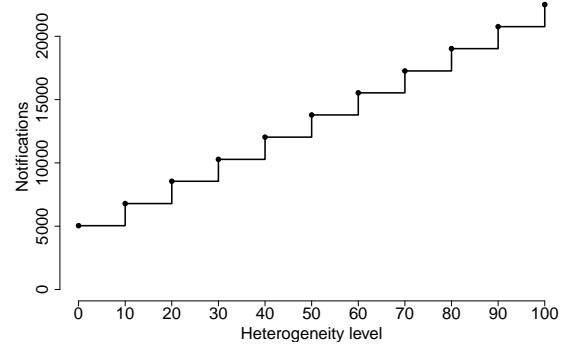


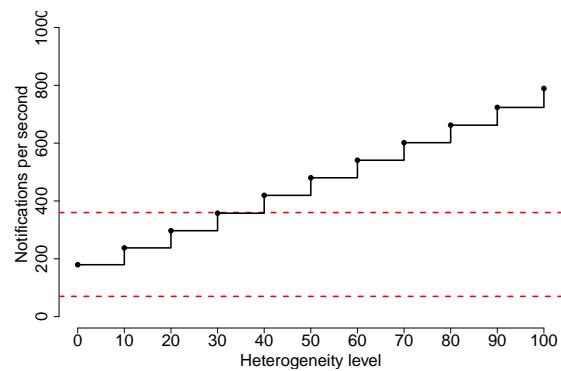
Figure 6.3.: Star plots showing CP_4^{pos} and CP_5^{pos} as well as their ability to support requirements about atomic and broker-side properties defined in expectation X_5^{pos} .



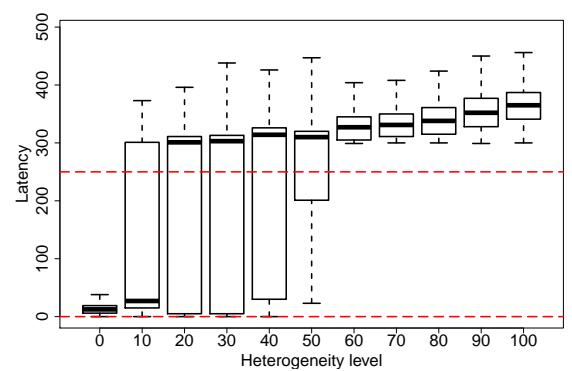
(a) Decreasing number of suitable publishers.



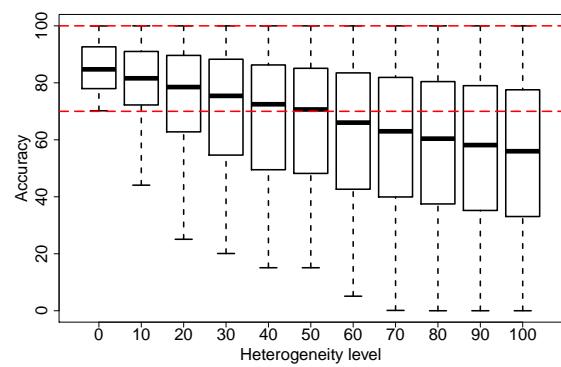
(b) Increasing total number of notifications.



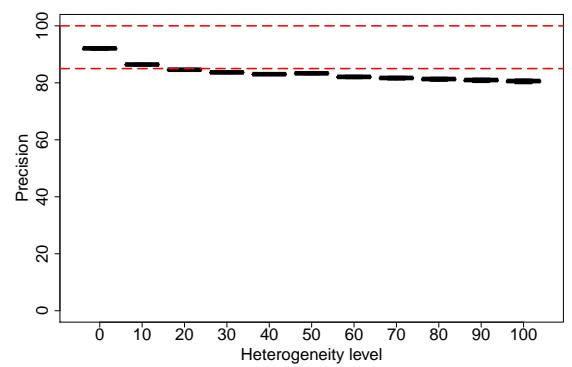
(c) Increasing rates received (notifications/sec).



(d) Increasing latency (msec).



(e) Decreasing accuracy (%).



(f) Decreasing precision (%).

Figure 6.4.: Typical EBS: measured results obtained when increasing heterogeneity show that properties of received notifications are affected by increasing heterogeneity of publishers in a system without support for QoI. The horizontal dashed red lines denote the lower and upper bounds defined for each requirement in X_5^{pos} .

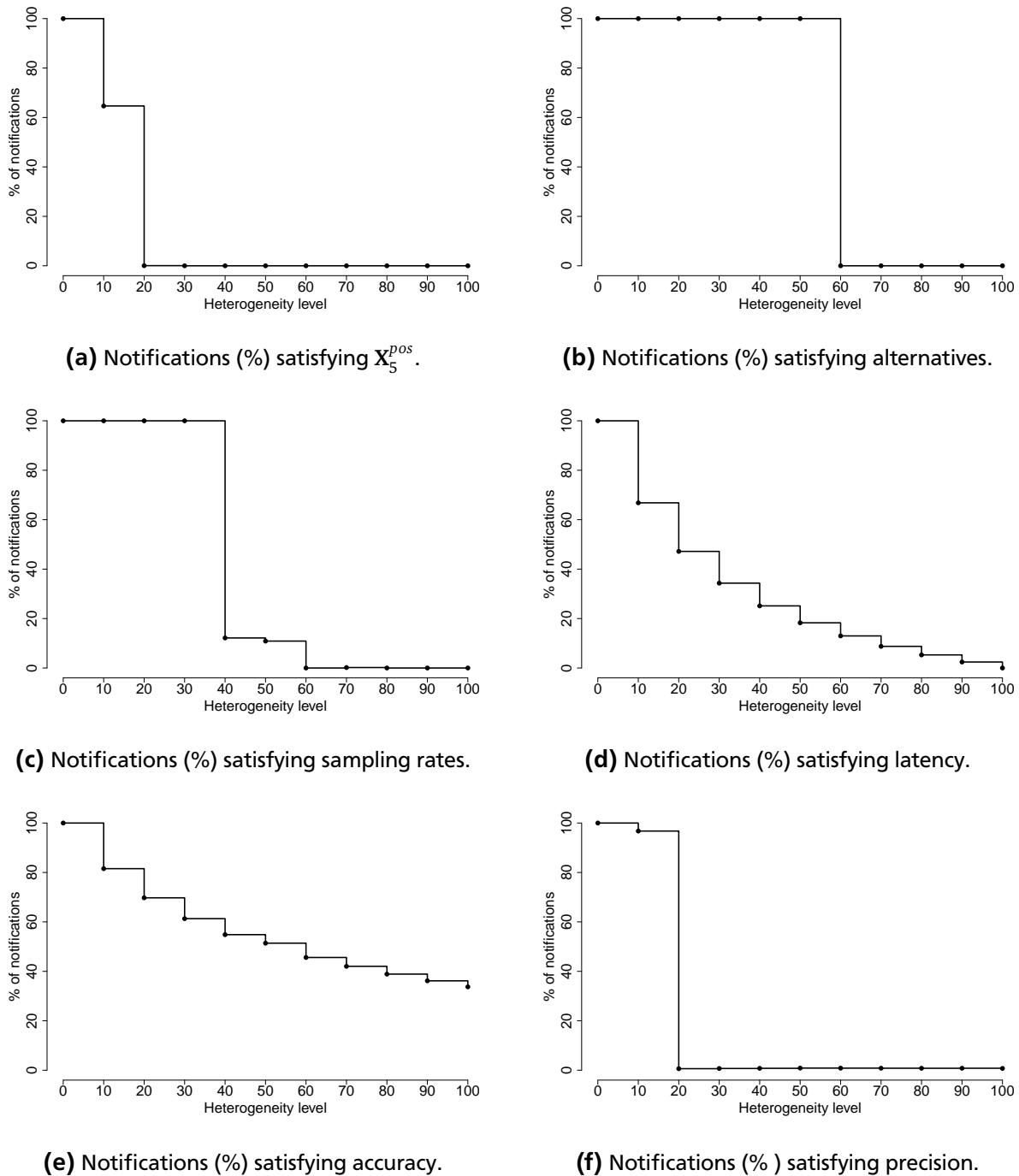


Figure 6.5.: Consequences using a typical EBS: decreasing conformance with requirements. Deteriorating data quality due to increasing heterogeneity in an EBS without support for QoL results in less notifications satisfying the requirements of a subscriber. The requirements are indicated by red dashed lines in Figure 6.4 for each property.

6.2.2 Comparing Expectations with Features of Related Approaches

Using a typical EBS in a scenario with increasing levels of heterogeneity results in data that is less and less conforming to the requirements of subscribers. This is due to the fact that a typical EBS is not aware of subscribers' requirements and publishers' capabilities regarding QoI as we have seen in the previous section.

In Section 6.1.1, we have discussed several features used by related approaches to support QoI in an EBS. We examine how well an EBS can cope with increasing levels of heterogeneity when using these different features in isolation. Consequently, we measure and compare the results received by subscribers for increasing levels of heterogeneity when using an EBS that either uses just filtering, applies encoded types, or uses expectations, capabilities and feedback.

We use four different configurations for ActiveMQ to emulate an EBS that relies only on specific features to support QoI: Without Expectations (WOE), Expectations + Self-Adaptation (ESA), Expectations, Filtering Only (EFO), and Encoded Types (ET). The workload generated at each level of heterogeneity is the same for each configuration.

Without Expectations (WOE). The EBS is unaware of QoI. Notifications are processed based on their conformance with the subscription about the type or content. This configuration resembles a typical EBS and provides subscribers with the biased data presented in the previous section. We use WOE as a baseline for quantifying the benefits or drawbacks of the different configurations in terms of QoI and resource utilization.

Expectations + Self-Adaptation (ESA), our approach. The EBS is aware of QoI by using expectations, capabilities and feedback to reactively enforce requirements about QoI. In addition to filtering out unsuitable data, publishers or brokers are advised to adapt if the currently provided notifications do not conform to the requirements defined in X_5^{pos} .

Expectations, Filtering Only (EFO). The EBS is aware of requirements about QoI by using expectations and capabilities. However, it does not use feedback to advise publishers to adapt in order to reactively enforce requirements; nor does it apply self-adaptation at the MOM. In this scenario, the MOM just filters out notifications that do not conform to the requirements defined by subscribers. This behavior mimics the limitations of related approaches such as INCOME/QoCIM or IndiQoS (for properties other than latency).

Encoded Types (ET). We use EncodedTypes to provide support for properties in a typical EBS. This configuration is based on the WOE scenario in as much as that the underlying EBS is unaware of expectations, capabilities, and QoI. Requirements and capabilities about a subset of properties are expressed using encoded types when advertising, publishing or subscribing as described in Section 6.1.1. We assume a typical EBS that does not provide participants with feedback about its population. Consequently, subscribers and publishers do not know if a particular encoded type is still subscribed to or published on.

However, we make several simplifying assumptions to reduce the number of encoded types to be maintained in parallel in this configuration. We assume that publishers encode only their lower and upper bounds as well as their current values into their encoded types while subscribers encode their lower and upper bounds. This way, publishers can announce their spectrum of adaptation to subscribers while subscribers can announce their upper and lower bounds of accepted values. Subscribers subscribe only to encoded types that are as good as or better than their lower bounds. Publishers publish on the encoded types that

match their current values or are covered by their current values, i.e., they do not only publish on the encoded type that is closest to their current values.

Furthermore, we address only precision and accuracy as both properties are controlled by publishers. Using Equation (6.1), this already results in a set of $2 \cdot 2^3 + 2^2 - \text{duplicates} = 19$ unique encoded types that have to be maintained when using the values defined in CP_5^{pos} , CP_4^{pos} , and X_5^{pos} as shown in Table 6.5.

In principle, sampling rate is another property that is controlled by publishers and could be used in encoded types. However, including the sampling rate property into our example would inflate the total number of unique encoded types to be maintained at the MOM from 19 for precision and accuracy to $2 \cdot 3^3 + 2^3 - \text{duplicates} = 44$ encoded types. In addition, encoded types that represent wild cards such as Pos, Pos_Accuracy, or Pos_Precision are omitted to reduce the set of encoded types to maintain.

Consequently, subscribers with X_5^{pos} subscribe to six encoded types that match their requirements as shown in Table 6.5. Please note that suitable publishers have to publish each notification on 17 different encoded types in this scenario. This is due to CP_4^{pos} describing precise and accurate data that covers all other encoded types promoting less precise or accurate notifications. Contrarily, publishers with CP_5^{pos} only have to publish the same notification on four encoded types (i.e., Pos_Precision40_Accuracy50, Pos_Precision40_Accuracy60, Pos_Precision60_Accuracy50, Pos_Precision60_Accuracy60).

Table 6.5.: Encoded types used in the Encoded Types (ET) configuration based on X_5^{pos} , CP_4^{pos} and CP_5^{pos} as described in Tables 6.3 and 6.4.

EncodedType	Defined by			Published on by		X_5^{pos}
	X_5^{pos}	CP_4^{pos}	CP_5^{pos}	CP_4^{pos}	CP_5^{pos}	
Pos_Precision85_Accuracy70	■	□	□	■	□	■
Pos_Precision85_Accuracy100	■	□	□	■	□	■
Pos_Precision100_Accuracy70	■	□	□	□	□	■
Pos_Precision100_Accuracy100	■	□	□	□	□	■
Pos_Precision50_Accuracy70	□	■	□	■	□	□
Pos_Precision50_Accuracy100	□	■	□	■	□	□
Pos_Precision100_Accuracy70	□	■	□	■	□	□
Pos_Precision100_Accuracy100	□	■	□	■	□	□
Pos_Precision95_Accuracy70	□	■	□	■	□	■
Pos_Precision95_Accuracy100	□	■	□	■	□	□
Pos_Precision40_Accuracy50	□	□	■	■	■	□
Pos_Precision40_Accuracy80	□	□	■	■	□	□
Pos_Precision40_Accuracy60	□	□	■	■	■	□
Pos_Precision95_Accuracy50	□	□	■	■	□	□
Pos_Precision95_Accuracy80	□	□	■	■	□	■
Pos_Precision95_Accuracy60	□	□	■	■	□	□
Pos_Precision60_Accuracy50	□	□	■	■	■	□
Pos_Precision60_Accuracy80	□	□	■	■	□	□
Pos_Precision60_Accuracy60	□	□	■	■	■	□

6.2.3 Benefits: Higher Data Quality

In the previous sections, we have shown how heterogeneity impacts data processed in a typical EBS without any support for QoI. In this section, we quantify the benefits of our approach when varying the level of heterogeneity. We compare our concept of expectations, capabilities and feedback against a typical EBS (WOE configuration), related approaches, and a best-case workload. The best-case workload (BC) represents an upper bound for the cumulative fidelity and the resource savings. As defined in Section 6.2.1, the BC workload is achieved in a situation where all publishers offer suitable data and no notifications from surplus publishers have to be processed or discarded. Related approaches are represented by the EFO configuration, where requirements about QoI are enforced by filtering notifications, and by the ET configuration where encoded types are used in a typical EBS. Our approach is represented by the ESA configuration where we adapt the MOM, publishers, or both using feedback.

Figure 6.6 shows the cumulative fidelity that measures the conformance between requirements and processed data for each configuration at each level of heterogeneity. In the initial situation with a heterogeneity of 0%, the fidelity generated in each configuration is similar to the best case, except for ET. The fidelity in ET is negative (i.e., the subscriber is unsatisfied) as a subscriber has to deal with too many notifications due to publishers publishing the same notifications on multiple topics that the subscriber has also subscribed to.

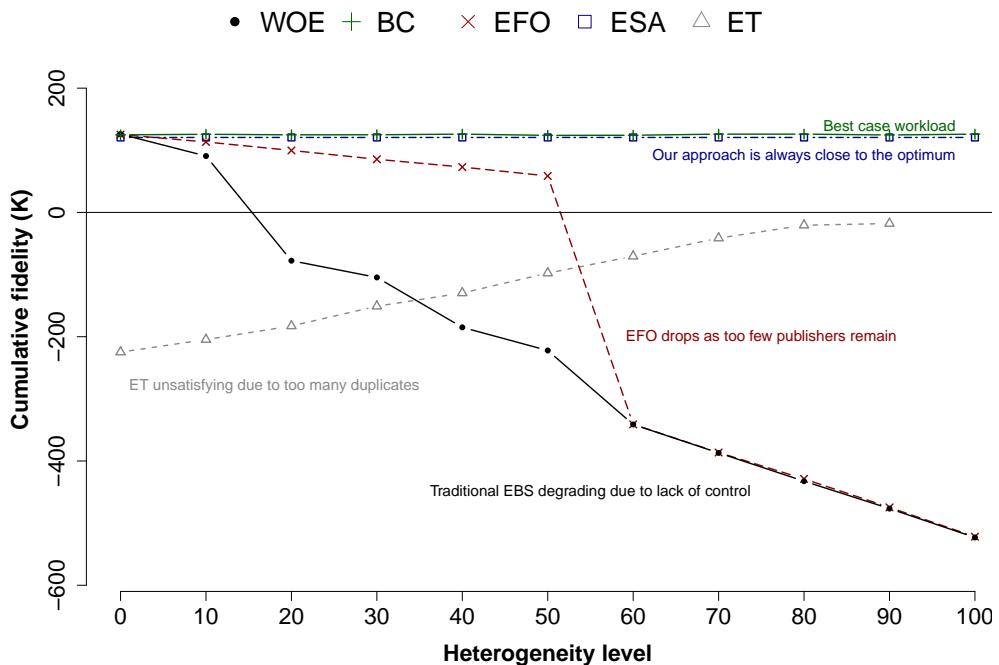


Figure 6.6.: Cumulative fidelity for all configurations in comparison: ESA generates the same fidelity as BC due to adapting publishers and MOM; EFO cannot provide satisfying results for $h > 50$ as it filters but does not adapt; ET does not provide satisfying results at all; baseline WOE represents a typical EBS with decreasing results.

With increasing heterogeneity we can see that the fidelity generated in a typical EBS without any support for QoI (WOE) is monotonically and quickly decreasing as the percentage of unsuitable notifications is increasing but these notifications are not filtered out (cf., Figures 6.4 and 6.5). Discarding these notifications generates costs for the subscriber that are reflected by penalties in

the fidelity metric (cf., Equations (3.3) and (3.6)). In EFO, the fidelity is also decreasing with increasing heterogeneity but at a much slower rate than in WOE. This is due to the fact that the EBS in EFO supports requirements about QoI by filtering out unsuitable notifications. While this releases a subscriber from having to discard unnecessary notifications on its own and avoids penalties, the overall number of notifications that are received by a subscriber is much lower than with the best-case workload BC or in the ESA configuration.

The limitations of the EBS in EFO become apparent for situations with a heterogeneity of more than 50% where not enough suitable publishers are available. As the alternatives requirement cannot be satisfied by the system anymore the expectation is rejected and the system operates like a typical EBS in that it routes notifications solely based on their type and content. This is reflected by a sharp drop in the fidelity for EFO as shown in Figure 6.6: while the subscriber has still been satisfied for a heterogeneity level $h \leq 50$, the data received for higher levels of heterogeneity result in an overall dissatisfaction.

Contrarily, when the EBS uses our approach in the ESA configuration to enforce requirements about QoI, the generated fidelity is always similar to a best case workload, i.e., a situation where the maximum number publishers that is required only provides suitable data. Simply speaking, using adaptation, our approach turns every situation into an ideal situation if the available publishers are capable of satisfying the requirements of a subscriber.

The fidelity generated in our setup when using EncodedTypes is increasing with increasing heterogeneity as shown by the ET line in Figure 6.6. This might seem counterintuitive at first glance but is quite logic on further analysis: although the fidelity remains negative, the continuous increase stems from the fact that only suitable publishers publish on those encoded types that are currently subscribed to; with a decreasing number of publishers, the percentage of redundant notifications diminishes. This is acknowledged by a less severe penalty.

Drill Down View On Data Processed in Each Configuration

We now analyze the different reasons for the fidelity shown in Figure 6.6. The heat maps in Figures 6.8 and 6.10 provide a drill-down view on these factors. The heat maps visualize and compare the results measured in each configuration for a given level of heterogeneity. Green indicates an overall satisfying result while darker colors up to black indicate unsatisfactory results as shown in Figure 6.7. Each row in a heat map corresponds to a configuration where a set of features is tested in isolation. Each column compares the results measured for different configurations when using the same workload. Each cell represents the results measured for a given configuration and level of heterogeneity: in Figures 6.8a and 6.9 each cell shows the percentage of notifications conforming to a given requirement while a cell in Figure 6.10 shows the measured average value for a property.



Figure 6.7.: Colors used in heat maps to indicate good or bad results.

As we can see from the heat maps in Figures 6.8a, 6.9a to 6.9d and 6.10e, using expectations with self-adaptation (ESA) satisfies all requirements regardless of the level of heterogeneity that is simulated. This is due to the ability of the EBS to adapt publishers that do not provide suitable

data at the beginning of a run. With expectations and self-adaptation, the EBS is able to provide notifications with the same properties than in a best-case workload (BC), even if the level of heterogeneity is high. This results in the consistently high fidelity shown in Figure 6.6.

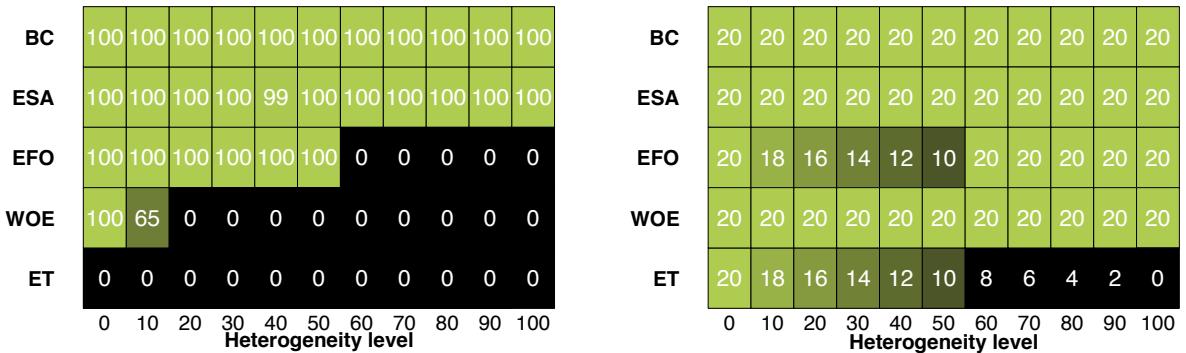


Figure 6.8.: Higher degree of conformance with the QoL requirements of subscribers: our approach (ESA) generates the same fidelity as with the best case workload (BC).

In comparison, not using adaptation but purely relying on filtering, as in EFO, works fine as long as there are sufficient suitable publishers available. As the EBS in EFO is unable to adapt publishers, the expectation cannot be satisfied anymore for $h > 50$ (cf., Figure 6.8a) as more than 50% of publishers in the setup do publish notifications with unsuitable characteristics. In this case, the expectation is declined and the EBS acts as a typical EBS by routing all notifications that conform to the subscription by their type or content. This turning point is most prominent for the number of alternatives shown in Figure 6.8b: the number of publishers that subscribers actually receive notifications from is declining from 20 to 10 for a heterogeneity up to 50% in compliance with the number of suitable publishers available at each level of heterogeneity. For $h > 50$ the expectation is rejected and the subscriber receives notifications from all publishers.

In ET, the EBS is able to provide notifications that conform to requirements about precision and accuracy using encoded types as shown in Figures 6.9a and 6.9b. However, the EBS does not have any control over the publishers and no way of adjusting the number of suitable publishers that subscribers consume data from as shown in Figure 6.8b. Please note that the number of publishers in ET is declining with increasing heterogeneity as only suitable publishers publish on the types subscribed to by subscribers in our scenario. The alternatives requirement is still violated in ET at all times as each publisher is sending with a sampling rate that is too high and thus is not considered as a suitable alternative (cf., Figure 6.9c and Figure 6.9d). This is due to each publisher having to publish the same notification to different encoded types. In Figure 6.10b, the results measured for Encoded Types (ET) have been omitted, as the latencies for this configuration are so extreme that they dwarf the differences between the other configurations in comparison and make this figure hard to comprehend. Publishers being overloaded cause the high latencies in the ET configuration.

While all notifications received by subscribers in EFO conform to their requirements for $h \leq 50$, the total number of notifications received by subscribers is decreasing due to the decreasing number of suitable publishers (cf., Figure 6.10a). As shown in Figure 6.10, number and characteristics of notifications measured in EFO converge to the behavior observed for WOE for $h > 50$.

Performance Evaluation

Quality of Service (QoS) Requirements

The QoS requirements are evaluated based on the following properties:

• Accuracy requirement: The percentage of notifications satisfying the accuracy requirement.

• Precision requirement: The percentage of notifications satisfying the precision requirement.

• Rate requirement: The percentage of notifications satisfying the rate requirement.

• Alternatives requirement: The percentage of notifications satisfying the alternatives requirement.

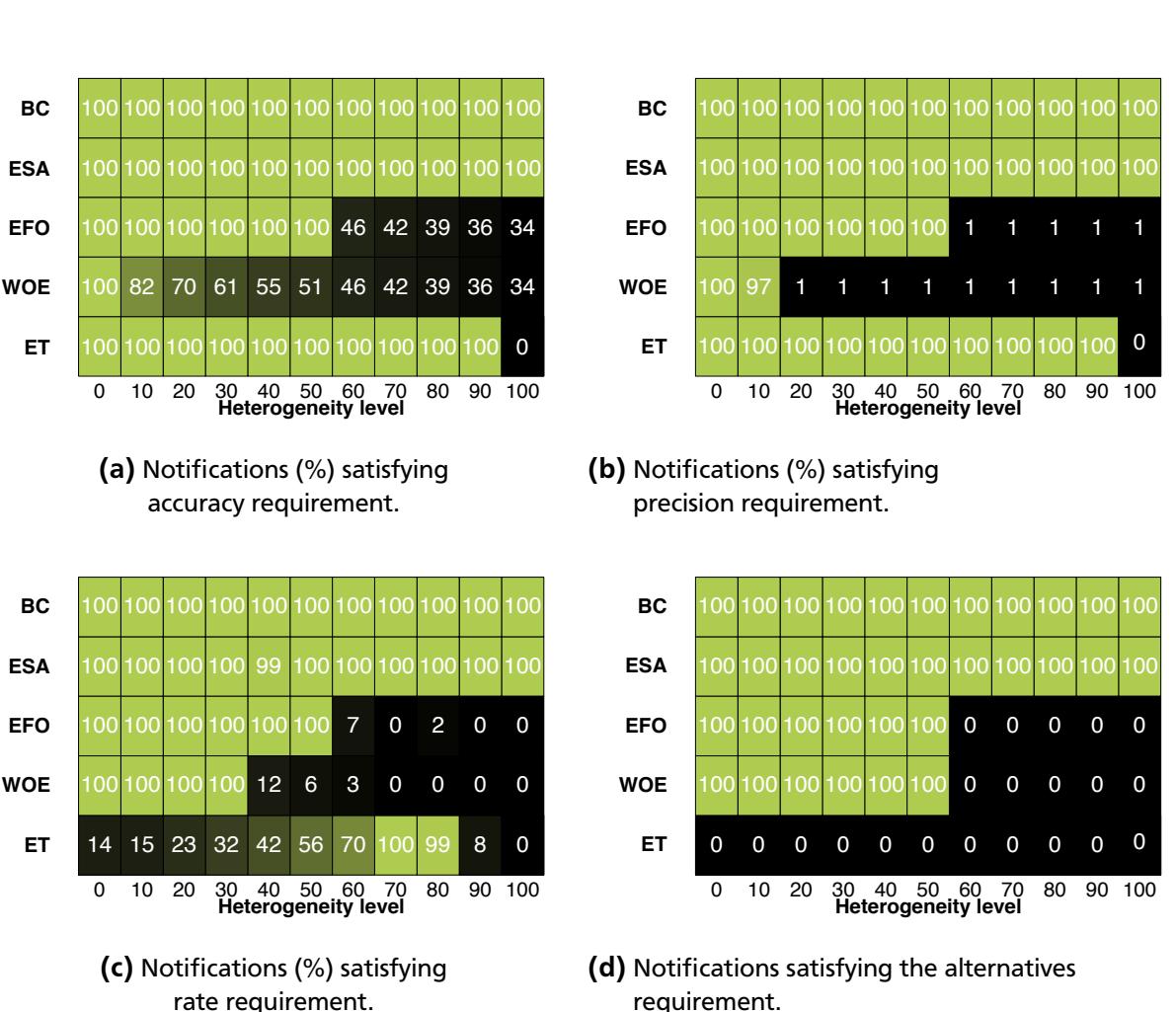
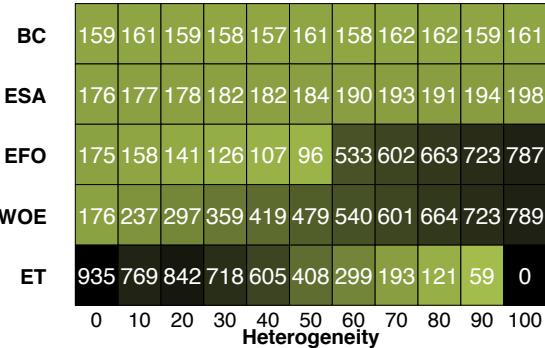
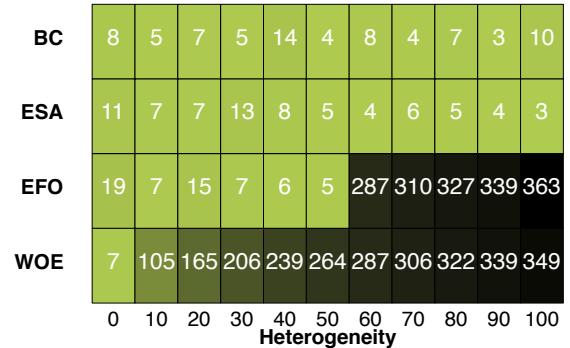


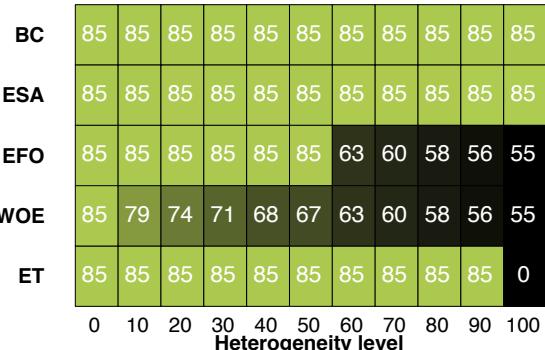
Figure 6.9: Higher degree of conformance with the QoS requirements of subscribers using expectations and self-adaptation (ESA); configurations using filtering only (EFO) or encoded types (ET) fall behind for one or more properties but still perform better than a typical EBS without any support for QoS (WOE).



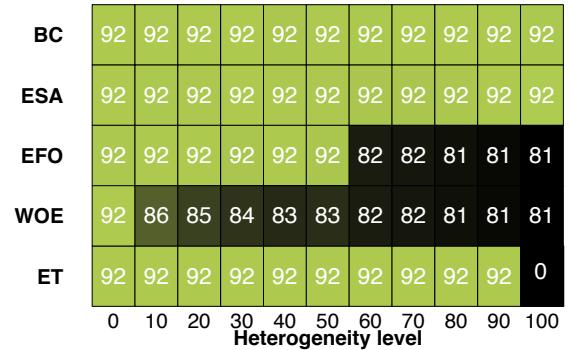
(a) Rates received (notifications/sec).



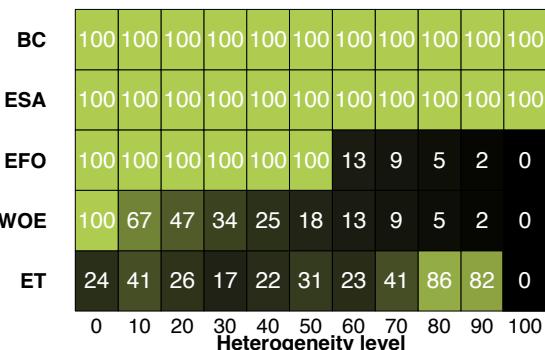
(b) Average latency of notifications (msec).



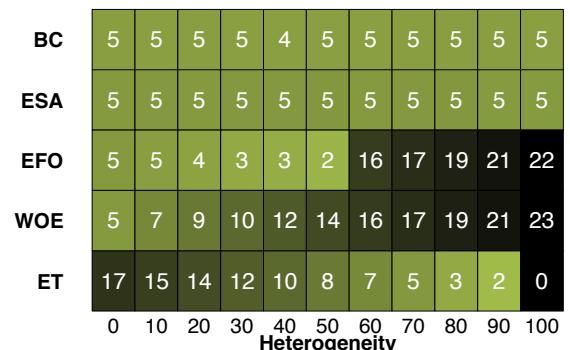
(c) Average accuracy of data received (%).



(d) Average precision of data received (%).



(e) Notifications (%) satisfying latency requirement.



(f) Absolute number of received notifications (K) in each configuration.

Figure 6.10.: Comparing properties of notifications received in different configurations: an EBS using expectations and self-adaptation (ESA) provides subscribers with notifications that have same characteristics than in the Best-Case workload (BC). When using only filtering but no adaptation (EFO) or encoded types (ET), the EBS cannot deal with high levels of heterogeneity but still performs better than an EBS without any support for QoI (WOE). Data for ET omitted in Figure 6.10b for the sake of clarity.

Dealing With Surplus Publishers

Please note that the fidelity generated in the baseline configuration (WOE) that represents a typical EBS is only positive for $h \leq 10$ because the number of publishers coincides with the upper bound for alternatives defined in X_5^{pos} (i.e., $10 \leq 20 \leq 20$).

Let us assume a slightly different scenario where a subscriber does not want to receive notifications from more than 18 publishers. Now it is crucial to reduce the number of publishers that the subscriber receives data from. As shown in Figure 6.11, the subscriber would never be satisfied by a typical EBS (WOE) as the EBS is not controlling the number of publishers – neither by adaptation (as in ESA) nor by filtering out notifications (as in EFO) based on QoI. The other configurations compensate surplus publishers by filtering out notifications (EFO, ESA) or even turning surplus publishers off (ESA).

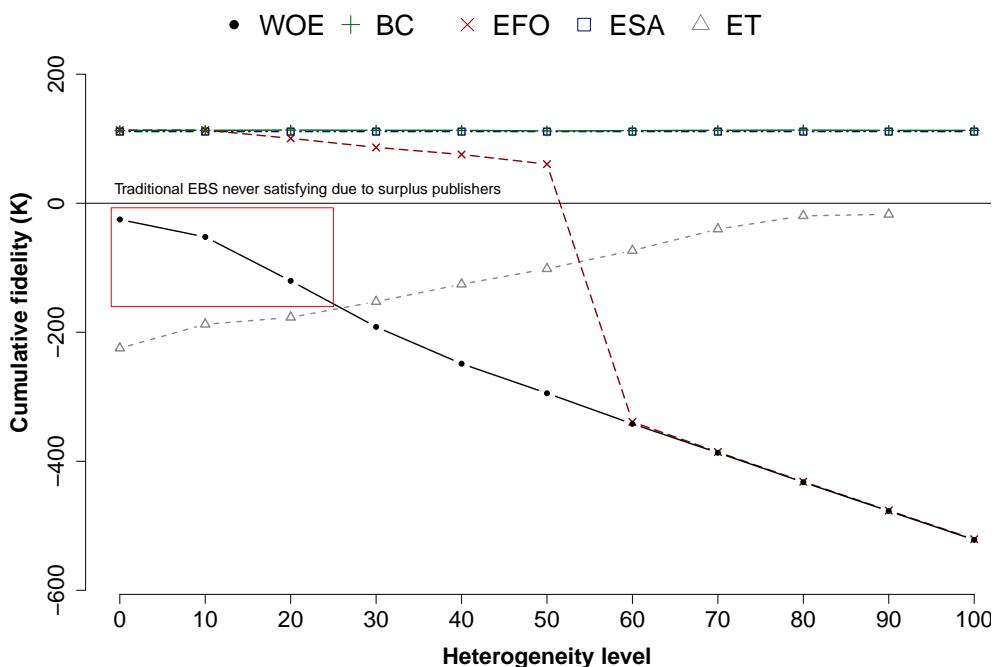


Figure 6.11.: Cumulative fidelity compared in case of surplus publishers ($p_{alt}.ub=18$): only ESA suffices, other configurations cannot deal with an insufficient number of publishers.

The heat maps containing detailed drill down data for the new set of requirements can be found in the appendix in Figures A.12 and A.13. They contain information about the percentage of notifications satisfying the new set of requirements in each configuration as well as details on the data measured for each configuration.

6.2.4 Benefits: Resource Savings

In this section, we gauge the benefits of our approach in terms of resource utilization for subscribers, MOM, and publishers. We compare the resource utilization regarding CPU utilization and network traffic for the different levels of heterogeneity across the configurations described in Section 6.2.2. We use the resource utilization monitored for a typical EBS (WOE) to compare against the resource utilization in all other configurations. All figures show the savings and overhead as Percentage Point (PP) of the respective utilization in WOE.

The resource utilization in terms of CPU utilization and network traffic are shown in Figure 6.14 for subscribers, in Figure 6.15 for the MOM, and in Figure 6.16 for publishers. In all cases we provide two sets of graphs: one comparing WOE with BC, EFO, and our ESA configuration; a separate graph also compares ET with WOE; this is due to the orders of magnitude that encoded types generate *overhead* instead of savings.

Subscribers

Subscribers would have to deal with a massive increase in CPU utilization in the baseline configuration compared to a best-case workload (BC). This is mainly due subscribers being flooded by an increasing number of unsuitable notifications at high rates (cf., Figure 6.10).

Subscribers benefit from all configurations where they receive only the data they require. This becomes apparent from the significant savings for the BC, ESA and EFO configurations shown in Figures 6.12a and 6.12b. When seen in isolation, the savings in EFO are even higher than for the best-case workload as less messages are processed in total than in any other configuration, including the best case.

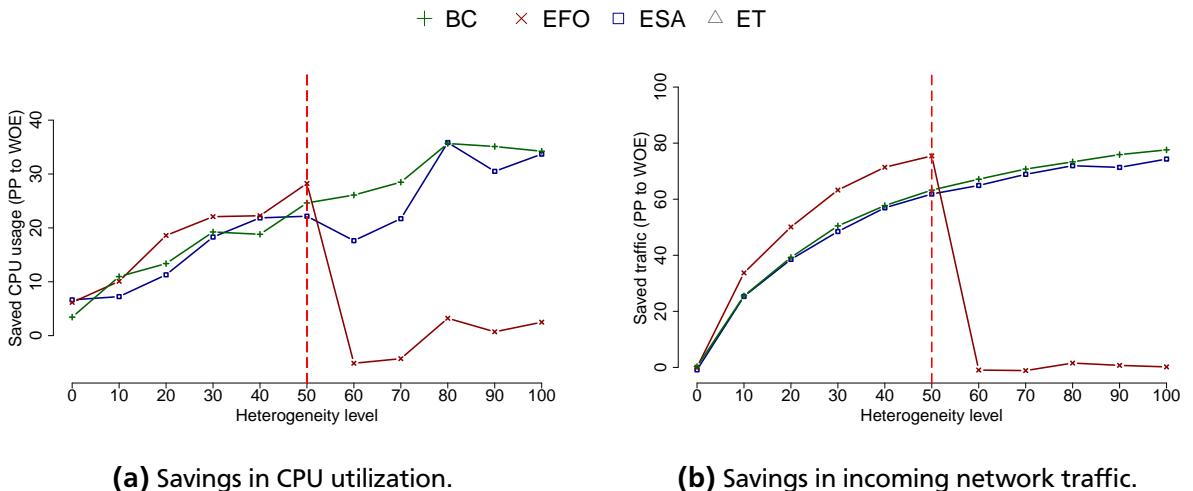
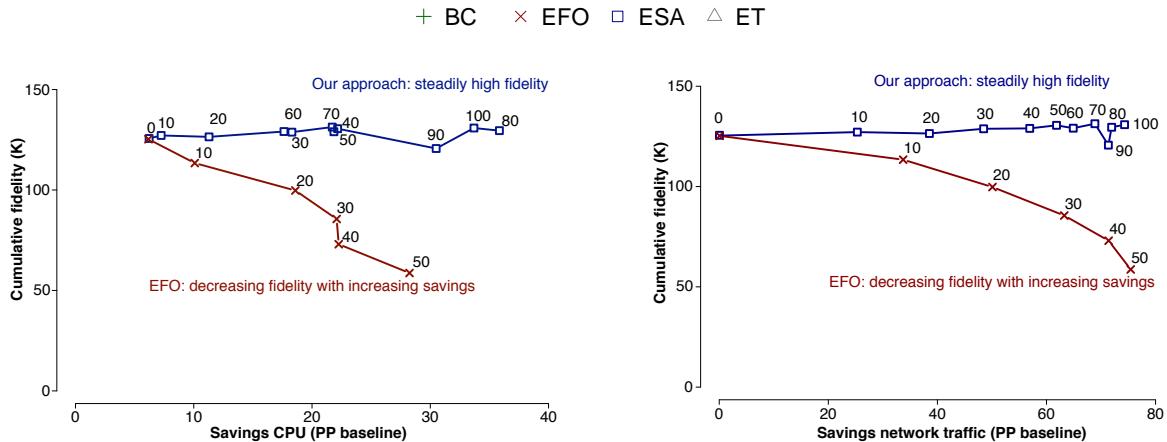


Figure 6.12.: Subscribers: savings and overhead regarding resource utilization measured in PP of WOE. Subscribers save resources when using expectations and self-adaptation: significant savings in terms of CPU utilization and incoming network traffic for subscribers for Expectations + Self-Adaptation (ESA); Best-Case workload (BC) and an EBS aware of QoL but not using self-adaptation (EFO) shown for comparison.

Putting these savings in context to the fidelity generated in each configuration, however, shows that subscribers pay for their additional savings in EFO with a decreasing fidelity that is a lot lower than in a configuration where our approach with self-adaptation is used. In contrast, a

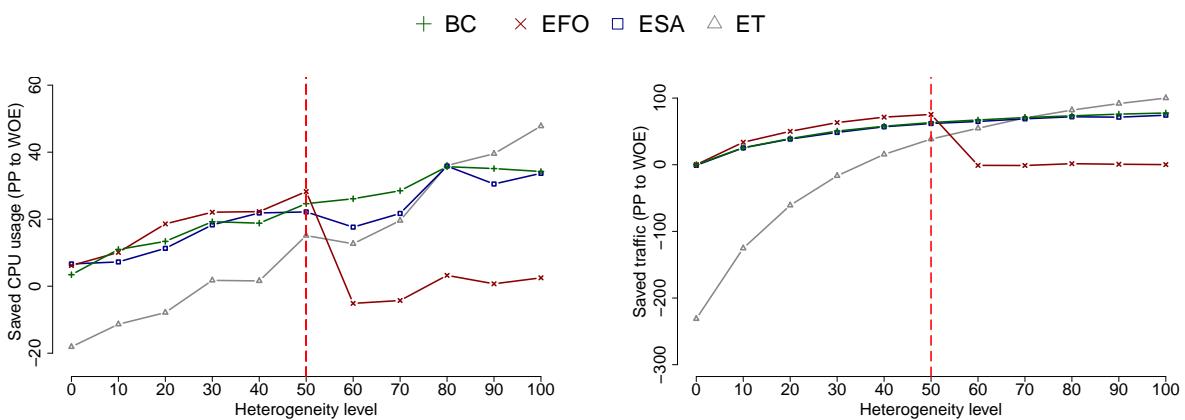
configuration based on our approach using self-adaptation (ESA) does not lead to a degradation in fidelity when savings increase. This is shown in Figure 6.13b for network traffic and in Figure 6.13a for CPU utilization: both figures show the cumulative fidelity generated in each configuration for different levels of heterogeneity plotted over the savings achieved by each configuration. The decreasing red line represents the EFO configuration while the ESA configuration is shown by a blue line.



(a) EFO: fidelity suffers with CPU utilization savings. (b) EFO: fidelity suffers with network traffic savings.

Figure 6.13.: Resource savings are bought dearly by decreasing fidelity for the subscriber in EFO. Our approach ESA, however, provides the same savings but does not impact fidelity.

Please note that subscribers do not benefit from savings in resource utilization in the ET. Rather, they suffer from additional overhead in this configuration, as the same suitable notification has to be published multiple times by each publisher and consumed by subscribers. As this overhead is several orders of magnitude higher than the overall savings even for the best-case workload, we have included them in a separate set of graphs in Figures 6.14a and 6.14b.



(a) Savings in CPU utilization (with ET).

(b) Savings in incoming network traffic (with ET).

Figure 6.14.: Subscribers: encoded types (Encoded Types (ET)) introduce a significant overhead.

The sudden drop in savings for EFO between a heterogeneity of 50 and 60 is due to the fact that there are not enough suitable publishers left for $h > 50$. This violates the alternatives requirement defined in X_5^{pos} as the EBS has no means of adjusting publishers in EFO. Thus, the

behavior in EFO resembles that of a typical EBS (WOE) in that subscribers are flooded with unsuitable notifications, zeroing any savings.

Message-oriented Middleware

Apart from subscribers, the MOM is also benefiting from using our approach. The savings for the MOM shown in Figure 6.15 are more significant than for subscribers as the MOM has additional effort for discarding unsuitable notifications in EFO or processing them in WOE. This effort is nil when the MOM is able to advise publishers to adapt and thus minimizes traffic overhead right at the source (ESA). For the MOM, the savings in EFO are not as high as for subscribers. This is due to the MOM having to discard an increasing number of notifications that do not adhere to the requirements of subscribers until the alternatives requirement is not satisfied anymore.

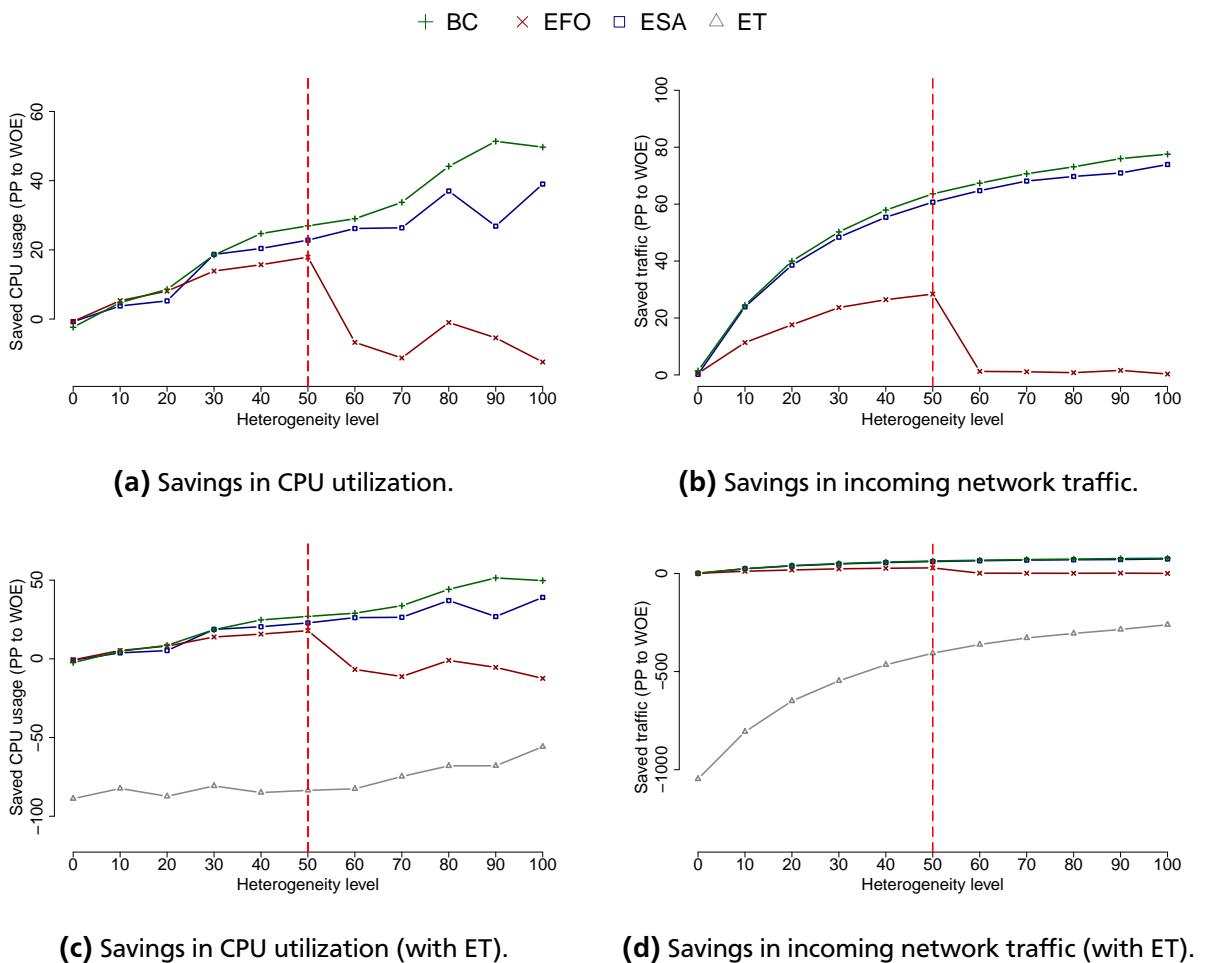


Figure 6.15.: MOM: savings and overhead regarding resource utilization measured in PP of WOE.
Using expectations and self-adaptation results in significant savings for the MOM in terms of CPU utilization and incoming network traffic. Encoded Types (ET) shown in separate graphs 6.15c and 6.15d as they introduce a significant overhead.

The difference between the best-case workload (BC) and our proposed approach (ESA) is still significant. This is due to the additional effort required by the MOM to negotiate expectations with capabilities and send feedback. Applying this feedback and adapting also takes time during which still unsuitable notifications arrive at the MOM and have to be discarded.

Publishers

Even publishers can profit from a setup where the EBS applies our approach of expectations, capabilities and feedback. As shown in Figure 6.16, adapting unsuitable publishers to publish at a lower sampling rate in the ESA scenario frees up resources to the same degree as in the best-case scenario (BC). Please note that publishers do not profit from a setup where insufficient notifications are discarded at the MOM without notifying the publisher that its notifications are not used (EFO); also note the significant overhead introduced by using encoded types (ET).

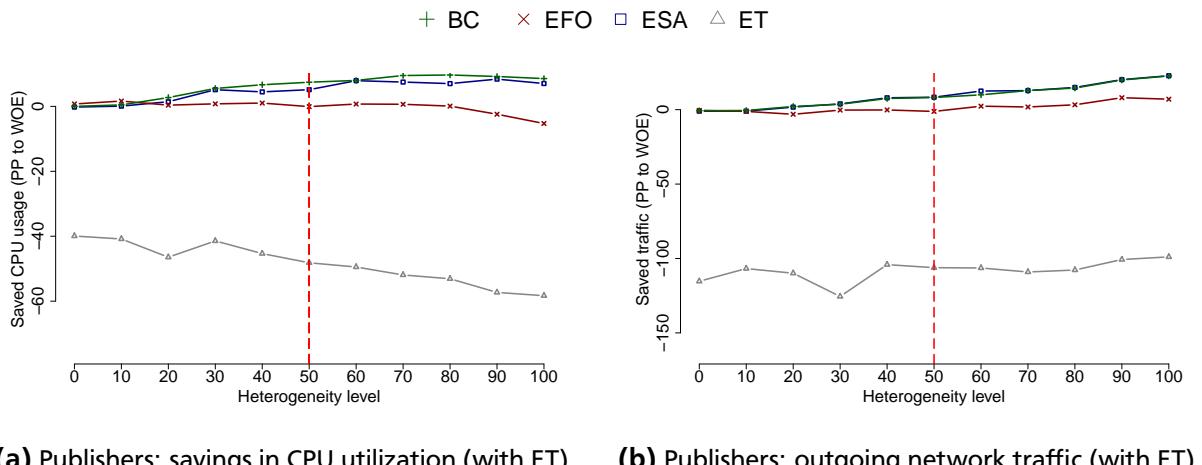


Figure 6.16.: Publishers: savings and overhead regarding resource utilization for publishers measured in PP of WOE. Better resource utilization for publishers when using expectations and self-adaptation (ESA). Significant overhead results from a configuration that uses encoded types (ET) while filtering out unsuitable notifications at the MOM does not have any advantages for publishers (EFO).

6.3 Scalability and Execution Costs for Brokers Applying our Approach

Negotiating expectations with capabilities, deciding on suitable adaptations, and coordinating feedback comes at a cost for the MOM. We have briefly touched upon this already in Section 6.2.4 when discussing the resource utilization of participants in the heterogeneity scenario.

In this section, we investigate the execution costs for the MOM in more detail. The key questions regarding execution costs and scalability we are going to answer are

1. What are the limits of the current prototype on a single broker in terms of the number of supported participants, throughput, dynamics and number of generic properties?
2. Which part of the system becomes a bottleneck?
3. What are the cost drivers for the execution costs and how do they influence them?

We maximize the reproducibility and comparability of our results by relying on a standardized and industry-strength benchmark to evaluate our prototype built on top of ActiveMQ.

We use jms2009-PS [368, 369] that extends the official Standard Performance Evaluation Corporation (SPEC) benchmark¹ SPECjms2007 for Java Message Service (JMS). SPECjms2007 is a general MOM benchmark that tries to exploit a wide spectrum of functionalities provided by most MOMs [259, 370, 371, 372]. Hence it focuses on both point-to-point and many-to-many communication models as well as a diverse set of message types [369, 371, 372]; transactional behavior and durable subscriptions based on persistence are other features exploited by SPECjms2007 but irrelevant to our evaluation.

The costs and benefits of the many-to-many communication model that is particular for PS and EBS are addressed and examined by jms2009-PS in more detail using the workload and scaling strategy of SPECjms2007 [368, 369].

Like for SPECjms2007, a single run of jms2009-PS is successful ("passed") for a given configuration if the MOM processes the workload and maintains

- the completeness and order of notifications (i.e., no notifications are lost);
- the defined received rate for 95% of all notifications;
- an end-to-end latency of at most 5000msec for 90% of the processed notifications.

We have extended jms2009-PS to include expectations and capabilities in its scaling strategy as described in Section 5.3.

6.3.1 Used Scenario and Characterization of Workload

The workload used by SPECjms2007 and jms2009-PS is based on an Supply Chain Management (SCM) scenario where the flow of goods is tracked by Radio-Frequency IDentification (RFID) sensors and managed by an EBS. Four types of participants interact in seven interactions by exchanging notifications over a MOM [368, 370, 371, 372]. Each interaction models a complete business operation. Each type of participant represents a role in the scenario: supermarkets (SMs), suppliers (SPs), headquarters (HQs), or distribution centers (DCs). Participants exchange information based on a given number of product families that are sold in SMs, provided by SPs, and distributed using DCs. Each instance of an SM is selling different products based on the workload definition. The modeled interactions are [368]:

¹ <http://spec.org/jms2007/>

1. Order and shipment handling between SM and DC;
2. Order and shipment handling between DC and SP;
3. Price updates sent from HQ and SM;
4. Inventory management inside SM;
5. Sales statistics sent from SM to HQ;
6. New product announcements sent from HQ to SM; and
7. Credit card hot lists sent from HQ to SMs.

Depending on the interaction, notifications of different types (e.g., Order, Confirmation, CreditCardHotlist) and formats (e.g., binary, Extensible Markup Language (XML), plain text) are exchanged between instances of publishers and subscribers that represent different physical locations in the scenario [370]. The content of each notification is based on the identity of the communication partners, the interaction and the step of each interaction. Upon receiving a notification, each subscriber checks the conformance of the received notification with the content assumed at this step from the respective publisher.

As each interaction models a complete business operation between participants in the scenario, there are no surplus notifications, subscribers, or publishers; each notification is relevant as well as the temporal order notifications are received with. Thus we, cannot apply the reactive behavior we have utilized in Section 6.2 without violating the constraints defined by the benchmark.

The workload generated by jms2009-PS/SPECjms2007 can be scaled with the BASE scaling parameter. BASE is an abstract metric of SPECjms2007 that scales the workload either in terms of the population (horizontal topology) or throughput (vertical topology). The size of the workload generated by SPECjms2007 in each topology is monotonically increasing with an increasing value for BASE. The results obtained for a given BASE value, however, cannot be interpreted directly but only compared to results obtained with other values for BASE where a higher value for BASE always indicates a higher load for the system.

In the horizontal topology, the BASE_h parameter scales the number of physical locations (SMs, SPs, DCs) and the number of product families emulated by the benchmark. The number of notifications exchanged per combination of interaction, product family and physical location, however, is constant. The BASE_h parameter defines the number of physical SMs that have to be emulated by different SM instances. Per SM instance, a configurable but fixed number of instances for DCs and SPs are generated using the formulas given in [370, 372]. jms2009-PS initializes a preconfigured number of publishers and subscribers for each physical location and interaction to share the load, i.e., to send all notifications defined in the workload to the MOM and check the received copies at subscribers to determine the KPIs for the MOM.

In the vertical topology, the number of emulated physical locations is fixed while the sampling rate for each combination of interaction, message type and participant is scaled. The BASE_v parameter scales the number of products sold in each supermarket SM while the number of physical locations (SMs, SPs, DCs, HQs) remains constant. This results in an increased traffic that has to be handled by the MOM.

Using the horizontal topology, we can gauge the impact of large population on the performance of the MOM when using expectations and capabilities while we can use the vertical topology to gauge the overhead when scaling higher throughput generated by a small population.

6.3.2 Tailoring jms2009-PS to Gauge Execution Costs of Runtime Negotiation

The configuration for all our experiments is based on the published SPECjms 2007 evaluation results for ActiveMQ V5.4 provided by Sachs et al. [395].

In particular, we have not changed the configurations for

- transactional vs. non-transactional notifications;
- persistent vs. non-persistent notifications;
- usage of different message types, e.g. TextMessages or ObjectMessages;
- usage of notifications of different sizes (small, medium, large); and
- durable vs. non-durable subscriptions.

Durable subscriptions and persistence requirements at the MOM are supported by in-memory storage in our experiments. This way, we minimize the risk of introducing additional performance bottlenecks that would distort the performance results as our evaluation does not focus on durable subscriptions or persistence.

However, we had to tailor jms2009-PS to be suitable for evaluating the execution costs of our prototype by adjusting the configuration as well as extending the source code of jms2009-PS to send, receive and monitor expectations and capabilities. We have been in close contact with the authors of jms2009-PS to maintain the reliability of the benchmark.

The configuration of jms2009-PS can be adjusted to a certain degree without compromising the validity of its results. Both jms2009-PS and SPECjms2007 provide a set of tuning parameters to attune the benchmark to a given test environment and MOM [259, 368, 369, 371]. In jms2009-PS, the communication model (i.e., point-to-point using queues or many-to-many using topics) can be configured for each combination of interaction, destination instance, message type, and product family. This way, different communication styles (i.e., point-to-point, topic-based, event-bus) can be simulated and their costs compared [368, 369]. We have configured jms2009-PS to use topics wherever possible. Furthermore, we maximize the number of different topics to be maintained for each BASE by configuring the benchmark to generate a new topic for each combination of physical location (e.g., SM_2, DC_1) and message type (e.g., Order_ SM_1, SP_2). As jms2009-PS generates dedicated publishers and subscribers for each generated topic, we scale the total number of publishers and subscribers emulated by jms2009-PS with an increasing $BASE_h$.

Extending the source code of jms2009-PS has been necessary to enable publishers and subscribers to register, update, and revoke expectations and capabilities as well as receive and log feedback provided by the MOM. Details on the necessary extensions can be found in Appendix A.6.

In addition to the configuration parameters offered by SPECjms2007 and jms2009-PS to configure the workload, we also provide additional parameters to configure the size of each expectation and capability profile in terms of the number of generic properties, as well as the frequency with that publishers and subscribers update their expectations and capabilities to simulate changes in their context or state (e.g., due to self-adaptation).

The effect of satisfied or unsatisfied expectations on the fidelity of subscribers and the resource utilization of participants in general has already been explored in Section 6.2. In this part of the evaluation, we focus only on the effort necessary for the MOM to reach a conclusion about an expectation and the set of capability profiles available at that moment, regardless of the result being a satisfied or unsatisfied expectation. Thus, we generate random definitions for the number of generic properties that should be used during each single run. These definitions

balance range-based and list-based generic properties for both numeric and textual types. Based on these randomized definitions, we also generate expectations and capabilities for each agent with randomized ranges of accepted values. The randomized values used are based on a uniform distribution. We ensure that each randomly generated expectation/capability profile is valid in that it is defined over the ranges defined in each definition with each lower bound being smaller or equal to the randomized upper bound.

6.3.3 Testplan, Scaling Parameters and Measured KPIs

Using jms2009-PS, we want to investigate the execution costs of our approach at the MOM. We want to analyze how using our approach to support QoI at runtime impacts the maximum number of participants or the maximum throughput a single ActiveMQ broker can still handle without violating the constraints set by SPECjms2007 regarding received rates, completeness, order and end-to-end latency of notifications.

As illustrated in Figure 6.17, the maximum population and the maximum throughput still manageable by a bare ActiveMQ broker are identified by the maximum values for $BASE_h$ and $BASE_v$. These upper bounds represent the overall system limits and our baseline. Within these boundaries we can explore how negotiating expectations and capabilities impacts the ability of a single broker to deliver valid results when running jms2009-PS against it. In particular, we want to investigate the effect of varying the number of generic properties that expectations are defined over as well as the frequency that participants update their preferences or capabilities at runtime.

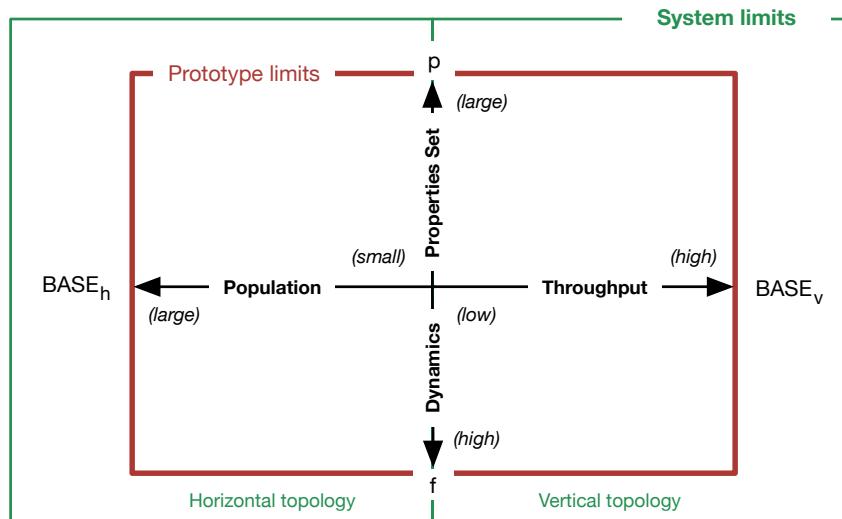


Figure 6.17.: Experiments explore the limits of the current prototype compared to the system limits determined by the baseline measurements for both $BASE_h$ and $BASE_v$.

Thus, we first calibrate our setup by running jms2009-PS against a bare ActiveMQ to identify the range² of values for $BASE_h$ and $BASE_v$, that represent the overall system limits. We vary $BASE_v$ and $BASE_h$ while participants emulated by jms2009-PS do not provide expectations and capabilities or request feedback, hence not putting any extra load on the broker.

Having established a baseline for both the horizontal and the vertical topology, we activate the support for expectations and capabilities in jms2009-PS and run it against a single ActiveMQ

² Please note that includes finding lower bounds for our configuration, as too low values for $BASE_h$ or $BASE_v$ can result in unpredictable results (cf., <http://spec.org/jms2007/docs/FAQ.html#Q37>).

broker that deploys our prototype. Comparing the measured results against the results obtained for a comparable baseline allows us to quantify the execution costs.

For each single run of jms2009-PS we vary

1. BASE_h in the horizontal topology to scale the number of active publishers and subscribers that have QoI requirement or capabilities which need to be negotiated at runtime; or
2. BASE_v in the vertical topology to scale the throughput of notifications to be processed by the MOM in addition to processing and negotiating expectations and capabilities;
3. p to scale the number of generic properties each expectation and capability profile is defined over by each participant; and
4. f as the frequency of changes to a participant's context or state that result in changing expectations or capabilities at runtime and requiring the MOM to renegotiate.

In a first set of tests, we vary only one parameter while keeping the others fixed (*ceteris-paribus*). In a second set of tests, we vary two parameters *ceteris-paribus*. In a third set of tests, we finally vary all three parameters for a given topology. We test the same combinations of parameters for both BASE_h and BASE_v , to gauge the impact of large population with moderate throughput and of small populations with high throughput.

We vary the number of generic properties each expectation and capability profile is defined over from 5 to 50 in steps of 5 (i.e., $p \in [5; 50]$) while we vary the duration that each participants waits before sending an update about its preferences or capabilities from 1000 seconds to 2 seconds. The resulting update frequency $f = \frac{1}{d}$ is defined over the duration d in seconds elapsing between a single update is triggered by *each* participant. (i.e., $f \in [0.001; 0.2]$).

Please note that this configuration results in *bursts* of updates that the MOM has to deal with. This behavior has been designed on purpose to stress the MOM to a maximum degree when scaling the population, the update frequency, or the number of generic properties: a single peak in processing overhead can already result in enough temporary routing backlog at the MOM to violate the constraints set by jms2009-PS in terms of the end-to-end latency of notifications.

For each variation of our scaling parameters, we compare the resource utilization of an ActiveMQ messaging broker running our prototype with the resource utilization of a bare ActiveMQ instance processing the same workload, i.e., using the same value for BASE_h or BASE_v .

We measure the following five KPIs for the MOM to quantify the execution costs in terms of the overhead our prototype adds to a baseline configuration:

1. CPU utilization;
2. memory usage;
3. incoming and outgoing network traffic; and
4. end-to-end latency of notifications.

As we consider only results obtained from successful jms2009-PS test runs, we can assume that the order and completeness of notifications is maintained.

We run each set of tests multiple times and average the values measured for each KPI. In total we have been running more than 900 successful runs of jms2009-PS with varying scaling parameters against our prototype deployed on a single ActiveMQ broker. The measurement phase of each run is 30 minutes as required by the benchmark, preceded by a warmup phase of 15 minutes and succeeded by a drain phase of another 15 minutes.

Table 6.6.: Baseline: linear regression analysis results for bare ActiveMQ: $KPI \approx \alpha + \beta \cdot BASE + \epsilon$

KPI	Horizontal $BASE_h$				Vertical $BASE_v$			
	α_h	β_h	R^2	adj. R^2	α_v	β_v	R^2	adj. R^2
CPU utilization	15.8	3.01	1.00	1.00	17.6	0.62	0.99	0.99
Memory Usage	18.9	0.56	0.95	0.95	24.0	0.08	0.89	0.88
Network traffic	0.1	0.27	1.00	1.00	0.0	0.11	1.00	1.00

6.3.4 Discussion of Measured Results

In the remainder of this section we discuss the results obtained from our experiments.

Baseline: Measured System Limits

The resulting baseline measurements obtained for running jms2009-PS against an ActiveMQ broker without our prototype show that Central Processing Unit (CPU) utilization, memory usage and network traffic increase linearly with an increasing $BASE$. This hypothesis of a linear relationship for both the horizontal and the vertical topology cannot be rejected as shown by the linear regression analysis results listed in Table 6.6.

SPECjms2007 and jms2009-PS have not been designed with the objective to allow for fine-tuning the number of active publishers and subscribers. Thus, we can only indirectly influence them in the horizontal topology by varying $BASE_h$. We have measured the impact of varying $BASE_h$ and $BASE_v$ on the number of publishers and subscribers that are emulated by jms2009-PS. For each run, we log the identities of the different subscribers and publishers that are emulated by jms2009-PS and maintain JMS connections to the broker. 1790 participants are active for $BASE_h = 21$ while 1252 participants are active for $BASE_h = 15$. In the horizontal topology, the population varies scales linearly between ≈ 800 participants for $BASE_h = 10$ and 2062 participants for $BASE_h = 25$. In contrast, the population in all vertical tests is 19 and does not vary when scaling $BASE_v$.

In the horizontal topology, a bare ActiveMQ is saturated for a $BASE_h \geq 25$, which corresponds to ≈ 2000 participants processing an average total rate of ≈ 2100 notifications per second. In the vertical topology, a bare ActiveMQ is saturated for a $BASE_v \geq 125$, which corresponds to an average total rate of ≈ 4000 notifications processed per second by 19 participants.

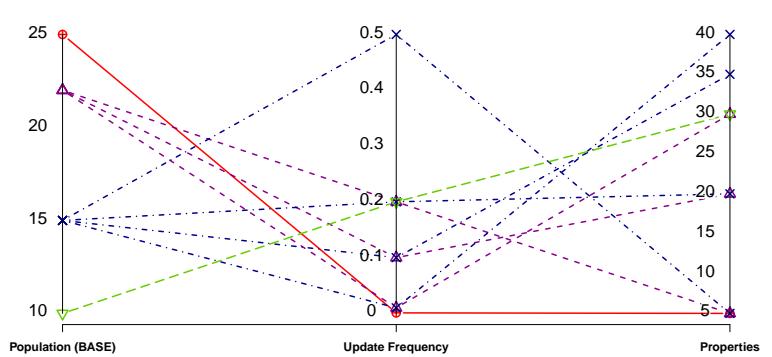
The obtained ranges of $BASE_h = [10; 25]$ and $BASE_v = [35; 125]$ that constitute the system limits correspond to the results published by Sachs et al. in [368] for an older version of ActiveMQ. We use these ranges for our tests when varying the population ($BASE_h$) or throughput ($BASE_v$). In cases where we keep the population fixed, we set $BASE_h = 15$; where we want to keep the throughput fixed, we set $BASE_v = 50$.

Measured Prototype Limits and Trade-offs

The key findings of our quantitative evaluation using jms2009-PS are summarized in Figure 6.18: larger populations (higher $BASE_h$) or higher throughput (higher $BASE_v$) can be supported when update frequencies are lower or less generic properties are used to define expectations/capability profiles (and vice-versa).

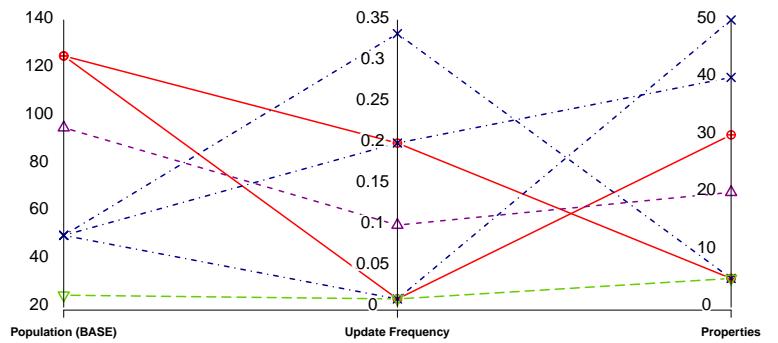
The figure shows the skyline of our experimental results for the horizontal (Figure 6.18a) and vertical (Figure 6.18b) topology, illustrating both the limits of our prototype and the trade-offs between the cost drivers in each topology. Each line represents a combination of population size (BASE_h)/throughput (BASE_v), update frequency $f = \frac{1}{d}$ and the size of each expectation and capability profile (p as the number of generic properties) that successfully passes jms2009-PS where at least one parameter is not dominated by others.

BASE_h	f	P	Symbol
25	0.001	5	\oplus
22	0.010	30	\times
22	0.100	20	\times
22	0.200	5	\times
15	0.010	40	\triangle
15	0.100	35	\triangle
15	0.200	20	\triangle
15	0.500	5	\triangle
10	0.200	30	∇



(a) Skyline of population size, update frequency and number of generic properties (horizontal topology).

BASE_v	f	P	Symbol
125	0.010	30	\oplus
125	0.200	5	\oplus
95	0.100	20	\times
50	0.010	50	\triangle
50	0.200	40	\triangle
50	0.333	5	\triangle
25	0.010	5	∇



(b) Skyline of throughput, update frequency and number of generic properties (vertical topology).

Figure 6.18.: Skyline of successfull parameter combinations show the trade-off between population/throughput, update frequency, and number of generic properties.

The results obtained in the horizontal topology show that a single ActiveMQ broker can support QoI requirements of more than 2000 participants ($\text{BASE}_h = 25$) without saturating its CPU and deteriorating the latency and order of notifications. In our experiments, this can only be achieved, however, if expectations and capability profiles are defined over at most five generic properties each and the minimum duration between updates from a participant is at least 1000 seconds.

The trade-off between those parameters becomes obvious when comparing this configuration to a situation with 10% less participants ($\text{BASE}_h = 22$), the same number of generic properties per expectation and capability profile ($p = 5$) but an update frequency that is 200% higher: all participants can update their requirements or capabilities at least every 5 seconds ($d = 0.2$). Alternatively, a population of the same size ($\text{BASE}_h = 22$) can be supported when expectations and capabilities become four times larger ($p = 20$) but are updated only half as often ($d = 0.1$). For the same population ($\text{BASE}_h = 22$), we can support 30 properties per expectation/capability profile ($p = 30$) if all participants update at least every 100 seconds ($d = 0.01$).

Again, trading off participants for generic properties allows us to increase the number of generic properties to the maximum of 40 generic properties per expectation/capability profile we have measured: ≈ 1300 participants ($\text{BASE}_h = 15$) can be supported while keeping the same update frequency ($d = 0.01$). The same population can be supported in our experiments for updates sent every 2 seconds ($d = 0.5$) if we limit the set of generic properties to five again.

For small populations but higher throughput, our prototype is able to provide support for QoI even for 50 generic properties per expectation/capability profile that are updated every 100 seconds ($d = 0.01$) if the average throughput per second does not exceed ≈ 1500 notifications per second ($\text{BASE}_v = 50$). For the same update frequency, a maximum set of 30 generic properties can be supported for the maximum throughput of ≈ 4000 notifications per second ($\text{BASE}_v = 125$). Maintaining this throughput, all 19 participants could even send updates every 5 seconds ($d = 0.2$) if the number of generic properties is reduced to five per expectation.

Relating these measurements to the building blocks of our model we have discussed in Chapter 3 shows that the maximum number of generic properties that a single instance of our prototype is able to support corresponds to the total number of generic properties we have discussed in Section 3.2.1. While having to reduce the number of generic properties to support very large populations ($\text{BASE}_h = 25$) or high throughput ($\text{BASE}_v = 125$), our prototype is still able to negotiate expectations and capability profiles defined over the same number of properties we have discussed in Section 6.1.

Drill-down: Investigating the Impact of Parameters

Having identified and analyzed the limitations of our prototype, we now investigate the cost drivers and their impact. The data visualized in Figures 6.21 to 6.28 provides a more detailed drill-down view on the impact of each parameter and the combination of parameters when applied to a scenario where the MOM has to deal with large populations (horizontal topology) or high throughput (vertical topology). Thus, the results measured for the same KPI (i.e., CPU, memory, network traffic, latency) and combination of parameters are shown side by side for the horizontal and vertical topology.

The obtained results show that CPU becomes a bottleneck for high update frequencies, large populations and/or high throughput (cf., Figure 6.21). Varying two or three of these cost drivers together amplifies the impact on the CPU as shown in Figures 6.25 and 6.29. Memory becomes a bottleneck besides CPU when a single broker has to support large populations that require the broker to negotiate a large set of generic properties (cf., Figure 6.26). Network does not become a bottleneck because the traffic overhead is negligible (cf., Figures 6.23, 6.27 and 6.29).

While the latency increases for large populations ($\text{BASE}_h > 18$) or very high throughput ($\text{BASE}_v = 120$), this increase is due to the CPU being more and more occupied in these situations. The resulting maximum latencies³ of 363msec for vertical and 352msec for horizontal, however, are still well within the limits of 5000msec defined by SPECjms2007.

The regression analysis performed for each parameter shows a linear relationship between an increasing overhead for CPU and memory with an increasing number of participants (BASE_h),

³ SPECjms2007 periodically reports the 90th latency percentile for each interaction during the measurement period; we average these values across all interactions. The numbers presented here represent the maximum values measured during multiple iterations.

throughput (BASE_v), number of generic properties, or update frequency (cf., Tables A.5 and A.6).

Investigating the impact of the update frequency, we have defined three categories of update behavior: *scarce*, *moderate* and *aggressive*. For each category, we use a different step size to increase the update frequency with: for *scarce*, we decrease the number of seconds each participant waits before sending an update from 1000 seconds to 100 seconds in steps of 50 (i.e., [0.001; 0.01]); for *moderate*, we decrease in steps of 10 from 100 seconds to 10 seconds (i.e., [0.01; 0.1]); and for *aggressive*, we decrease *every* second from 10 seconds to 2 seconds (i.e., [0.1; 0.5]). The measured results are visualized in Figure 6.19 for the horizontal and in Figure 6.20 for the vertical topology; the linear regression results for each category are listed in Table A.7. The results show that an update behavior as simulated in *scarce* does not impact CPU, memory or traffic. Update behavior as in *moderate* and *aggressive*, however, impacts CPU linearly; update behavior in *aggressive* does also linearly impact memory consumption and network traffic in our experiments for larger populations (cf., Figures 6.19f and 6.19i) or high throughput (cf., Figures 6.20f and 6.20i). Latency is never affected in our experiments from just varying the update frequency unless the CPU becomes saturated.

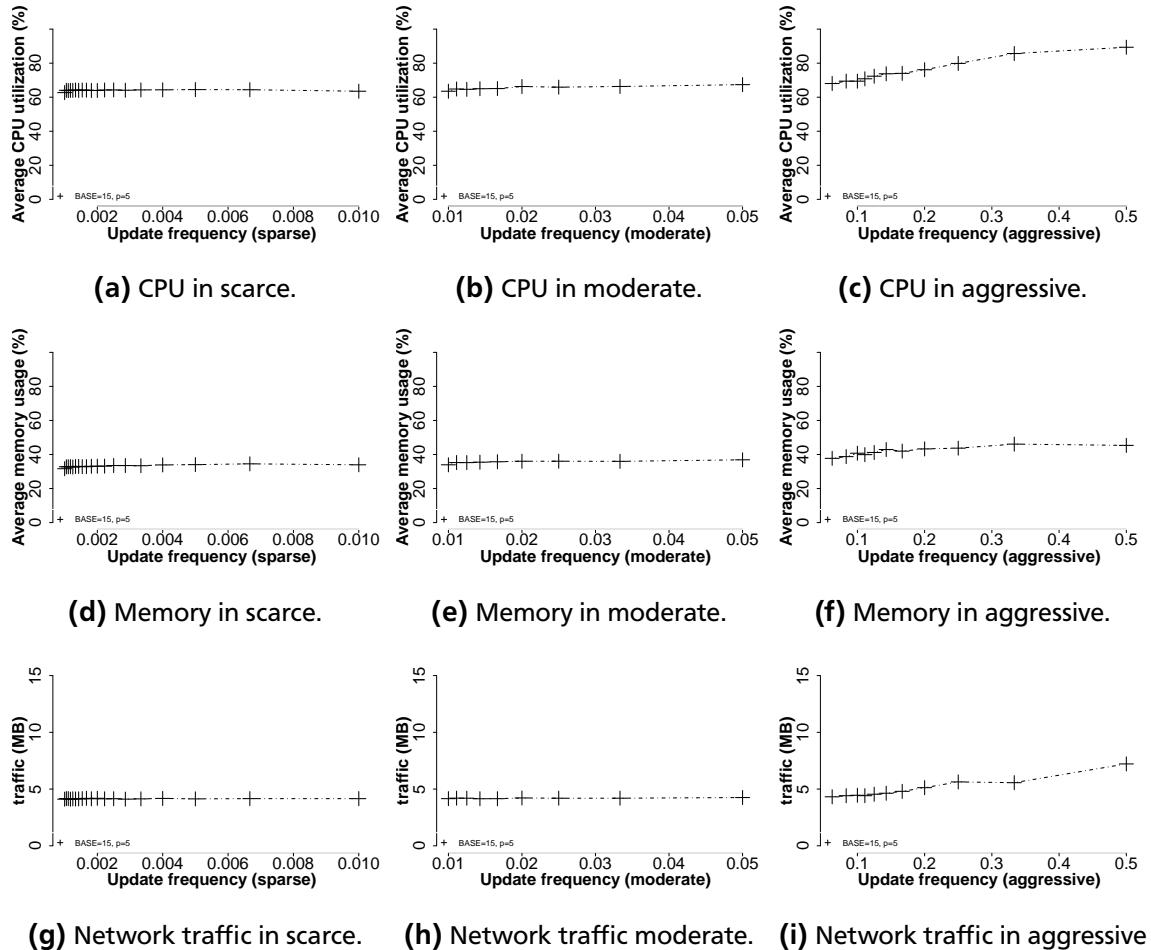


Figure 6.19.: Population: impact on CPU utilization, memory usage, and network traffic when varying the update frequency simulating either *scarce*, *moderate*, or *aggressive* update behavior of large populations in the horizontal topology.

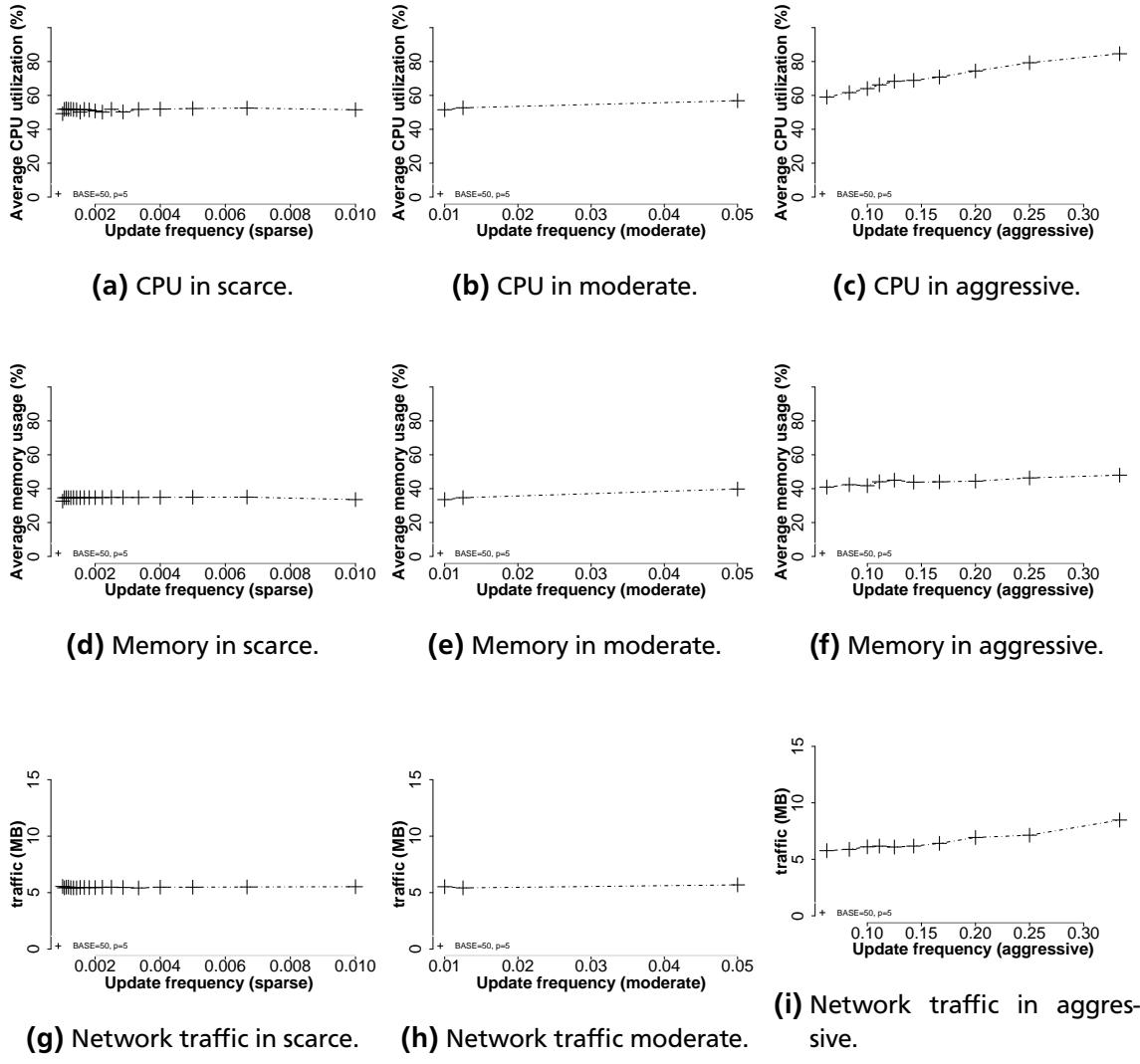


Figure 6.20.: Throughput: impact on CPU utilization, memory usage, and network traffic when varying the update frequency simulating either *scarce*, *moderate*, or *aggressive* update behavior for small populations but high throughput in the vertical topology.

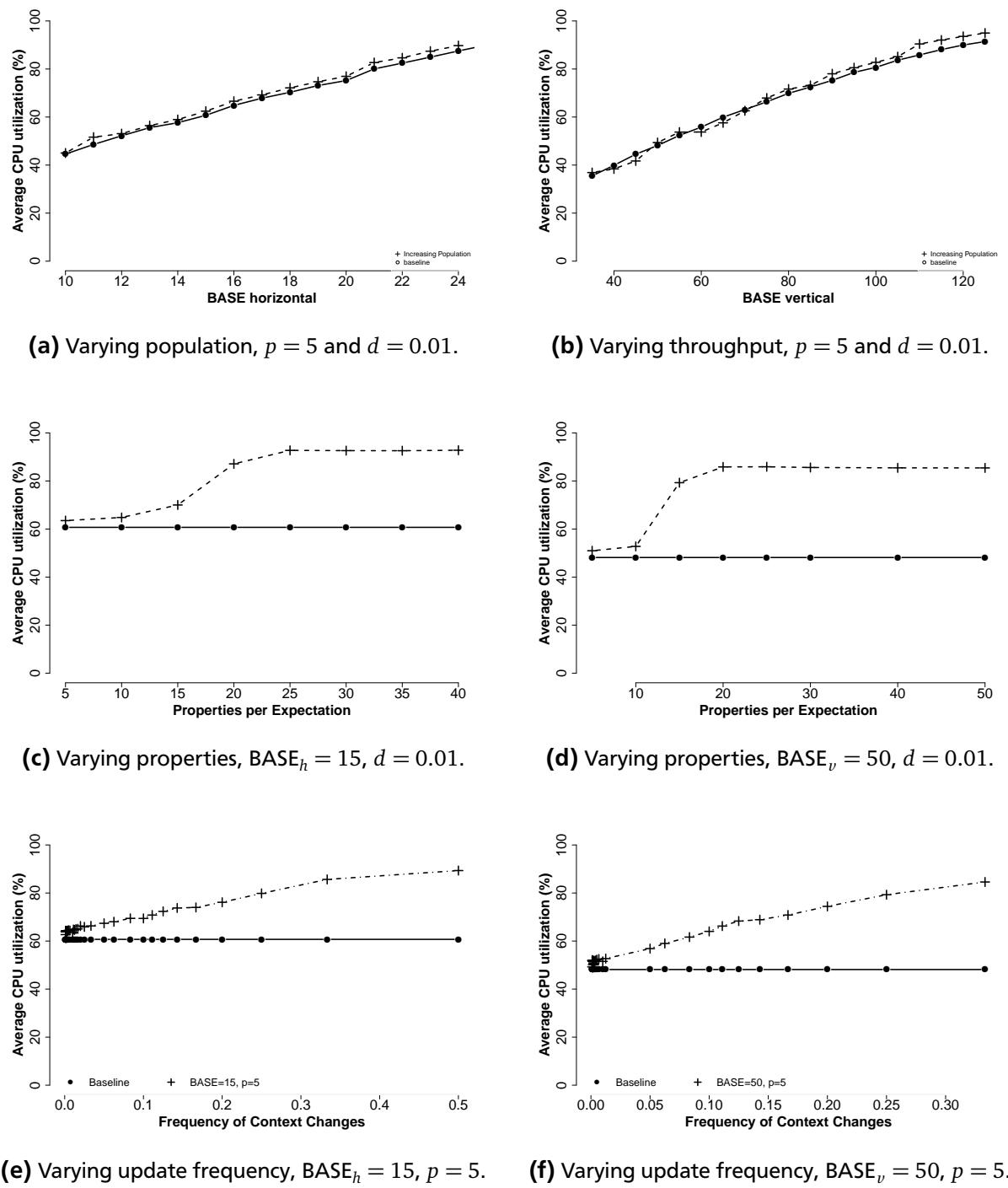


Figure 6.21.: Impact on CPU utilization when varying one parameter ceteris-paribus: horizontal topology (left), vertical topology (right).

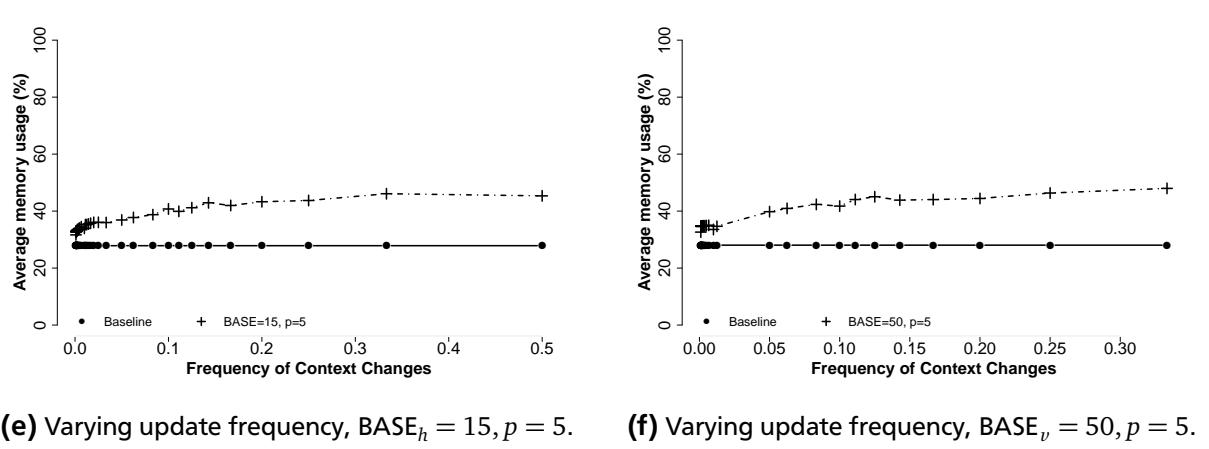
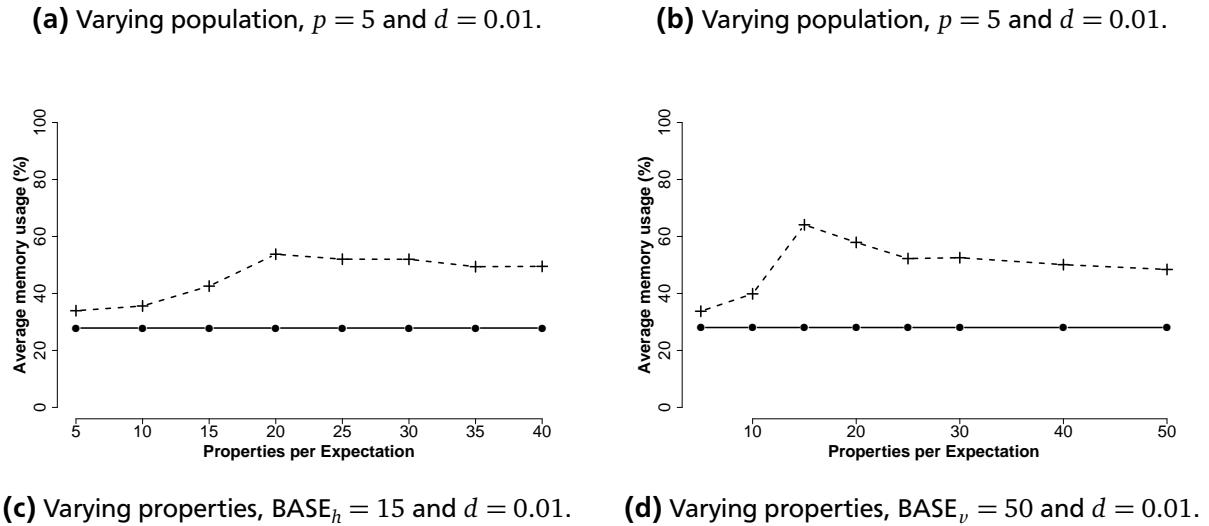
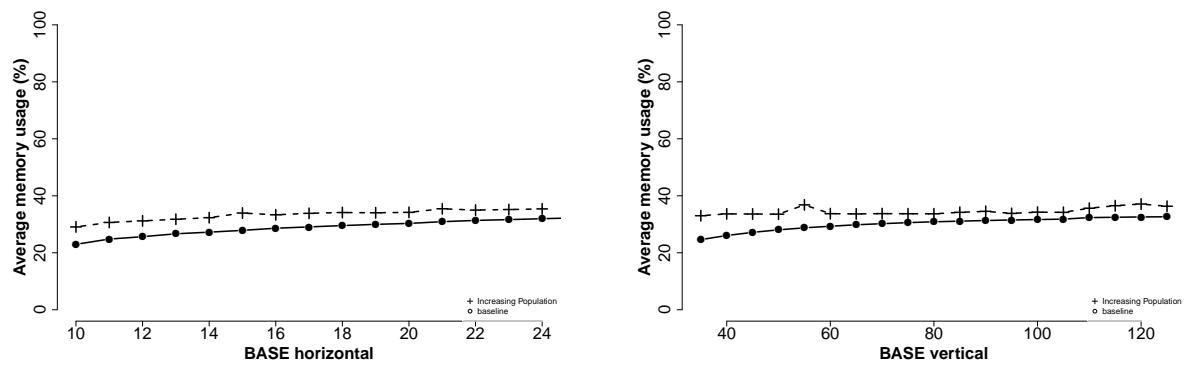


Figure 6.22.: Impact on memory usage when varying one parameter ceteris-paribus: horizontal topology (left), vertical topology (right).

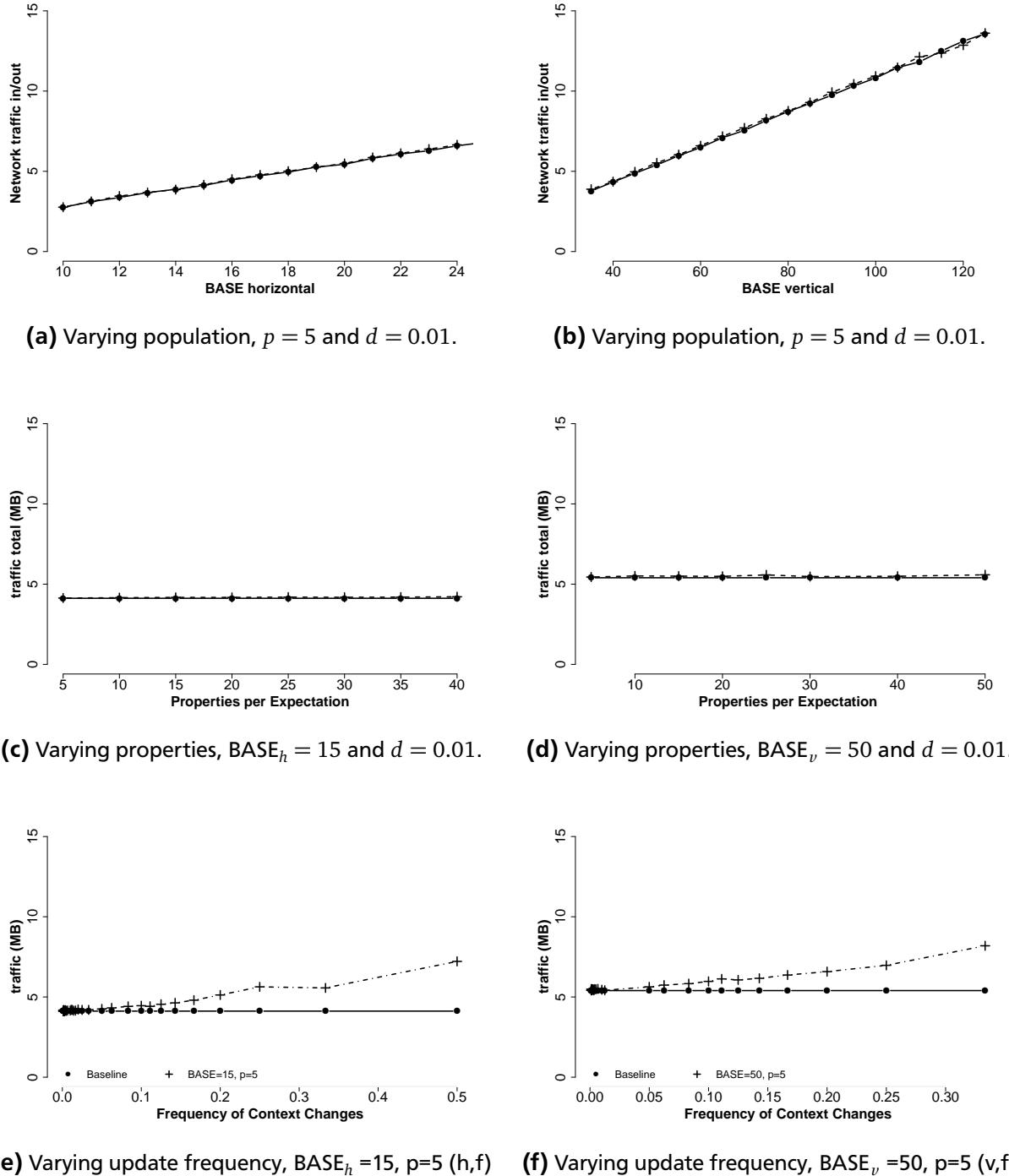


Figure 6.23.: Impact on network traffic when varying one parameter ceteris-paribus: horizontal topology (left), vertical topology (right).

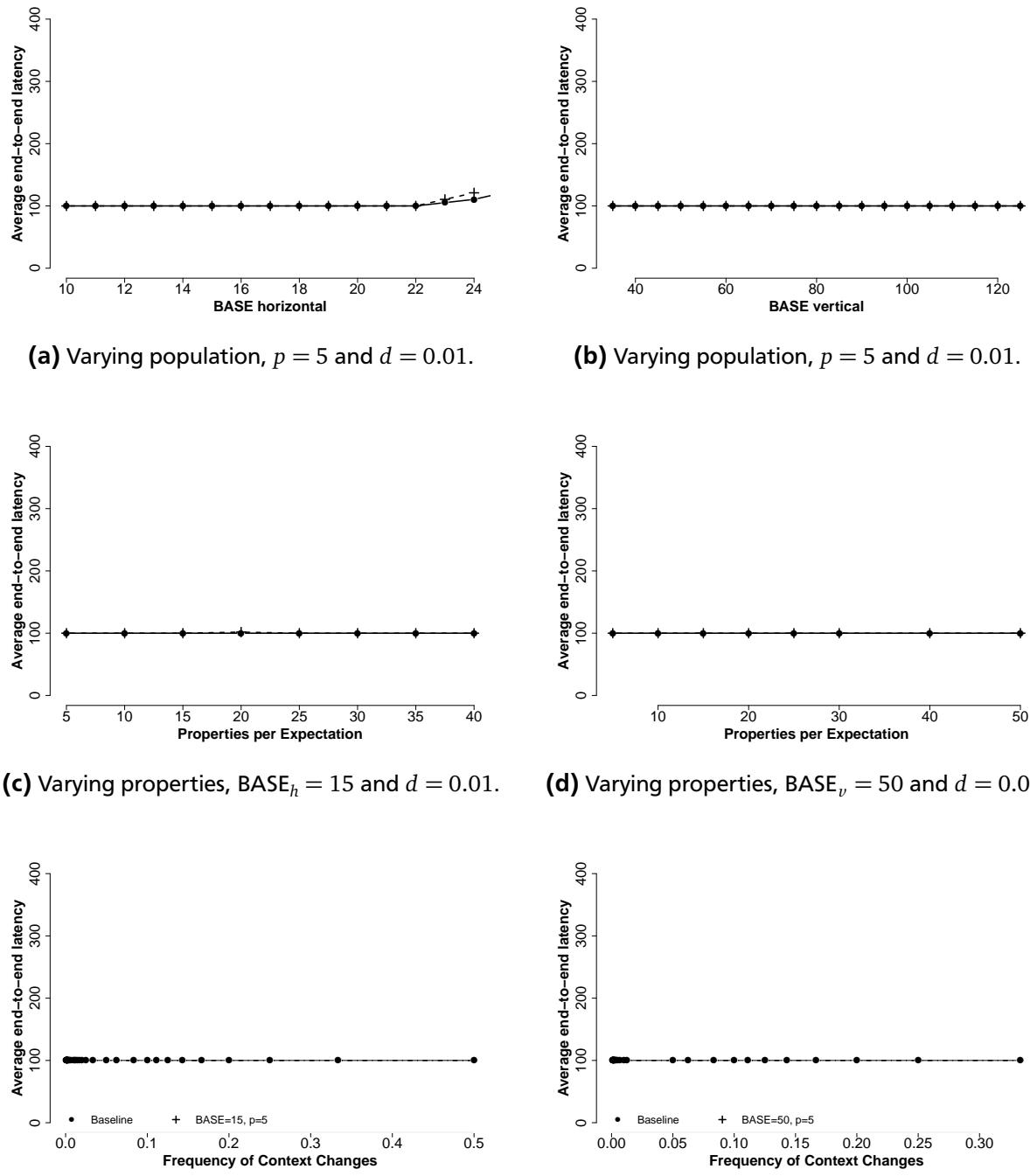


Figure 6.24.: Impact on end-to-end latency when varying one parameter ceteris-paribus: horizontal topology (left), vertical topology (right).

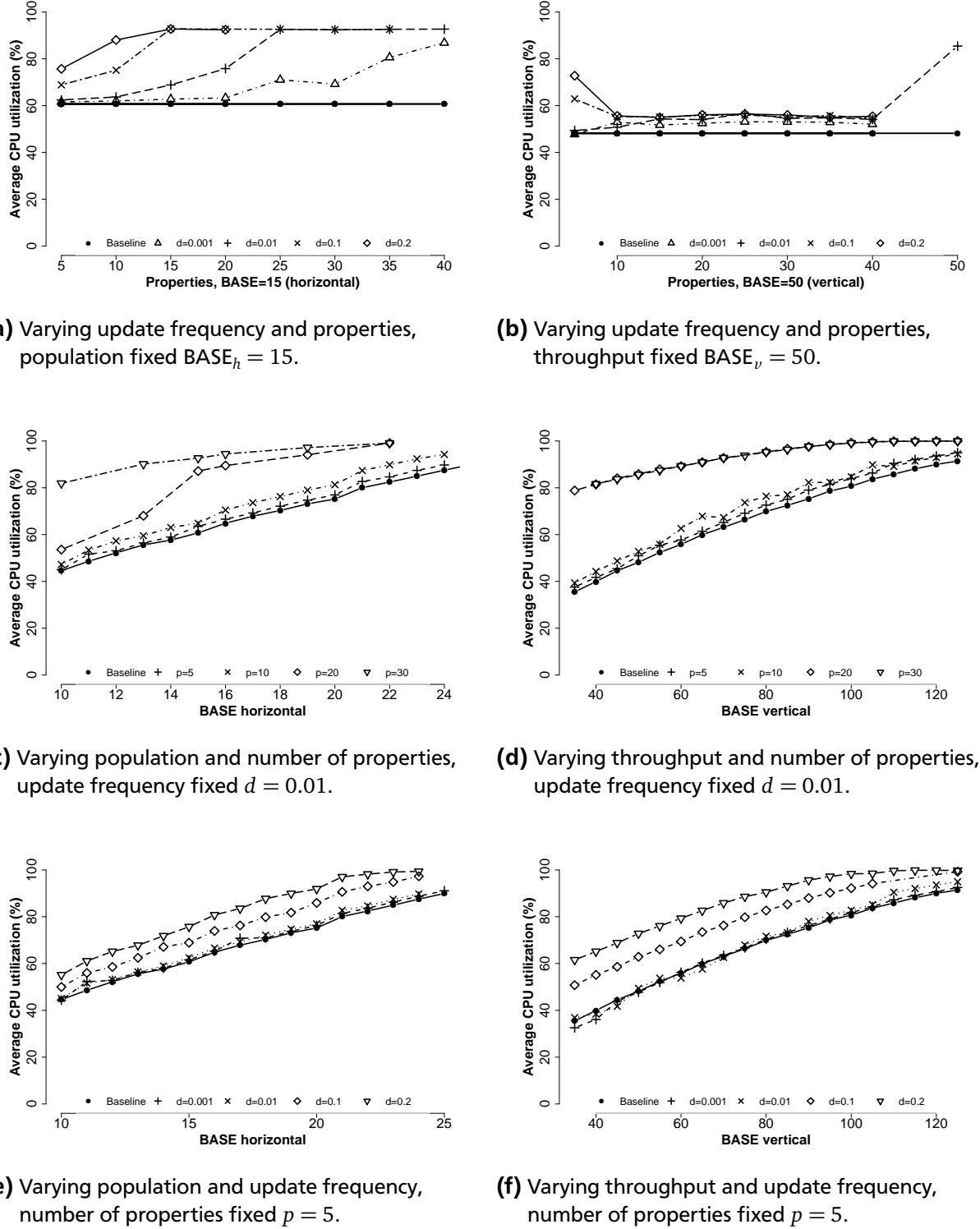
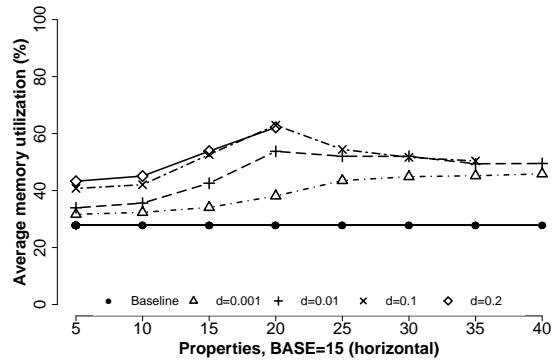
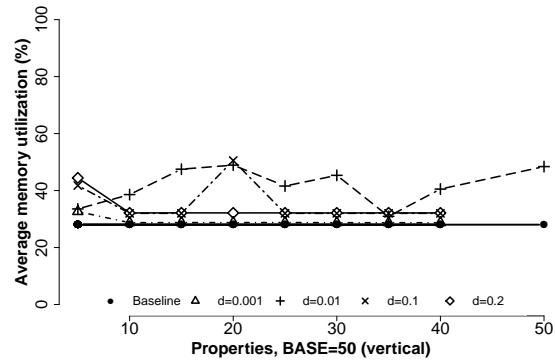


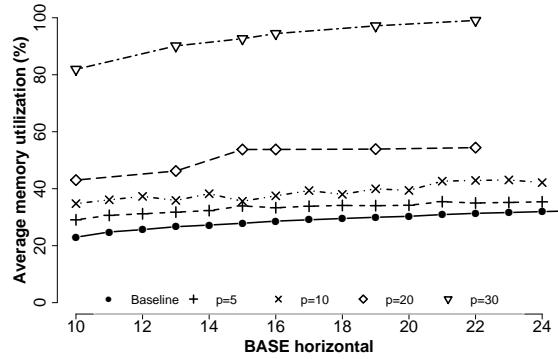
Figure 6.25.: Impact on CPU utilization when varying two parameters ceteris-paribus: horizontal topology (left), vertical topology (right).



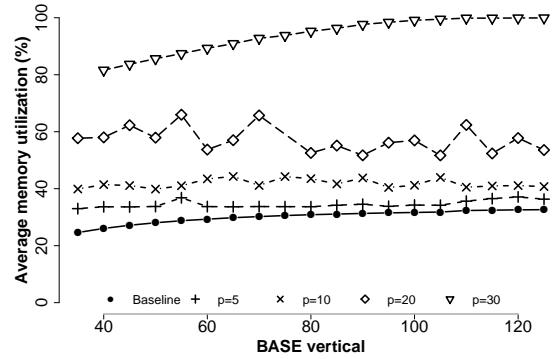
(a) Varying update frequency and number of properties, population fixed $\text{BASE}_h = 15$.



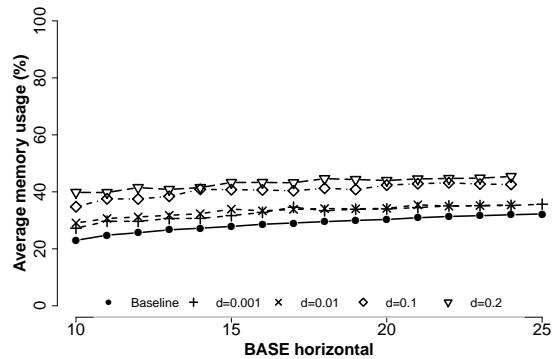
(b) Varying update frequency and number of properties, throughput fixed $\text{BASE}_v = 50$.



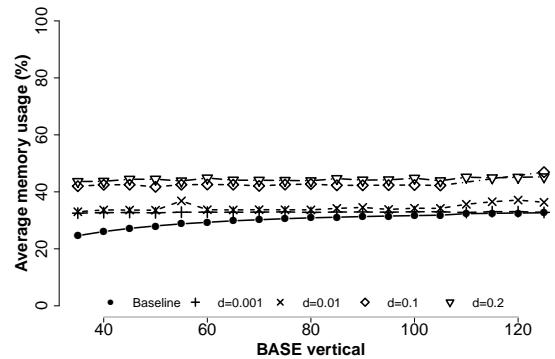
(c) Varying population and number of properties, update frequency fixed $d = 0.01$.



(d) Varying throughput and number of properties, update frequency fixed $d = 0.01$.



(e) Varying population and update frequency, number of properties fixed $p = 5$.



(f) Varying throughput and update frequency, number of properties fixed $p = 5$.

Figure 6.26.: Impact on memory usage when varying two parameter ceteris-paribus: horizontal topology (left), vertical topology (right).

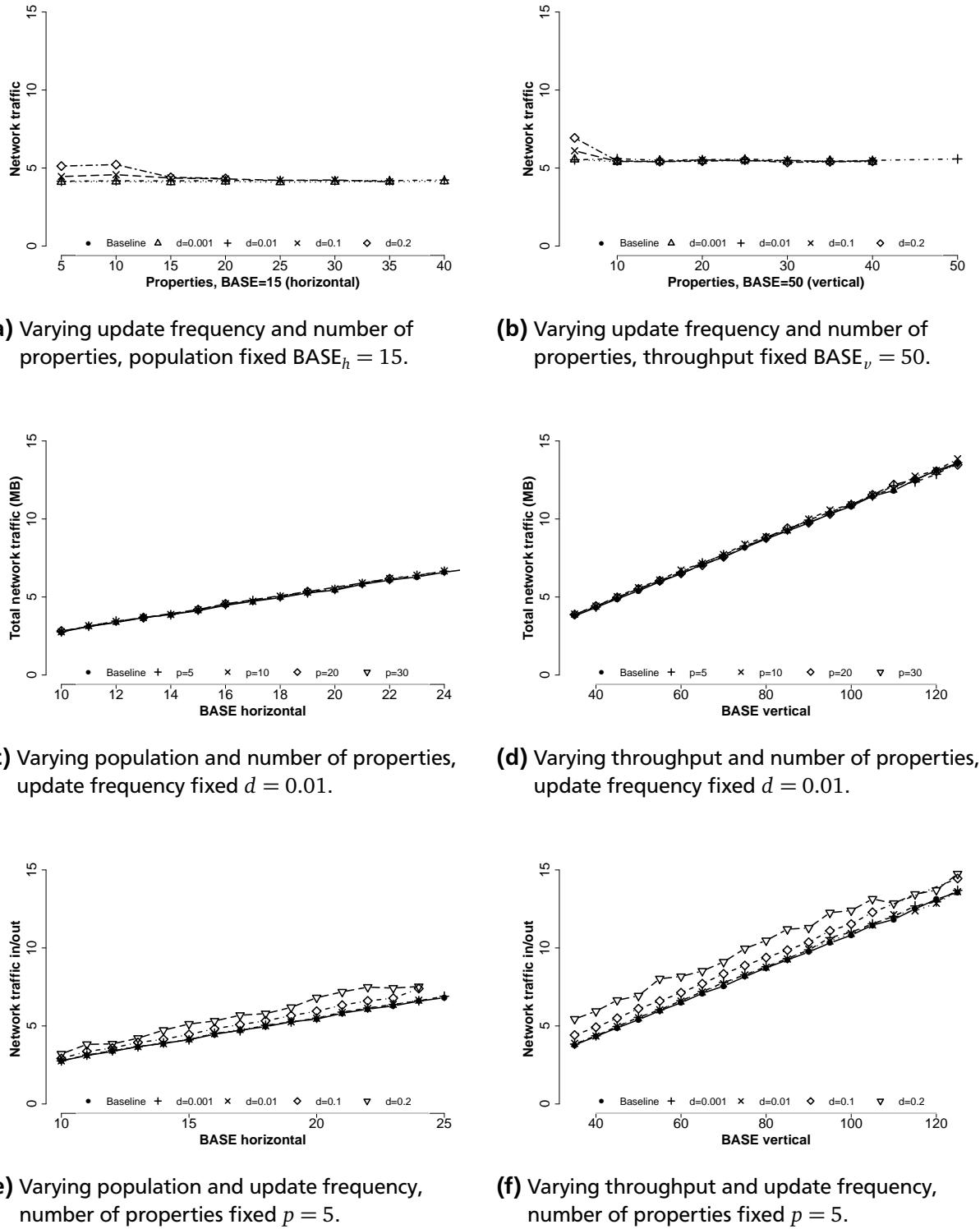
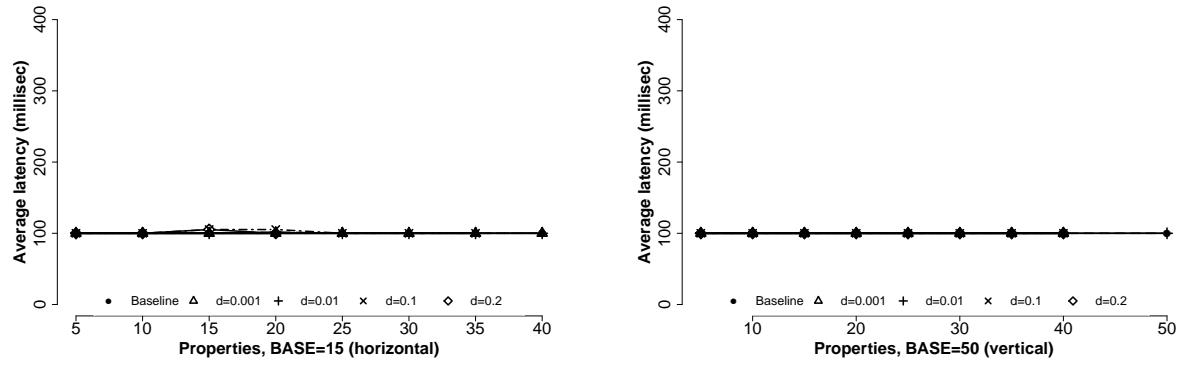
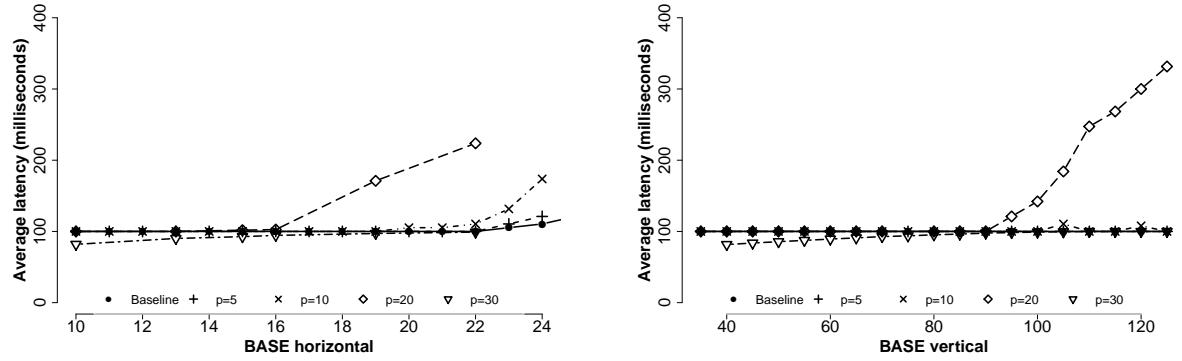


Figure 6.27.: Impact on network traffic when varying two parameter ceteris-paribus: horizontal topology (left), vertical topology (right).



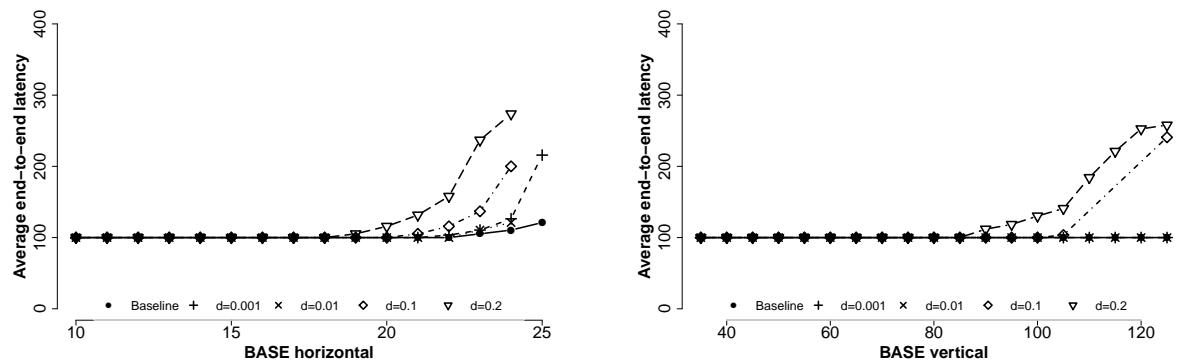
(a) Varying update frequency and number of properties, population fixed $\text{BASE}_h = 15$.

(b) Varying update frequency and number of properties, throughput fixed $\text{BASE}_v = 50$.



(c) Varying population and number of properties, update frequency fixed $d = 0.01$.

(d) Varying throughput and number of properties, update frequency fixed $d = 0.01$.



(e) Varying population and update frequency, number of properties fixed $p = 5$.

(f) Varying throughput and update frequency, number of properties fixed $p = 5$.

Figure 6.28.: Impact on end-to-end latency when varying two parameters: horizontal topology (left), vertical topology (right).

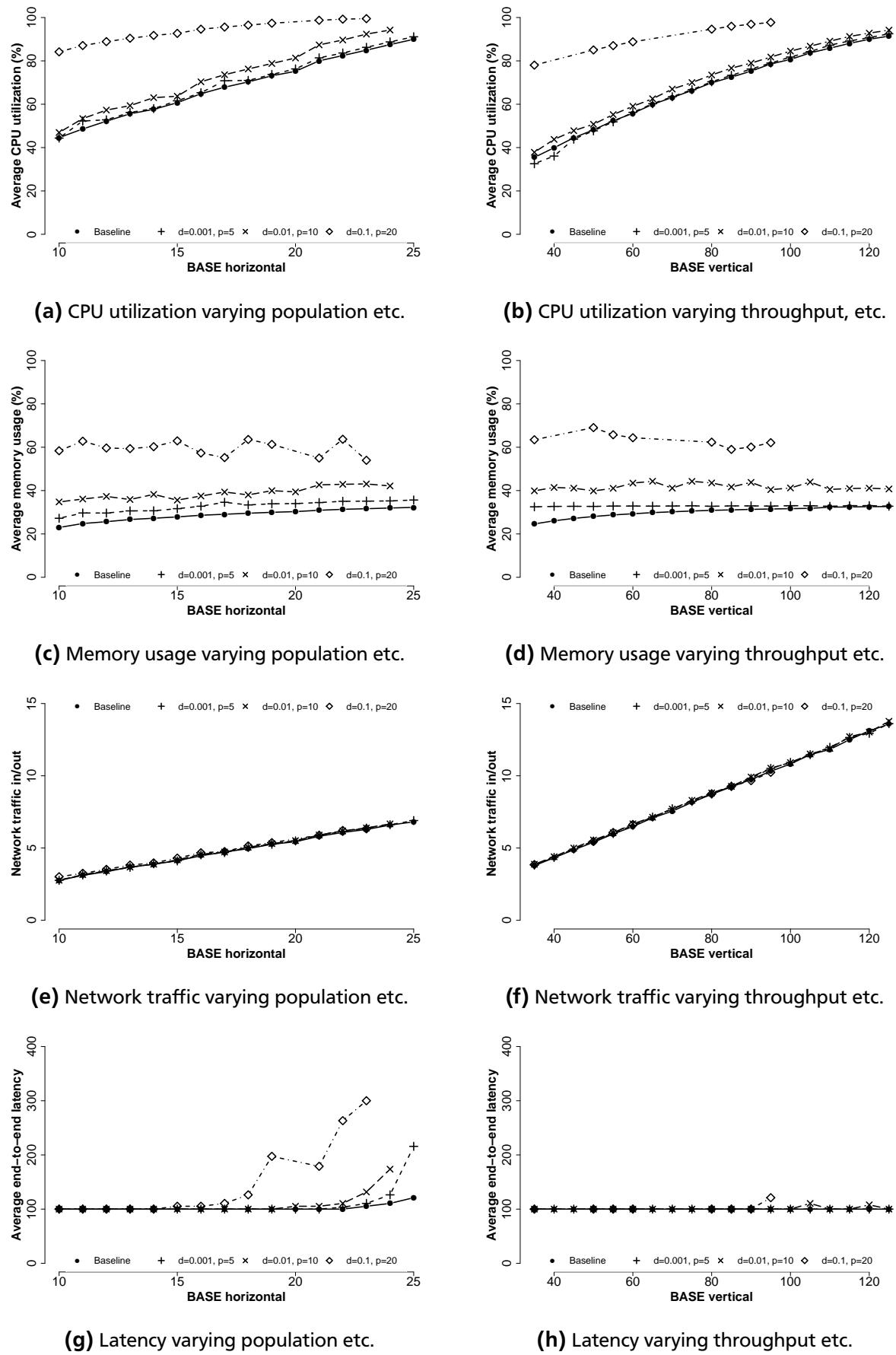


Figure 6.29.: Impact on KPIs when varying all parameters: horizontal (left), vertical (right).

6.4 Effectiveness of Using ASIA to Monitor a DEBS

As part of our approach, we provide participants with aggregated feedback about the population and dynamics of an EBS to aid self-adaptation at runtime. In Section 3.5.2, we have already discussed several aggregated metrics that participants can be updated about at runtime.

In an EBS with a centralized MOM, such as ActiveMQ, these metrics can be provided conveniently at runtime. However, in a DEBS with a distributed MOM, such as REDS, capturing and updating the necessary statistics usually requires the use of an additional monitoring system that is deployed on top of the MOM. This additional monitoring system can be designed as a distributed or a centralized application. Having to deploy and maintain a separate monitoring system, however, increases the execution costs for an EBS that has to deal with large and dynamic populations [172, 177].

Thus, we propose the concept of *application-specific integrated aggregation* (ASIA) we have described in detail in Section 4.5.2 to effectively monitor large-scale DEBS. Instead of adding a separate monitoring overlay, ASIA dynamically integrates monitoring functionality into the broker network, using an approach that is inspired by Aspect-oriented Programming (AOP).

In this section, we quantitatively evaluate the prototype of our monitoring approach implemented within the distributed open-source middleware REDS to show the benefits for participants and MOM in a DEBS and gauge the execution costs in terms of latency and throughput. We chose the academic prototype REDS as it is designed for a distributed setup while ActiveMQ is designed and used mainly for centralized setups.

In Section 6.4.1, we focus on the benefits for participants using our approach to be updated about the system state. We verify that aggregated feedback provided by ASIA always reflects the current system state with the precision that is defined by the requesting participant. We show in particular that the precision of the aggregated feedback is neither affected by the number of brokers making up the MOM nor by workloads that change dynamically over time. Furthermore, we show that participants can balance the frequency of being triggered by updates they have to react to with the precision of the aggregated feedback they want to be updated about.

In Section 6.4.2, we describe the experimental setup that we use for investigating the benefits for brokers – in terms of minimal network traffic overhead – as well as quantifying the execution costs – in terms of throughput and end-to-end latency – for a DEBS applying our approach.

In Section 6.4.3, we focus on the benefits for brokers in terms of network traffic when applying ASIA. We show that our approach generates significantly less overhead in terms of network traffic than a separate monitoring system. In particular, we show that exploiting precision relaxations defined by participants enables brokers applying ASIA to massively reduce the overhead in network traffic compared to a configuration where a separate aggregation system is used.

In Section 6.4.4, we gauge the execution costs in terms of throughput and end-to-end latency for a DEBS using our approach. We show that the benefits provided by applying ASIA do not result in a significant impact on throughput and end-to-end latency of notifications. Rather, integrating the monitoring capabilities into the broker introduces less overhead than using a separate centralized aggregation system.

6.4.1 Benefits: Adjustable Precision

Participants can use aggregated feedback to decide if they should adapt or not. Based on the business logic, context, or state of each participant that requires aggregated feedback, the requirements about the precision of the enclosed information can vary. In any case, however, the participant has to rely on the information to be within those precision boundaries (cf., Section 3.5.2).

Thus, we evaluate if aggregated feedback provided by a distributed MOM applying ASIA does adhere to these precision boundaries for dynamically changing metrics and different numbers of brokers that make up the DEBS.

Based on the default workload provided by FINCoS we have defined a workload pattern for the sampling rate of notifications that changes dynamically over time to different degrees between 25 and 105 notifications per second. Figure 3.28a in Section 3.5.2 shows the workload and its changes over time.

We process the same workload during each run but increase the level of imprecision $\hat{\nu}$ accepted by participants for aggregated feedback about the average sampling rate. In a second test series, we also increase the number of involved brokers from 1 to 15 brokers. Each single run is repeated ten times.

FINCoS is used as a workload-generator and has been extended to request and receive updates on aggregated feedback provided with ASIA. Details on the extensions can be found in Appendix A.4. For each single run we measure:

1. the degree to which aggregated feedback received by subscribers varies from the real system state measured by the maximum difference between either the lower or the upper bound of the interval reported by an ASIA aggregation (cf., Figure 3.27a); and
2. the number of notifications containing aggregated feedback that have to be processed by the receiving participant for each level of imprecision.

The measured results are shown in Figure 6.30a as boxplots. Each boxplot shows the results measured for a given level of imprecision when using 1, 5, 10 or 15 brokers to connect subscribers and publishers. The red dashed lines denote the maximum deviation that would be tolerated for a given level of imprecision. The results show that aggregated feedback about the average sampling rate provided by our approach does always adhere to the precision requirements set by participants; the number of brokers involved in providing this aggregated feedback does not have an impact on the precision either.

For the same experiments, we have measured the number of updates sent to participants to represent the system state. The results are shown in Figure 6.30b as the percentage of notifications with piggybacked aggregated feedback that has to be processed by a participant in proportion to the total number of notifications processed for the same workload. A low imprecision of 1 results in 46% of the total number of processed notifications to be about aggregated feedback; an imprecision of 5 already reduces this to 9% while the number of necessary updates drops to less than 1% for an imprecision of 10 (0.49%), 25 (0.18%) or 50 (0.06%).

In conclusion, participants are triggered less often about changes to the system state if they do not care about minute changes to that metric and express this to the MOM by an increased imprecision. This allows them to free up resources, as the received feedback has to be evaluated

less often and only if the system state has changed in an order of magnitude that is significant to the participant. Consequently, the resource utilization can be adjusted to the level of detail required by the participant.

6.4.2 Experiment Setup for Gauging Traffic, Throughput and Latency

In the previous section we have shown that participants can rely on the aggregated feedback provided by a distributed MOM applying ASIA to be precise. Providing this information, however, requires a continuous synchronization between brokers in a DEBS. Thus, we evaluate the overhead incurred by providing aggregated feedback on the KPIs of a DEBS.

The workload used for this evaluation is inspired by an SCM scenario like the one used by SPECjms2007 and discussed in Section 6.3.1. The objectives are: distributed topology, ranges of aggregations, high load, dynamic population.

Consider a global logistics operation being supported by an Event-driven Architecture (EDA) [170, 176]. Shipping containers (e.g., with the help of RFID) publish position updates, while analytical applications, headquarters (HQs), supermarkets (SMs), and distribution centers (DC) subscribe to notifications about the status of a shipping, the position of containers, or other aspects of the intertwined business processes.

In our setup, 16 brokers form a DEBS representing DCs at waypoints for shipping containers along a supply chain. The broker in each DC is connected to that of 3 other DCs and handles 100 directly connected publishers and subscribers. Each subscriber is subscribed to 3 from 100 potential types of notifications, and each publisher publishes at an average rate of 1 notification per second to emulate a sequence of position updates while moving. Subscriptions filter on their type alone, with each subscriber receiving on average about 3% of the published notifications. Advertisements and subscriptions are uniformly distributed over all types of notifications. Every 10 seconds a subscriber issues a subscription or an unsubscription to mimic containers passing waypoints (subscribers and publishers are handled by the DC next to them).

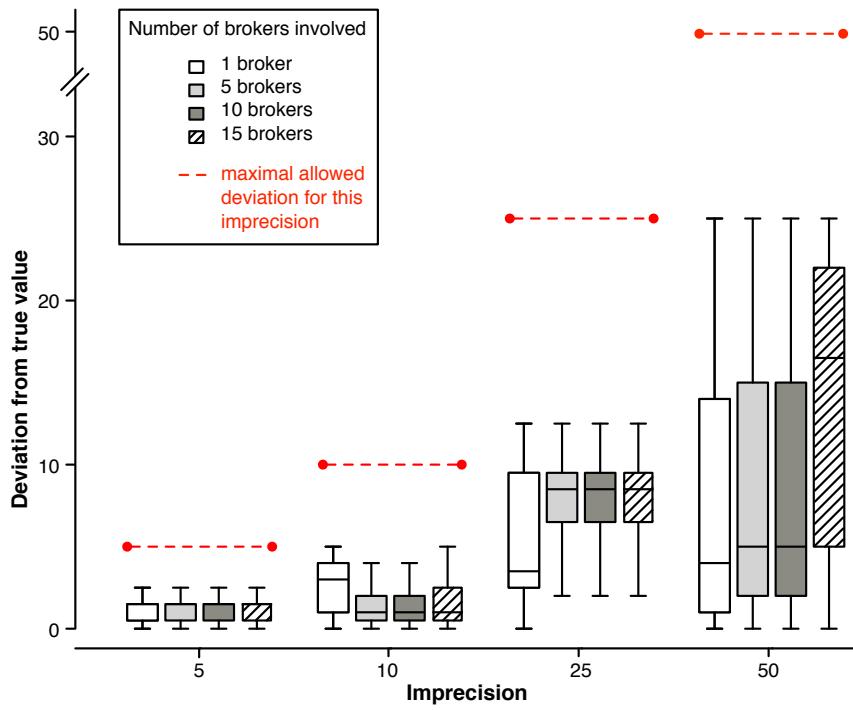
All experiments have been repeated five times with different seeds for generating notifications and subscriptions. Each data point in our figures represents the average value measured. The topology is randomly created for each repetition but the same random topology is used for all configurations that have to be compared for each experiment. Further details on the experimental setup are described in Appendix A.8.

During the evaluation, we consider four different aggregation functions: two *population* count functions (`subscriberCount` and `publisherCount`) and two *dynamics* rate functions (`subscriptionRate` and `publicationRate`).

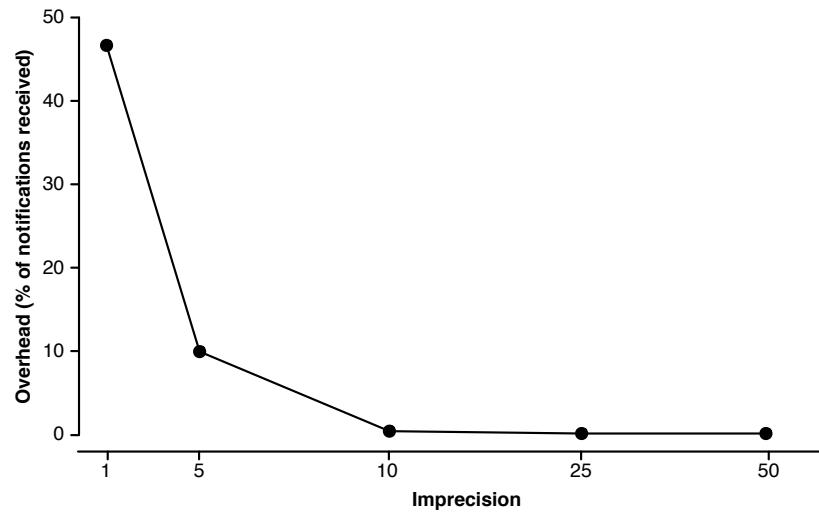
6.4.3 Impact on Network Traffic

In this section, we investigate how integrating the monitoring functionality provided by ASIA into the brokers of a DEBS affects the network traffic as the state of aggregations has to be synchronized between brokers. We demonstrate that ASIA can provide additional aggregated feedback about the system state with a limited traffic overhead.

Using the workload as described in Section 6.4.2, we explore how the resulting network traffic is affected when varying the maximum imprecision allowed for aggregated feedback.



(a) Precision of aggregated feedback stays within defined boundaries.



(b) Relaxed precision requirements result in less updates for participants.

Figure 6.30.: Aggregated feedback provided by ASIA always adheres to the precision requirements defined by participants while allowing them to define trade-offs between the precision of aggregations and the number of updates they receive.

For this, we measure the network traffic generated in our scenario in terms of the average number of notifications processed per second. The network traffic between subscribers, publishers, and brokers includes the additional traffic generated by aggregation updates that cannot be piggybacked but are required for synchronization between brokers or notifying participants. When using the `subscriberCount` and `subscriptionRate`, we also include the traffic generated by subscriptions: in ASIA, they are processed only once and used to piggyback updates, while an independent aggregation system evaluates them separately to compute aggregates. We support `publisherCounts` and `publicationRates` analogously.

We compare the network traffic generated by a DEBS applying ASIA with the traffic generated in a configuration where a separate distributed aggregation system is deployed on top of the broker network (DEBS+Agg). As a baseline, we compare the measured network traffic with the network traffic generated in an EBS without any support for aggregated feedback (DEBS). All configurations are implemented based on REDS with the separate aggregation system being designed similar to the *network imprecision (NI)* approach introduced by Jain et al. [241] so that it is able to handle imprecision.

Each participant requests aggregated feedback for three types of notifications that are chosen randomly for a single experiment. The maximum imprecision defined by participants is set to 0 to stress the system to the maximum degree as updates have to be delivered immediately even for minor changes to the true value under observation. The topology is randomly generated for each run.

Figure 6.31 shows the measured average number of notifications processed per second when changing the maximum imprecision accepted by participants. In each graph, the x-axis represents the maximum imprecision allowed by participants. The DEBS configuration represents our baseline: as it does not perform aggregation, the network traffic measured in this configuration is independent of the level of imprecision allowed by participants. Conversely, as expected, the network traffic decreases with an increasing imprecision for both ASIA and DEBS+Agg.

Comparing a DEBS applying ASIA to a configuration where the DEBS uses a separate distributed aggregation system shows that ASIA produces significantly less traffic than DEBS+Agg in all considered experiments. This advantage increases with the level of imprecision, becoming especially visible for dynamic rate aggregates as shown in Figures 6.31b and 6.31d. Here, the overhead introduced by ASIA drops much faster compared to the configuration that uses a separate aggregation system (DEBS+Agg). The main reason for this is that ASIA reduces network traffic by piggybacking aggregation updates to the messages exchanged between brokers, whenever possible. As a separate aggregation system does not have access to the broker state to piggyback information on the original notifications, separate notifications have to be sent at all times.

Comparing ASIA to the baseline configuration (DEBS) where the DEBS does not provide aggregated feedback shows a relatively small increase in network traffic even when participants request precise feedback. For example, when considering an imprecision of 20 publications/subscriptions every 5 seconds, we observe an overhead of only 9% for `subscriptionRate` provided by ASIA compared to 225% overhead for DEBS+Agg (cf., Figure 6.31b); for feedback about the global sampling rate, we observe 20% overhead for ASIA compared to 104% overhead for DEBS+Agg (cf., Figure 6.31d).

Analyzing the results for feedback about the population with an imprecision of 20 participants over 1600, we measure the following overhead for ASIA: 82% for `subscriberCount` (vs. 465% in

DEBS+Agg) and 110% for publisherCount aggregators (vs. 257% in DEBS+Agg) as shown in Figures 6.31a and 6.31c.

Furthermore, the overhead introduced by ASIA drops below 20% in all experiments when increasing the level of imprecision to 100. An increase in imprecision allows ASIA to fully exploit the combination of piggybacking and relaxed precision requirements.

6.4.4 Execution Costs: Throughput and Latency

We gauge the execution costs of our approach by measuring how integrating ASIA into the broker network affects end-to-end latency and throughput. Our results demonstrate that ASIA can offer aggregation support without significantly impacting both KPIs.

Throughput

The first experiment explores whether the aggregation process creates a bottleneck in the system. We compare the throughput of a DEBS applying ASIA against a typical DEBS that does not provide aggregated feedback. This configuration (DEBS) is also implemented in REDS using the same routing protocol as our implementation of ASIA that we compare against.

We increase the sampling rate of publishers while measuring the received rate at subscribers to measure the maximum throughput. Each participant requests aggregated feedback about four different aggregations: subscriberCount, publisherCount, subscriptionRate, and samplingRate. Each participant requests those aggregations for different types of notifications that are randomly selected from all available types.

Figure 6.32a shows our results. For both the typical DEBS system without support for aggregated feedback (DEBS) and for ASIA, the throughput initially grows with the sampling rate. If we compare DEBS and ASIA, we observe that the throughput saturates at the same sampling rate per publisher and that the maximum throughput value is almost identical for both configurations: brokers become saturated as incoming notifications at a broker are queued before processing, and the throughput stabilizes. This indicates that ASIA does not introduce a noticeable overhead even under an extreme workload when each client requests feedback about different aggregations, which requires the computation of different aggregation functions.

Latency

We gauge the impact ASIA has on the latency for notifications and update propagation for aggregated feedback in a separate set of experiments. We configure participants to request aggregated feedback about two different aggregation functions: publisherCount and samplingRate. These aggregation functions are the most challenging ones for a DEBS applying ASIA as the observed system state changes rapidly and update information has to be continuously synchronized between brokers. While piggybacking update information on top of notifications whenever possible enables ASIA to minimize the overhead in terms of traffic, this mechanism has a higher likelihood of causing queuing at intermediate brokers: information has to be piggybacked at the sending broker while the receiving broker has to check every notification for additional metadata that might have to be retrieved.

We compare a configuration using ASIA with two configurations using a separate monitoring system for computing and updating the aggregated metrics: one configuration where the DEBS

uses a *distributed* aggregation system (DEBS+Agg) and one where the DEBS uses a *centralized* aggregation system (DEBS+AggC). In the centralized configuration DEBS+AggC, all aggregates are computed by a single broker that also sends updates to participants whenever necessary. All three configurations use the same overlay topology and the same protocol for distributing advertisements, subscriptions, and notifications.

For aggregation updates, we maximize the additional load on the aggregation components by setting the imprecision to 0, requiring all updates to be delivered immediately to all interested participants. All publishers publish at a sampling rate of 100 notifications per second.

Figure 6.32b shows the measured values for end-to-end latency of notifications. The latency of notifications processed by a MOM applying ASIA is comparable to the latency in a system that uses a separate aggregation system (EBS+Agg and EBS+Agg Centr), both in terms of average delay and in terms of 95th percentile.

Considering the end-to-end latency for aggregation updates as shown in Figures 6.32c and 6.32d), we notice how the topology significantly impacts the results if a separate aggregation system is used. There is a significant difference depending on whether the DEBS uses a centralized or a distributed aggregation system. When considering a centralized aggregation system, the end-to-end latency is up to 300% higher. This is because all information has to be delivered to a central node first where the aggregations are computed and updates distributed to participants. The resulting overhead highlights the limitation of such a centralized approach as the reported feedback lags behind the real system state.

Comparing ASIA with DEBS+Agg that uses the same topology shows that ASIA adds a moderate latency overhead of up to 30% in the 95th percentile. As mentioned before, this is due to the piggybacking of aggregation updates to notifications and could be removed by defining a separate queue for processing notifications with updates in the broker so that piggybacking and retrieving metadata does not affect the routing of other notifications. Still, we observe virtually no overhead when comparing the average latency measured for ASIA and DEBS+Agg.

We have also monitored the resource utilization of brokers in terms of CPU utilization and memory usage. However, REDS is a lightweight academic prototype not built for high performance. Thus, a broker saturates before any significant impact on the resource utilization of its host machine could be measured. The average CPU utilization of a broker during our ASIA experiments is about 2% when using an EBS with ASIA and does not significantly change with the imprecision used; the average CPU utilization for a broker running DEBS+Agg is 3%.

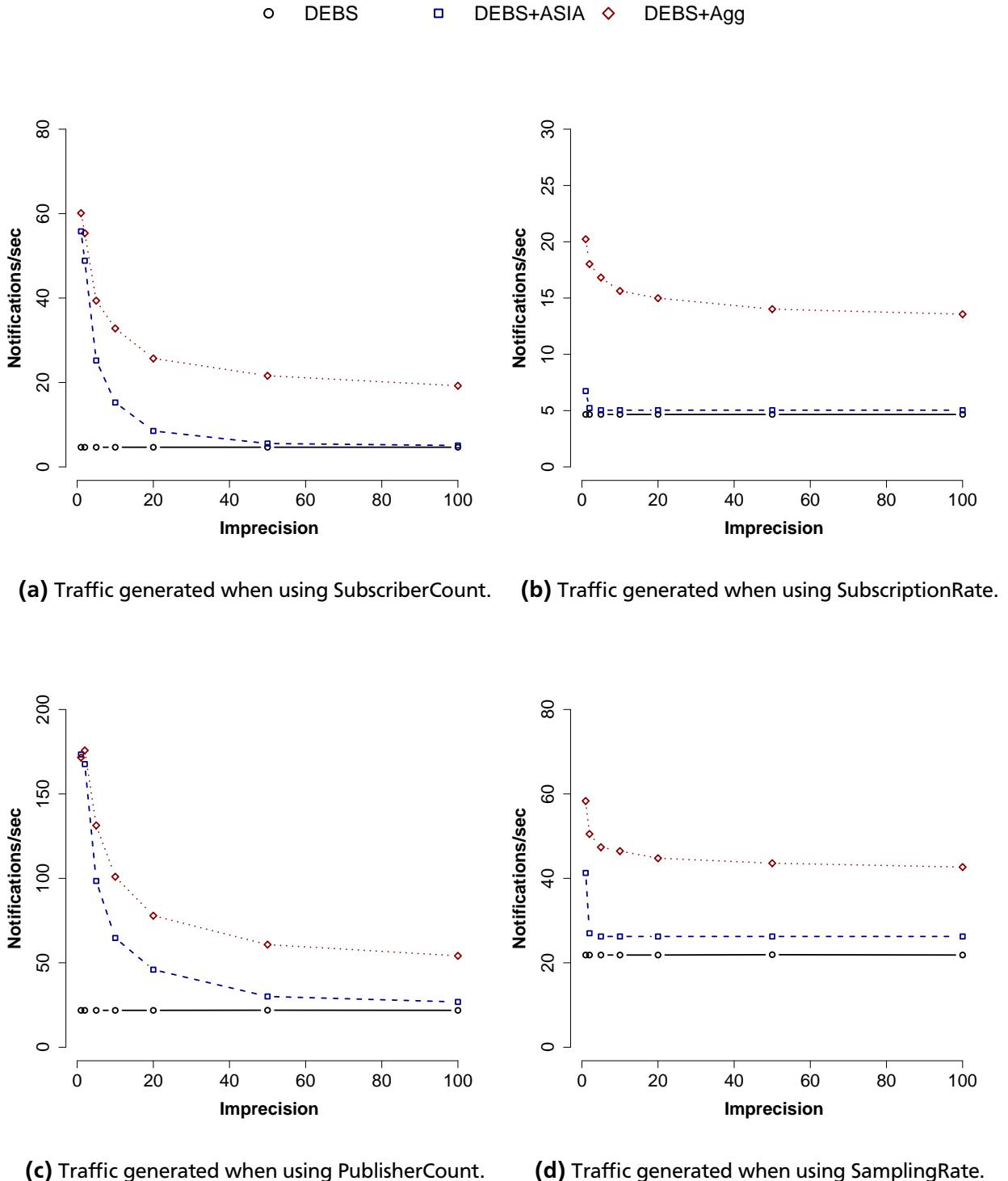
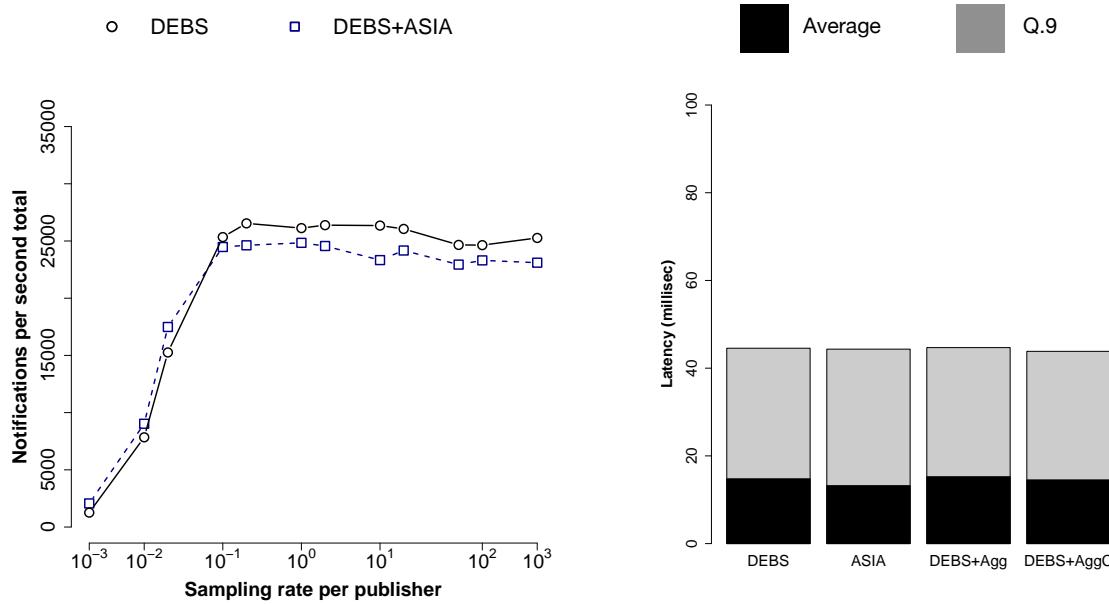
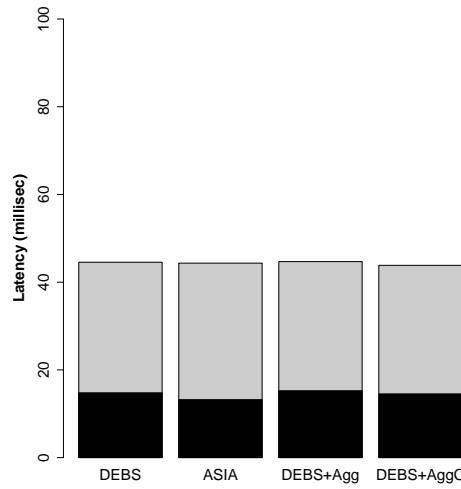


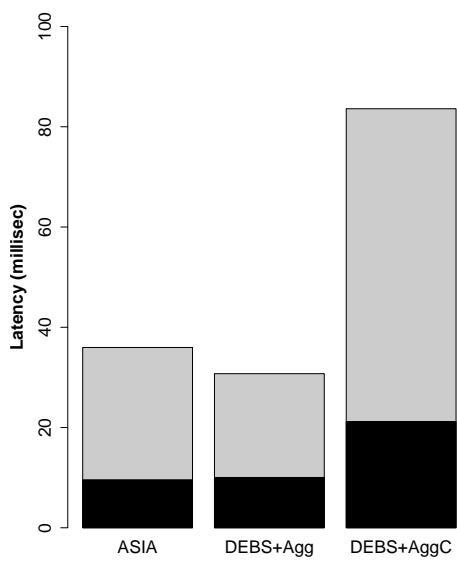
Figure 6.31.: Network traffic generated by ASIA and a DEBS with an additional distributed aggregation system (DEBS+Agg) when using different aggregation functions; network traffic in a DEBS without support for aggregated feedback (DEBS) for comparison.



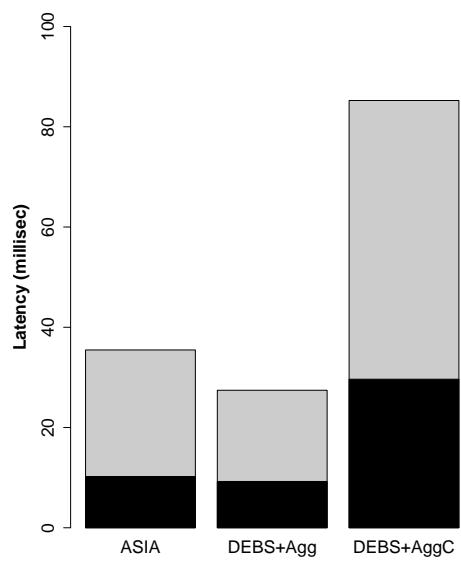
(a) Throughput vs. total sampling rate.



(b) Latency for notifications.



(c) Latency for samplingRate updates.



(d) Latency for publisherCount updates.

Figure 6.32.: Measured throughput and latency to gauge execution costs. Comparing throughput and end-to-end latency of a DEBS applying our approach (ASIA) with a typical DEBS without support for aggregated feedback (DEBS), a DEBS using a separate *distributed* aggregation system (DEBS+Agg), as well as a DEBS that uses a *centralized* aggregation system (DEBS+AggC).

6.5 Summary

In this chapter, we have qualitatively evaluated the expressiveness of our model to support QoI requirements in an EBS; we have quantitatively evaluated the benefits of our approach as well as the execution costs for our prototypes built on ActiveMQ and REDS.

In Section 6.1, we have explored the expressiveness of our model. We have shown that requirements about QoI in an EBS can be expressed and supported using expectations and capabilities in a more expressive manner than in related approaches. Subscribers can express complex preferences using expectations and capabilities, while publishers can expose their execution and production costs at runtime; the MOM utilizes this information to dynamically decide on suitable reactions to support arbitrary quality-related properties in an EBS.

In Section 6.2, we have identified and quantified the benefits for participants when the EBS applies our approach compared to a typical EBS. Based on a scenario where multiple publishers provide the same type of information with varying QoI properties, we have shown that our approach provides subscribers with data that always conforms to their QoI requirements. Using the fidelity metric (cf., Section 3.3.3) to quantify the degree of conformance between requirements and the delivered data, we have shown, that the fidelity generated by our approach is constantly higher than in configurations where the EBS applies only features used by related approaches. For subscribers, the MOM, and publishers we have measured significant resource savings in terms CPU utilization and network traffic when applying our approach compared to a typical EBS.

In Section 6.3, we have analyzed the execution costs arising at the MOM when applying our approach. Using the industry-strength jms2009-PS benchmark, we have investigated the overhead introduced by our prototype compared to a bare ActiveMQ deployment. We have shown that the CPU of a single ActiveMQ broker becomes the limiting factor when supporting QoI requirements of large and very dynamic populations that use a huge variety of generic properties. We have shown that the overhead introduced by negotiating expectations and capabilities linearly increases with an increasing number of generic properties and update frequency that participants update their quality-related requirements or capabilities at runtime with; both cost drivers get amplified linearly by the size of the population that the MOM is catering to. The results of our experiments show that these cost drivers can be traded-off against each other: supporting large populations or high throughput requires either lower dynamics or smaller sets of properties to negotiate and vice-versa. These interdependencies can be leveraged by scaling strategies such as using a distributed network of brokers instead of a centralized setup with a single broker to distribute the load across multiple brokers. Nonetheless, a single instance of our prototype can support QoI requirements about the same number of properties we have discussed in Section 6.1 even if more than 2000 participants update their requirements or capabilities at the same time at least every 1000 seconds.

In Section 6.4, we have quantitatively evaluated our approach to monitor the population and dynamics of a DEBS at runtime to provide participants with aggregated feedback. Using our prototype implemented on the distributed MOM REDS, we have shown that our approach of application-specific integrated aggregation (ASIA) provides precise monitoring information without significantly impacting the throughput, end-to-end latency or resource utilization of a DEBS. Our measurements show that our approach outperforms separate monitoring systems in terms of traffic-overhead and end-to-end latency while catering to 1600 participants. Moreover, we have shown that participants can express relaxations in their precision requirements about aggregated feedback and that both participants and the MOM benefit from exploiting these relaxations.

7 Related work

The concept of expectations, capabilities and feedback presented in this dissertation describes a novel approach to support requirements about Quality of Information (QoI) in Event-based Systems (EBSs) at runtime. The approach synthesizes and expands several existing concepts and approaches devised in pull- and push-based systems.

Throughout this dissertation, we have already addressed related work for specific domains, aspects or concepts. In Chapter 2, we have introduced the basic pull-and push-based paradigms and concepts addressed in this dissertation. We have pointed out related work about EBSs, Wireless Sensor Networks (WSNs), Cyber-physical Systems (CPSs), Data Stream Management System (DSMS), System-of-Systems (SoS), Service-oriented Architectures (SOAs) and Cloud computing. In Chapter 3, we have surveyed more than 70 peer-reviewed publications about these concepts to derive a generic model for representing and processing quality-related properties in our approach. In Chapter 4, we have pointed out related work about Multi-Objective Optimization Problems (MOOPs) and performance models. In Chapter 6, we have investigated the expressiveness of related approaches in detail.

In this chapter, we focus on the state-of-the-art directly related to the contributions of this dissertation. As shown in Figure 7.1, we discuss standards and Message-oriented Middleware (MOM) solutions that support requirements about quality-related properties before reviewing related work regarding monitoring, self-adaptation, and negotiation in pull- and push-based systems.

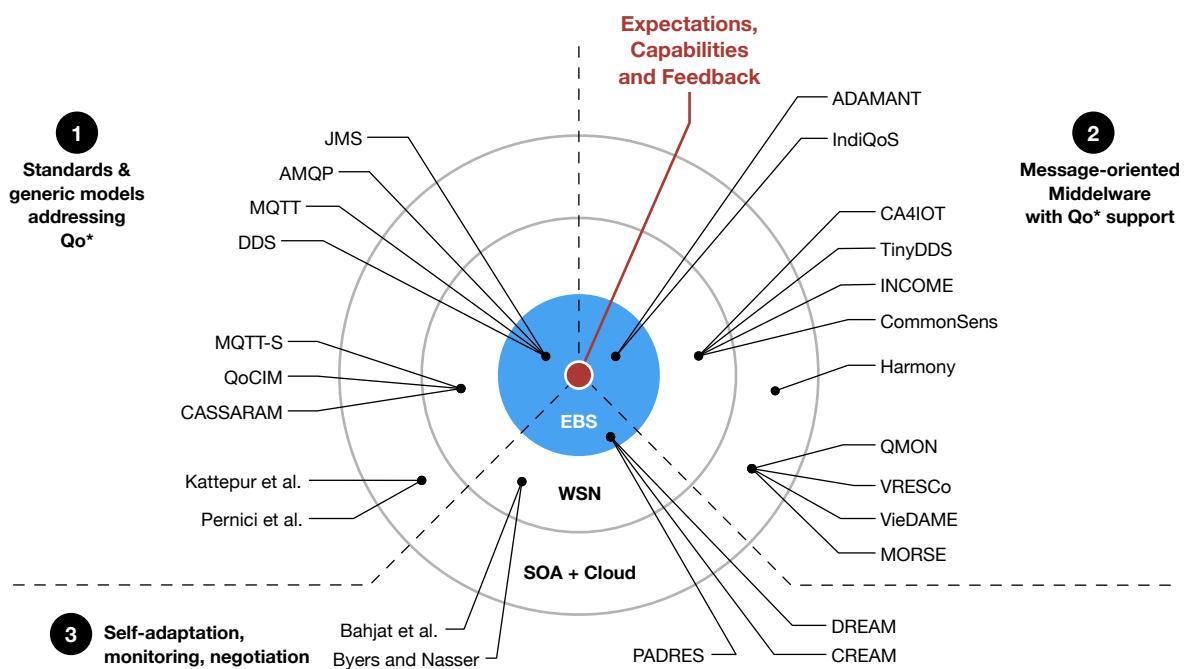


Figure 7.1.: Chapter overview: related work is structured based on the contributions.

We have structured the presentation of these topics according to the research areas each work originates from, reflecting the structure of our background Chapter 2: EBS and WSN for push-based approaches as well as Cloud computing and SOA for pull-based approaches.

7.1 Standards and Protocols for Asynchronous Communication

Several standards and protocols for push-based communication have been proposed over time. Amongst those that have attracted interest, we discuss the Java Message Service (JMS), Advanced Message Queuing Protocol (AMQP), MQ Telemetry Transport (MQTT), and Data Distribution Service (DDS) regarding their support for quality-related properties.

JMS is the widely used standard for industry-strength systems applying an asynchronous communication model. It defines an Application Programming Interface (API) for push-based communication between participants and a MOM written in Java [131, 404]. Participants can use any wire protocol to communicate. Implementing the JMS API, a MOM provides support for certain quality-related properties that focus on transactional behavior as well as persistence and prioritization on a per-notification basis. Please note that JMS does not provide any mechanism for feedback. As only publishers can set priorities for notifications, this mechanism cannot be used to support requirements of subscribers for privileged processing of certain notifications. [17, 138].

AMQP is a wire protocol for push-based applications that focus on high-performance [260, 417]. It has been originating from the financial services industry and is specified by the Organization for the Advancement of Structured Information Standards (OASIS) with the aim of easing interoperability between MOMs of different vendors. On the network transport level, AMQP supports the set of Quality of Service (QoS) properties provided by JMS together with additional guarantees that aim at reliably delivering notifications — in particular persistent or non-persistent delivery, notification priorities, expiration date, and encryption [17, 326, 328].

MQTT is a lightweight transport protocol based on TCP/IP developed for push-based Internet of Things (IoT) applications that use Publish/Subscribe (PS). MQTT has been specified by OASIS and supports QoS requirements about delivery guarantees "best effort", "at most once", "at least once", "exactly once" on the network protocol level [327]. MQTT-S is an adaptation of the MQTT transport protocol to be used in resource-constrained WSNs and CPSs that want to use a topic-based PS communication model [230].

DDS is a messaging protocol that has emerged from the domain of embedded real-time systems. The combination of wire protocol and API has been standardized by the Object Management Group (OMG) and is currently supported by several MOMs such as TinyDDS [65] or Adamant. We have already introduced the core components of DDS in Section 6.1.1 and compared them to our approach, pointing out the limitations of this approach to support QoI. In particular, DDS [334] relies massively on multicast to support its QoS guarantees on the transport protocol level [269]. This is fine for local deployments in private networks where DDS is predominantly used; important properties such as latency, however, cannot be guaranteed by DDS in Cloud deployments [197]. Compared to the relevance of JMS and AMQP for industry-strength productive enterprise systems, the adoption of DDS is rather insignificant [17].

7.2 Related Approaches with Explicit Support for Qo*

We see most approaches proposed in the domain of push-based systems as complementary to our concept as they provide mechanisms to enforce dedicated quality-related properties that we can model with expectations. Thus, we discuss general purpose MOMs and domain-specific ones for environments with resource-constrained devices.

General purpose self-adaptive MOMs that allow subscribers to define explicit requirements about quality-related properties are Adamant, IndiQoS, and Harmony; in Section 6.1.1, we have already discussed the expressivity of those approaches. Other MOMs such as CREAM [105], DREAM [77], HERMES [348], JEDI [116, 117], Mundocore [9, 8], PADRES [157], RE-BECA [318], SIENA [86], or TinyCOPS [200] do not focus on explicit requirements about QoS. In the domains of WSNs and CPSs, several systems have been proposed to support QoS and Quality of Context (QoC): TinyDDS, CommonSens, CASSARAM/CA4IOT, and QoCIM/INCOME.

Hoffert et al. propose Adamant [210, 211, 212, 213] as a self-adaptive implementation of DDS. Adamant focuses on low-level QoS properties of the MOM such as latency, loss, or jitter. Subscribers can define requirements about those properties as maximum or minimum values within their subscriptions. The Adamant MOM tries to satisfy these requirements by choosing a wire protocol that provides the best performance in a certain setting; feedback about the system state is used only internally in terms of reinforced machine learning to select the most suitable transport protocol. Publishers are not adapted to enforce requirements about properties.

Carvalho et al. propose IndiQoS [22, 23, 85] to enforce requirements about the maximum latency of notifications. Subscribers add their requirements about latency, precision, or sampling rate to their subscriptions. The MOM monitors the latency at runtime and tries to enforce requirements about latency by reserving sufficient bandwidth using the RSVP resource reservation protocol. Requirements about precision and sampling rate are only supported by filtering out notifications that have not been labeled with matching attributes by publishers (cf., Section 6.1.1). These attributes cannot be adjusted at runtime to conform to the requirements of subscribers, as IndiQoS does not provide feedback to publishers.

Harmony [135, 253, 428] is a distributed MOM with a custom wire protocol that focuses on providing support for end-to-end latency. Subscribers can specify a maximum latency for notifications. Harmony maintains multiple paths between brokers and sends the same notification over multiple connections to compensate for fluctuating latencies. In addition, Harmony monitors the latency on each inter-broker network link and adapts the routing at runtime to minimize the transport latency [138].

TinyDDS [65] is an implementation of DDS for resource-constrained devices in a WSN or CPS that require a small footprint. TinyDDS does not support the full spectrum of QoS guarantees but focuses on enforcing latency requirements on the network layer by internally prioritizing notifications.

Soberg et al. propose the CommonSens system [392] to automatically select sensors based on their sensing capabilities in the domain of assisted living in smart homes. Capabilities in CommonSens refer to the notion of Locations of Interest (LOI) in smart homes, e.g., the kitchen or the living room; they do not describe single properties like precision or accuracy but sensing functionality of a sensor node, e.g., face recognition, temperature readings, or heart frequency readings [391, 393, 394]. Any notion of QoI is implicitly included in the semantics of a capability; subscribers can directly address neither QoS nor QoI.

Perera et al. propose CASSARAM [340] to support the user of a large-scale CPS in selecting a set of sensor nodes that provide data with a certain set of quality-related properties. They model their context framework as an extension to the Semantic Sensor Network (SSN) Ontology [418] that allows sensor nodes to semantically describe their own sensing capabilities. Based on these semantics, users define two sets of properties: point-based requirements denote properties that sensor nodes must provide while proximity-based properties are not mandatory; all properties

can be weighted according to the preferences of the users. Their CA4IOT [341, 342] framework embeds CASSARAM to process notifications only from matching sensor nodes, as well as enrich and transform them to conform to the semantics of the user. The focus of CASSARAM and CA4IOT is on selecting suitable sensor nodes by semantically matching fuzzy user requirements to ambiguous sensor descriptions. Thus, there is no distinction of publisher-side, broker-side or interdependent properties and no reactive adaptation of publishers or processing. CASSARAM focuses on helping the user to define priorities for sets of properties and bridge the gap between syntactically different but semantically similar property definitions. In contrast to our approach, CASSARAM and CA4IOT do not focus on the degree to which these properties have to be provided at runtime while the focus of our work is on runtime negotiation and adaptation of both MOM and publishers to leverage the adaptivity of publishers during the negotiation process. In addition, our approach does not require the use of an ontology or the distinction of mandatory and optional requirements.

Marie et al. propose QoCIM [295] as a domain-specific metamodel for describing properties of contextual data in the IoT. This model is used within their INCOME system [296] and allows subscribers to incorporate XPath queries over contextual properties in subscriptions while publishers add properties to advertisements (cf., Listing 6.3). Notifications are labeled with certain properties by publishers and are filtered out by INCOME if properties do not match the XPath expression of subscriptions. Each property can be modeled as an ordered set or range of values. In this, QoCIM and INCOME are very similar to IndiQoS, Adamant, CA4IOT, and a very similar model proposed by Kattepur et al. in [248]. Overall, their model does not provide the same extent of support for QoI that our approach does (cf., Section 6.1). While their property model is quite similar to our generic property definition in its minimalism, the QoCIM model, as well as the INCOME system, have severe limitations. First, QoCIM does not indicate how properties should be manipulated while we associate actions to a property and define capabilities for this reason. Consequently, QoCIM does not distinguish different types of properties depending on which type of participant is able to manipulate it, e.g., broker-side or interdependent properties. Thus, INCOME cannot reactively support requirements about broker-side or interdependent properties as this would require a differentiation of properties to decide if the MOM has to adapt itself or whether a certain set of publishers has to adapt or be filtered out. Furthermore, QoCIM and INCOME do not allow subscribers to express their Value of Information (VoI) while publishers cannot expose their costs. Lastly, INCOME does not provide feedback to publishers or subscribers. Hence, notifications with unsuitable properties can only be filtered out but the system cannot reactively adapt to improve those properties. We have shown the shortcomings of this and the superiority of our approach in our evaluation in Section 6.2.

7.3 Related Approaches Regarding Monitoring, Self-Adaptation, Negotiation

Related concepts originating from push- and pull-based concepts have inspired and shaped the work presented here. While a full overview can be found in Tables A.1 to A.4, we want to discuss selected ones in more detail here.

Modeling and Negotiating Requirements About Quality-related Properties

Work done by Keeton et al. [250] on general considerations about information quality and by Wilkes [424] on balancing requirements with consumers' utility has highly influenced our work. Behnel et al. [42] and Appel et al. [21] identify a basic set of quality guarantees and the levels

of abstractions specific to EBS that we have used as a basic skeleton for structuring our literature review presented in Section 3.2.1.

Modeling requirements about QoS properties is actively pursued for pull-based systems, such as SOA, and systems applying a push-based communication model, such as WSNs.

Kattepur et al. [248] define a QoS metric for SOA very similar to properties in QoCIM while most other authors propose approaches for automated renegotiation and multi-round negotiation for Service Level Agreements (SLAs) based on WS-Agreement [198, 227, 267, 380, 436].

Pernici et al. [346, 347] use fuzzy parameters and fuzzy set theory for deciding on web service adaptation in a SOA. A service consumer defines ranges over a set of quality dimensions that are fuzzy or mandatory, similar to the proximity-based and point-based properties in [340]. Each dimension is monotonic or antitonic, denoting whether it is desirable for the service provider to be close to the maximum (monotonic), or to the minimum (antitonic) defined by the service consumer. Service providers are asked to adjust their provided QoS or are replaced if they violate the requirements of the service consumer. A violation is quantified as a penalty using fuzzy set theory and depends on the individual requirements of the service consumer, its state as well as the history of the relationship between consumer and provider.

Bahjat et al. [31] as well as Byers and Nasser [79, 80] propose frameworks to capture trade-offs between low-level QoI-related properties in a WSN to optimize network utilization. Abdelzaher et al. [4] show that graceful degradation of QoS properties by renegotiation can be a feasible mechanism to maximize utility in time-critical distributed systems; Mai et al. [289] exploit trade-offs between completion time and price in data analytic applications in Cloud environments.

The design of expectations and capabilities used in our approach is inspired by these works; in particular the basic ideas of both Pernici et al. and CASSARAM. While CASSARAM quantifies the preferences of users regarding the priorities of different properties, fuzzy parameters as used by Pernici et al. to add a certain tolerance to each requirement. Expectations and capabilities fuse and extend these two different concepts to capture the requirements of subscribers and the support provided by publishers: individual sets of properties to be defined in an expectation or capability profile while a range of accepted or provided values can be defined for each property.

Runtime Monitoring

As a contribution of this dissertation, we propose ASIA as a novel approach for runtime monitoring of distributed push-based systems.

Push-based approaches are often used for monitoring distributed pull-based systems [250]. For example, frameworks for SOA-based workflow implementations such as VRESCo [307, 308, 310], VieDAME [315, 316], and MORSE [219] use PS and Complex Event Processing (CEP); Smit et al. use STORM [409] to monitor heterogeneous Cloud settings [389]; Guinea et al. [195] use SIENA [86] for collecting runtime information about services; Agarwala et al. [7] use a topic-based PS system to process monitoring updates with QoS constraints.

In distributed push-based systems, most brokers monitor their own state to base decisions about load-balancing, self-stabilization or reliability upon. For example, each broker in a Distributed Event-based System (DEBS) based on CREAM [105], DREAM [77], or PADRES [157] monitors the availability of neighboring brokers to avoid routing notifications to unavailable brokers. This information, however, is not available to subscribers or publishers as aggregated feedback.

Several approaches provide such information about the runtime state to participants by relying on a separate aggregation system such as Astrolabe, SDIMS, or STAR.

Astrolabe [412] provides metrics based on user-defined aggregation functions, implemented via a single logical aggregation tree on top of an unstructured peer-to-peer gossip protocol. SDIMS [427] performs hierarchical aggregation based on attribute types and names using distributed hash tables. STAR [240] adaptively sets the precision constraints for processing aggregate queries; it is used by Jain et al. [241] to provide a consistency metric for large-scale distributed systems that calculates the system's stability in terms of currently reachable nodes (ideally high) and the number of updates that might have been repeatedly processed due to network failures (ideally low).

In contrast to ASIA, such generic aggregation systems are unable to leverage specific properties of DEBSs, such as overlay topologies or exchanged messages. Aggregation trees may not match routing trees, resulting in inefficiency and delayed adaptation.

ASIA, the runtime monitoring approach proposed in this dissertation, relies on efficiently aggregating and distributing state information in a distributed network of brokers. Energy efficient in-network aggregation is studied in WSNs [152]. This is complementary to our work, as we do not compute aggregations within the energy-constrained WSN but within the infrastructure of the enterprise software system where energy-efficiency is an issue on a different scale. In addition, none of these systems support generic, application-specific aggregation or imprecision within the broker network; neither are they able to piggyback information.

Self-adaptation

A key component of our approach is adaptation: self-adaptation of the MOM to support requirements about broker-side properties and adaptation of publishers to enforce properties that require an adjustment by publishers or in coordination with the MOM.

CREAM [105] and DREAM [77] are *reactive* push-based MOMs that allow adaptation beyond the MOM. Their design has inspired the ReactionManagerClient component of our reference architecture as introduced in Section 5.1.3.

Runtime adaptation of predefined workflows is actively investigated for pull-based systems that implement workflows using SOA. Most approaches are based on the OODA cycle and utilize monitoring information about the software as well as the infrastructure layer via CEP and PS. For example, Moser et al. propose the VieDAME framework [316] to monitor and adapt pull-based Business Process Execution Language (BPEL) workflows at runtime based on custom replacement strategies; MORSE [219] adds a model-driven component that checks the conformance of the system state with the defined model. Similarly, Guinea et al. propose a multi-layer monitoring and adaptation framework for SOA-based applications [195]. Using machine learning and decision trees, they decide at runtime about replacing services used as part of a BPEL workflow.

While those approaches in pull-based systems mostly define adaptation as substituting services within a workflow, their general approach has influenced the overall structure of runtime negotiation and runtime enforcement in our approach that also follows the OODA cycle.

7.4 Summary

In this chapter, we have discussed the state-of-the-art regarding the key contributions of this dissertation. We have discussed standards that define APIs and wire protocols for asynchronous push-based communication with support for quality-related properties as well as reactive MOMs directly related to our approach. Based on this comparison, we have reviewed further push- and pull-based concepts that have inspired our work.

Summing up, several MOMs empower subscribers to define requirements about some quality-related properties of notifications. They try to enforce these requirements either by self-adaptation of the MOM (Adamant, IndiQoS, Harmony, TinyDDS) or by filtering out all notifications not conforming to the requirements (CA4IOT, INCOME, CommonSens). For self-adaptation, they primarily leverage the features of the wire protocols defined by some standards. None of the discussed approaches provides individual and aggregated feedback to its participants about the state of the system. Consequently, none of these approaches actively enforces requirements about quality-related properties determined and manipulated by publishers – such as accuracy – or about complex and interdependent properties – such as alternatives.

As we have shown in our quantitative evaluation in Section 6.2, only filtering out unsuitable notifications at the MOM results in a suboptimal fidelity for subscribers while brokers and publishers have to deal with overhead in resource utilization compared to our approach. Self-adapting the MOM only, however, limits the set of properties that are supported and omits enforcing publisher-related properties.

In contrast, the concept of expectations, capabilities and feedback presented in this dissertation expands the scope of runtime support to include the enforcement of publisher-related properties by runtime adaptation of publishers in addition to self-adaptation of the MOM. All related approaches discussed here could be utilized by our approach as requirements about the generic properties of our approach can be mapped to the properties supported in each approach.

8 Conclusion

Modern reactive software systems turn fine-granular real-time data about processes in the physical world into information and knowledge to react in time. Push-based architectures based on an Event-based System (EBS) complement pull-based architectures, such as Service-oriented Architectures (SOAs), and enable enterprises to react to meaningful events in a timely manner. Applications for algorithmic trading, energy-aware reactive data center management, or smart supply chain management are just three examples of reactive systems where information provided by heterogeneous data sources has to be interpreted and where false alarms, missed events or otherwise information of inadequate quality carries a cost.

Whether the Quality of Information (QoI) of received notifications is adequate depends on the purpose each receiver intends to use the information for. This purpose is application- as well as context- or even situation-specific. Thus, the notion of QoI incorporates two different aspects: objectively measurable *properties* of a notification and an application-specific *assessment* of these properties that determine the current Value of Information (VoI) for the receiver.

While QoI is crucial in modern reactive software systems, it is supported only to a limited degree in a typical EBS: subscribers can subscribe to dedicated properties of notifications that can be expressed with encoded types. This adds avoidable overhead to the system as an excessive number of encoded types has to be maintained when expressing multiple properties. Furthermore, publishers and the Message-oriented Middleware (MOM) cannot adjust properties due to missing feedback; changes to the context or situation of a subscriber require changes to its subscriptions to reflect an adjusted VoI. Above all, however, the set of supported properties is restricted to non-interdependent properties, excluding crucial properties required in modern reactive applications.

Domain-specific MOMs support a limited set of properties that are only manipulated by the MOM adapting itself but require specific platforms or custom transport protocols that may not be available in heterogeneous environments or public Clouds. Crucial properties that require the adaptation of publishers at runtime as well as interdependent properties that require multiple publishers to contribute to are not supported at all. Subscribers cannot weigh their requirements and publishers cannot expose their costs or adaptation capabilities. As shown in Chapter 6, this lack of support results in significant execution costs for subscribers, publishers and MOM.

In this dissertation, we have introduced the concept of expectations, capabilities and feedback as a holistic concept to express, negotiate and enforce QoI requirements at runtime in push-based systems. Our approach supports QoI requirements in heterogeneous systems as it abstracts requirements and capabilities about QoI from the implementation of participants and specific platforms. Instead of being limited to a fixed set of supported properties, our solution enables subscribers to define requirements about arbitrary quality-related properties and manage them at runtime without having to adjust subscriptions.

Subscribers expose preferences and tradeoffs between requirements in an information-centric way as expectations. They indicate the VoI of their requirements by ranking expectations with utility values without the need to define an explicit utility function. The fidelity of a subscriber quantifies the conformance of the data provided by the EBS to the subscriber's requirements.

Publishers expose their current support for generic properties, as well as the spectrum of support they could realize with self-adaptation, as capabilities. As providing notifications with specific generic properties comes at a cost depending on the design, configuration, context, and situation of the publisher, cost functions can be defined for each capability and manipulated at runtime.

In our approach, negotiating QoI requirements can be done automatically at runtime based on custom decision strategies. Our approach enables the negotiation mechanism to take into account the costs of participants and the VoI of requirements. The decision strategies can include load-balancing considerations and range from simple heuristics to sophisticated Multi-Objective Optimization Problem (MOOP) solvers.

Runtime adaptation is used to enforce the decision by advising publishers to adapt or by applying platform-specific adaptation mechanisms at the MOM itself. Individual and aggregated feedback enables participants to adapt their behavior at runtime and extends the scope of supported properties to those influenced by publishers.

Using a minimalistic and generic format to represent properties, our approach can utilize existing work that focuses on dedicated properties: our generic format can be mapped to the representation required by a related approach to apply its mechanisms.

This dissertation contributes to the challenge of runtime QoI support in push-based architectures on a *conceptual* and *practical* level.

On the conceptual level, we contribute a *generic and extensible model* to express and manage requirements about arbitrary properties based on expectations, capabilities and feedback (Chapter 3), *algorithms for negotiation, safeguarding and enforcing* these requirements as well as a *concept for effective runtime monitoring* in a distributed and decentralized EBS (Chapter 4). The conceptual part of this dissertation synthesizes and expands approaches devised in pull- and push-based systems as well as in economics into a novel concept to support QoI at runtime.

On the practical level, we contribute a *reference architecture* for runtime support of QoI requirements, *two prototypes* built on a centralized and a decentralized MOM, *examples for applications applying our approach* (Chapter 5) as well as an *extensive evaluation* of our prototypes (Chapter 6). The practical part of this dissertation shows the feasibility of our approach on different platforms, demonstrates that our approach can be integrated into existing applications, and quantifies the benefits of actively enforcing QoI requirements using feedback.

In summary, we do not only contribute and evaluate a novel generic and extensible concept to express, negotiate and actively enforce requirements about arbitrary QoI properties. In fact, we demonstrate the practicability and benefits of our approach: we have shown that our approach can be easily incorporated into existing industry-strength as well as academic systems without the need to redesign them. We have shown that the higher QoI of the processed data and the significant resource savings for participants compensate for the additional execution costs for the MOM. Our approach makes this trade-off between fidelity and performance explicit and eases the design of a new generation of reactive software systems with an inherent support for QoI.

9 Outlook and Future Work

In this chapter, we address future work regarding the proposed concept of expectations, capabilities and feedback to support Quality of Information (QoI) requirements. The topics discussed relate to four areas of interest that are shown in Figure 9.1. They successively extend the scope of the work presented in this dissertation from the domain of push-based systems to distributed software systems in general by including pull-based approaches and economic aspects.

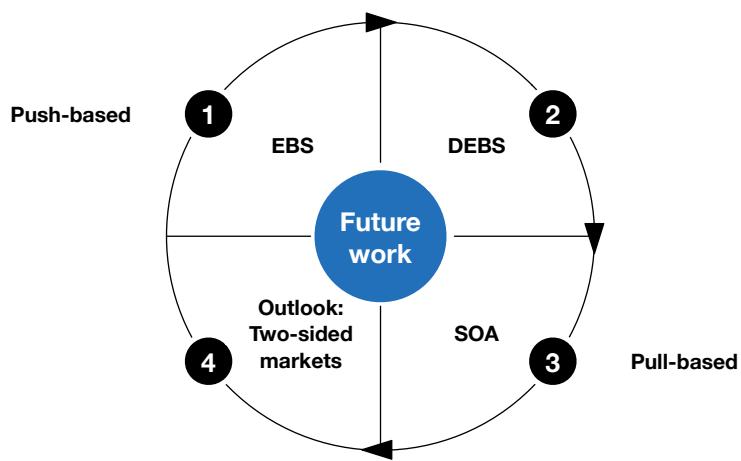


Figure 9.1.: Future work addresses challenges in push- and pull-based systems as well as general topics related to incentives and negotiation on two-sided markets.

The chapter is structured into four parts as shown in Figure 9.1. First, we focus on topics in the area of centralized Event-based Systems (EBSs) that immediately connect to the scope of the work as presented in this dissertation. Second, we discuss additional challenges in distributed and decentralized EBSs. Third, we extend our scope to include pull-based systems and the subsequent challenges there. Using the example of Service-oriented Architectures (SOAs), we show that our concept can be applied to pull-based software systems as well. Finally, we sketch out future work regarding incentives and negotiation in general as our concept connects the runtime support for QoI requirements to the concept of two-sided markets investigated in economics.

9.1 Centralized Event-Based Systems: Immediate Challenges

In this dissertation, we present a concept and working prototypes for modeling, negotiating and enforcing requirements about QoI in push-based systems. The work as presented in its current state, however, has some limitations that have to be addressed in future work. Performance and prediction models for forecasting the system behavior, alternative and vertically integrated adaptation mechanisms as well as approaches for coordinating rollbacks and compensation are areas of interest to be investigated in future work.

Expectations express requirements and preferences of subscribers in regard to different sets of generic properties. The system can satisfy or decline these bundles of requirements. Consequently, turning an expectation into a Service Level Agreement (SLA) between the subscriber and the Message-oriented Middleware (MOM) would be the next step. An SLA formalizes a

contract between a service provider and service consumer where the service provider guarantees the provisioning of a set of Service Level Objectives (SLOs); violating this guarantee results in penalties while adhering to it results in the subscriber paying a fee to the provider.

In our approach, an agreement comes about if the MOM notifies the subscriber that its expectation is satisfied. In the current development stage of our prototypes, however, the MOM does not *forecast* the system's behavior or *predict* the side effects of adaptations on other, currently satisfied, requirements before applying them. Thus, it may happen that the MOM cannot keep up the given guarantee due to deadlocks or conflicting effects of adaptations at runtime.

Consequently, future work in the area of QoI requirements should address system and performance models for push-based systems to enable prediction and forecasting as in pull-based systems [84]. Previous work as presented in [142, 259, 367, 410] can be used to build upon.

Investigating refined adaptation mechanisms and their interdependencies is another topic to be addressed in future work. Identifying new platform- or even protocol-specific adaptation mechanisms to be transparently applied by the MOM in an EBS is another area of interest. Increasing the level of vertical integration between the MOM, wire protocols for push-based systems such as the Advanced Message Queuing Protocol (AMQP) or the Channel-based Unidirectional Stream Protocol (CUSP) [406], or programming languages, would enable fine-tuned adaptation mechanisms that can be applied in specific environments. Machine learning approaches such as Fossa [180] should be investigated further to support the MOM in deciding on the set of adaptation mechanisms to apply in a promising situation; this would also tie in well with previous work done for domain-specific approaches such as IndiQoS or Adamant [211, 212].

Directly related to alternative adaptation mechanisms are concepts for coordination and compensation in an EBS: the more adaptation mechanisms can be applied to different degrees and in different combinations, the more important coordination and compensation become. Compensation is crucial, in particular, when applying adaptations to optimize the utilization of the system, i.e., to free up resources. While this topic has gained a lot of attention in the database community and is still actively pursued, compensation mechanisms in push-based systems should be addressed more actively in future work [196].

9.2 Decentralized Event-Based Systems: Synchronizing State

As part of our quantitative evaluation we have shown in Section 6.3 that a single broker becomes saturated at some point for large and dynamic populations, which require frequent renegotiations. Distributed Event-based Systems (DEBSs) can be used to overcome these limitations of a single broker as a network of brokers allows the load to be distributed across different brokers using load-balancing approaches such as publisher placement [101] or topology reconfigurations [239, 317, 319, 320, 336, 375, 429].

In a DEBS, however, additional research challenges emerge that have to be addressed in future work. While the challenges discussed in the previous section also apply to each broker in a distributed MOM, having to synchronize critical information between multiple brokers remains the key challenge in a DEBS.

Two types of critical information have to be synchronized at runtime to support QoI requirements: *state information* necessary for brokers to decide on expectations and *calibration information* to coordinate adaptations in a distributed setup without deadlocks or side effects.

In a centralized MOM consisting of a single broker, local knowledge about the system state is identical with the global state itself. In a distributed environment, however, local knowledge has to be enriched with the fraction of global knowledge required for local decisions at a particular broker. Thus, synchronizing state and calibration information includes aspects of partitioning and updating. We sketch out some ideas to address these synchronization aspects for negotiating and enforcing expectations in distributed and decentralized push-based systems.

In a network of brokers, only the edge broker decides about the expectations of its directly connected subscribers. Partitioning the critical state as such does not require changes to our set- and range matching algorithms: all edge brokers connected to publishers forward the capability profiles of their publishers along the routing tree towards the edge brokers of subscribers subscribed for matching types of notifications. With this knowledge, edge brokers can decide whether an expectation is satisfied, satisfiable or unsatisfied. Updated capabilities are propagated through the network and trigger renegotiations while updates to expectations trigger renegotiations as in the single broker case discussed in Chapter 4.

Scopes and *imprecision* are promising concepts to support scalability and provide availability for large and dynamic populations in a DEBS. They minimize the number of brokers affected by updates as well as the number of renegotiations performed at each affected broker. Future work should explore them in more detail.

Imprecision to minimize the number of updates. Instead of flooding the broker network with every update to a capability, only those updates are propagated that would either violate an already satisfied requirement or would satisfy a requirement that is hitherto unsatisfied. As in ASIA, the ranges of accepted values defined in each expectation can be aggregated to compute thresholds that trigger updates at edge brokers of publishers or inner brokers.

Scopes to minimize the number of affected brokers. Propagating updates from publishers to edge brokers of subscribers along the routing tree already partitions the network using subscriptions and avoids flooding. However, the set of brokers to be notified about a changing capability profile can still be narrowed down as changes to a publisher's capability profile are relevant only to those edge brokers that manage expectations this publisher is currently satisfying or that it could satisfy. Scopes [160, 235, 318] can be used to define such areas of interest in a topology: each expectation defines a scope that contains all publishers with capability profiles that satisfy the expectation or would be able to satisfy it. At the same time, each publisher defines a scope that contains all subscribers that this publisher is catering to. Only the edge brokers in a scope have to be notified about changes and only the expectations in a publisher's scope have to be renegotiated if the capabilities of the publisher change to a significant degree. Conversely, the scope defined by an expectation identifies those publishers that could benefit from the expectation being revoked or relaxed; these publishers could then be advised by the MOM to save resources.

Coordinated adaptation in distributed systems is another area of interest to be addressed in future work. As for the centralized case, coordination entails approaches for prediction, forecasting, and compensation but with the additional challenge of a distributed environment. We believe that enforcing adaptation in distributed environments should be built upon self-adaptive and self-stabilizing approaches as investigated in [237, 238, 239, 311, 319, 320, 375].

9.3 Applying our Approach to Pull-Based Systems

The work presented in this dissertation focuses on the support for requirements about QoI in EBS and DEBS. The concept of expectations, capabilities and feedback, however, is not limited to the domain of push-based systems. In fact, open issues emerging in the context of runtime governance of service-based systems are closely related to the challenges we have discussed for push-based systems. SOAs are a prominent example for pull-based systems where consumers and providers of a service interact directly following a request-reply communication pattern.

While our approach has been developed in context of push-based EBSs and focuses on the role of a Message-oriented Middleware (MOM), we can apply the model of expectations and capabilities to broker-less and even pull-based systems. In pull-based systems without an intermediate MOM, the service consumer has to negotiate directly with each suitable service provider instead of delegating this effort to the MOM. Especially requirements about intermediate generic properties, such as alternatives, require the service consumer to continuously check whether the requirement is satisfied, e.g., whether agreements with enough suitable service providers have been established; apart from this, our concept can be applied directly.

In this section, we show that our model easily integrates with existing protocols and standards for negotiating SLAs at runtime in pull-based systems. As an example, we extend the WS-Agreement protocol for automated SLA negotiation we have introduced in Section 2.2.3 to negotiate an SLA based on expectations between a service consumer and a service provider.

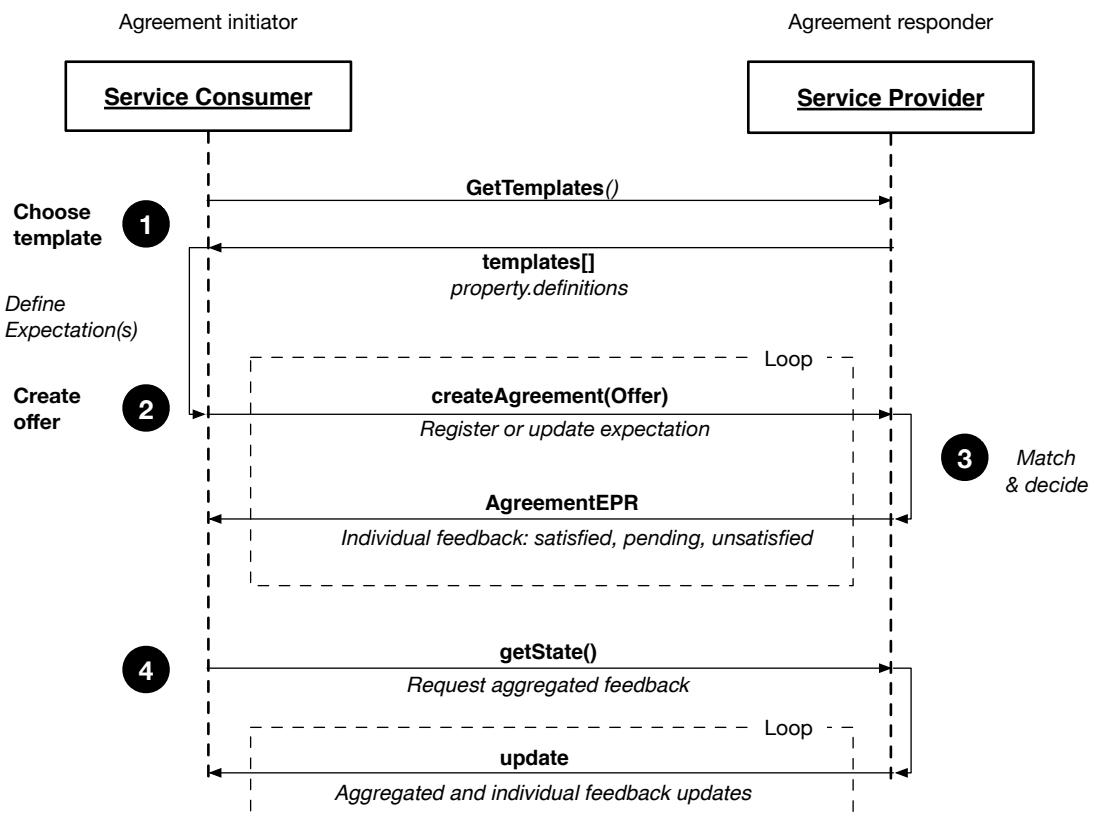


Figure 9.2.: Steps of the WS-Agreement protocol (**bold**) using expectations (*italic*).

The WS-Agreement protocol consists of four steps as illustrated in Figure 9.2.

1. The consumer requests *templates* from the provider. These templates contain all SLOs the provider is supporting. When using expectations, the provider's reply contains the definitions for all generic properties that the provider supports matching capabilities for.
2. The consumer compiles an *agreement offer* that contains a set of SLOs out of these templates. Using expectations, this step refers to registering or updating an expectation that is defined over generic properties contained in the template.
3. The provider rejects the agreement offer or accepts it by sending an *agreement EPR* that contains all clauses of the agreement. Using expectations, the service provider sends individual feedback about the state of an expectation: satisfied, pending, or unsatisfied. Either state can trigger a new iteration of the negotiation process at the service consumer who updates its expectation and sends a new offer to the service provider.
4. The consumer can request monitoring updates about the status of the agreement. Using expectations, the consumer can request updates about aggregated ASIA metrics. Individual feedback about the current state of an registered expectation is provided anyway.

Listings 9.1 to 9.3 illustrate how expectations and individual feedback can be expressed in WS-Agreement syntax. We assume a service consumer with preferences as described by expectation X_6^{pos} and a service provider with a matching set of capabilities. The negotiation with WS-Agreement results in a template (Listing 9.1) and an agreement offer (Listing 9.2).

The WS-Agreement syntax allows us to state that an expectation has to contain requirements about at least one generic property defined in the template (cf., Listing 9.1 line 14). The agreement offer that represents an expectation, however, is only satisfied if *all* requirements are satisfied (cf., Listing 9.2 line 14). The utility value used by the subscriber to rank a set of requirements is matched by the `importance` element embedded in the `BusinessValueList` of the agreement offer (cf., Listing 9.2 line 44 on).

Modeling the definitions of generic properties as freeform `ServiceDescriptionTerm` elements in the template (cf., Listing 9.1 line 15 on) enables us to define range- as well as list-based generic properties with improvement directions (e.g., line 49 on) as introduced in Chapter 3.

In our example, we assume that the expectation is satisfied at first (cf., Listings 9.3 and 9.4) but has to be declined later as the currently provided latency becomes too high and the precision too low as stated in the list of reasons shown in Listing 9.5 (line 6 on). Please note that the only element required by the WS-Agreement syntax of this response is the `state` element in line 4. Thus, we can structure the list of reasons for declining the requirements conveniently by reporting the required versus the currently provided value.

Listing 9.1: WS-Agreement template example with expectations.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <wsag:Template xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement" xmlns:propdef=
  "http://dvs.tu-darmstadt.de/XMLpropertydef" wsag:TemplateId="12345">
3
4   <wsag:Name>template1</wsag:Name>
5   <wsag:Context>
6     <wsag:ServiceProvider>AgreementResponder</wsag:ServiceProvider>
7     <wsag:TemplateId>12345</wsag:TemplateId>
8     <wsag:TemplateName>template1</wsag:TemplateName>
9     <wsag:AgreementInitiator>subscriberA</wsag:AgreementInitiator>
10    <wsag:AgreementResponder>providerB</wsag:AgreementResponder>
11  </wsag:Context>

```

```

12
13 <wsag:Terms>
14   <wsag:OneOrMore>
15     <wsag:ServiceDescriptionTerm wsag:Name="XPECT" wsag:ServiceName="XPECT_SERVICE">
16       <propdef:definitions>
17         <propdef:property abbrev="accuracy" improvement="MAX">
18           <propdef:value datatype="DOUBLE">
19             <propdef:range>
20               <propdef:minimum>0</propdef:minimum>
21               <propdef:maximum>100</propdef:maximum>
22             </propdef:range>
23           </propdef:value>
24         </propdef:property>
25         <propdef:property abbrev="latency" improvement="MIN">
26           <propdef:value datatype="DOUBLE">
27             <propdef:range>
28               <propdef:minimum>0</propdef:minimum>
29               <propdef:maximum>400</propdef:maximum>
30             </propdef:range>
31           </propdef:value>
32         </propdef:property>
33         <propdef:property abbrev="precision" improvement="MAX">
34           <propdef:value datatype="DOUBLE">
35             <propdef:range>
36               <propdef:minimum>0</propdef:minimum>
37               <propdef:maximum>100</propdef:maximum>
38             </propdef:range>
39           </propdef:value>
40         </propdef:property>
41         <propdef:property abbrev="sampling rate" improvement="MAX">
42           <propdef:value datatype="INTEGER">
43             <propdef:range>
44               <propdef:minimum>0</propdef:minimum>
45               <propdef:maximum>60</propdef:maximum>
46             </propdef:range>
47           </propdef:value>
48         </propdef:property>
49         <propdef:property abbrev="trust" improvement="MAX">
50           <propdef:value datatype="STRING">
51             <propdef:list>
52               <propdef:listvalue>none</propdef:listvalue>
53               <propdef:listvalue>low</propdef:listvalue>
54               <propdef:listvalue>medium</propdef:listvalue>
55               <propdef:listvalue>high</propdef:listvalue>
56             </propdef:list>
57           </propdef:value>
58         </propdef:property>
59       </propdef:definitions>
60     </wsag:ServiceDescriptionTerm>
61   </wsag:OneOrMore>
62 </wsag:Terms>
63 </wsag:Template>

```

Listing 9.2: Agreement offer with expectations (example).

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <wsag:AgreementOffer xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement"
  xmlns:expect="http://dvs.tu-darmstadt.de/XMLExpect" wsag:AgreementId="54321">
3
4   <wsag:Name>agreementOffer1</wsag:Name>
5   <wsag:Context>
6     <wsag:ServiceProvider>AgreementResponder</wsag:ServiceProvider>
7     <wsag:TemplateId>12345</wsag:TemplateId>
8     <wsag:TemplateName>template1</wsag:TemplateName>
9     <wsag:AgreementInitiator>subscriberA</wsag:AgreementInitiator>
10    <wsag:AgreementResponder>providerB</wsag:AgreementResponder>
11  </wsag:Context>
12
13  <wsag:Terms>
14    <wsag:All>
15      <wsag:GuaranteeTerm wsag:Name="XPECT-GUARANTEE" Obligated="ServiceProvider">
16        <wsag:ServiceScope wsag:ServiceName="XPECTService"/>
17        <wsag:ServiceLevelObjective>
18          <wsag:CustomServiceLevel>
19            <expect:expectation GUID="e5">
20              <expect:property abbrev="accuracy">
21                <expect:closed>false</expect:closed>
22                <expect:lower-bound>70</expect:lower-bound>
23                <expect:upper-bound>100</expect:upper-bound>
24              </expect:property>
25              <expect:property abbrev="precision">
26                <expect:closed>false</expect:closed>
27                <expect:lower-bound>85</expect:lower-bound>
28                <expect:upper-bound>100</expect:upper-bound>
29              </expect:property>
30              <expect:property abbrev="latency">
31                <expect:closed>false</expect:closed>
32                <expect:lower-bound>0</expect:lower-bound>
33                <expect:upper-bound>250</expect:upper-bound>
34              </expect:property>
35              <expect:property abbrev="sampling rate">
36                <expect:closed>true</expect:closed>
37                <expect:lower-bound>10</expect:lower-bound>
38                <expect:upper-bound>20</expect:upper-bound>
39              </expect:property>
40            </expect:expectation>
41          </wsag:CustomServiceLevel>
42        </wsag:ServiceLevelObjective>
43
44        <wsag:BusinessValueList>
45          <wsag:Importance>25</wsag:Importance>
46        </wsag:BusinessValueList>
47
48      </wsag:GuaranteeTerm>
49    </wsag:All>
50  </wsag:Terms>
51 </wsag:AgreementOffer>
```

Listing 9.3: Agreement example, satisfied expectation.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <wsag:Agreement xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement" xmlns:expect=
  "http://dvs.tu-darmstadt.de/XMLExpect" wsag:AgreementId="54321">
3
4   <wsag:Name>agreement1</wsag:Name>
5
6   <wsag:Context>
7     <wsag:ServiceProvider>AgreementResponder</wsag:ServiceProvider>
8     <wsag:TemplateId>12345</wsag:TemplateId>
9     <wsag:TemplateName>template1</wsag:TemplateName>
10    <wsag:AgreementInitiator>subscriberA</wsag:AgreementInitiator>
11    <wsag:AgreementResponder>providerB</wsag:AgreementResponder>
12  </wsag:Context>
13
14  <wsag:Terms>
15    <wsag:All>
16      <wsag:GuaranteeTerm wsag:Name="XPECT_GUARANTEE" Obligated="ServiceProvider">
17        <wsag:ServiceScope wsag:ServiceName="XPECT_SERVIC"/>
18        <wsag:ServiceLevelObjective>
19          <wsag:CustomServiceLevel>
20            <expect:expectation GUID="e5">
21              <expect:property abbrev="accuracy">
22                <expect:closed>false</expect:closed>
23                <expect:lower-bound>70</expect:lower-bound>
24                <expect:upper-bound>100</expect:upper-bound>
25              </expect:property>
26              <expect:property abbrev="precision">
27                <expect:closed>false</expect:closed>
28                <expect:lower-bound>85</expect:lower-bound>
29                <expect:upper-bound>100</expect:upper-bound>
30              </expect:property>
31              <expect:property abbrev="latency">
32                <expect:closed>false</expect:closed>
33                <expect:lower-bound>0</expect:lower-bound>
34                <expect:upper-bound>250</expect:upper-bound>
35              </expect:property>
36              <expect:property abbrev="sampling rate">
37                <expect:closed>true</expect:closed>
38                <expect:lower-bound>10</expect:lower-bound>
39                <expect:upper-bound>20</expect:upper-bound>
40              </expect:property>
41            </expect:expectation>
42          </wsag:CustomServiceLevel>
43        </wsag:ServiceLevelObjective>
44
45        <wsag:BusinessValueList>
46          <wsag:Importance>25</wsag:Importance>
47        </wsag:BusinessValueList>
48
49      </wsag:GuaranteeTerm>
50    </wsag:All>
51  </wsag:Terms>
52 </wsag:Agreement>
```

Listing 9.4: WS-Agreement feedback about a satisfied expectation.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <wsag:AgreementState xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement">
3
4   <wsag:State>Observed</wsag:State>
5
6 </wsag:AgreementState>
```

Listing 9.5: WS-Agreement feedback about a rejected (unsatisfied) expectation.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <wsag:AgreementState xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement"
3   xmlns:cappro="http://dvs.tu-darmstadt.de/XML_cappro" >
4
5   <wsag:State>Rejected</wsag:State>
6
7   <rejected:reasons>
8     <reason:capability abbrev="precision">
9       <required>85</required>
10      <provided>70</provided>
11    </reason:capability>
12    <reason:capability abbrev="latency">
13      <required>250</required>
14      <provided>400</provided>
15    </reason:capability>
16  </rejected:reasons>
17 </wsag:AgreementState>
```

Dependency trees that constrain the support for certain generic properties are a currently underestimated topic in service-based systems. These dependency trees emerge at runtime and stem from the reuse of existing services within a new service at design time: a service provider requires results provided by other services to provide its own service. These dependencies are hidden from the consumer of a service by the interface of the providing service. This is beneficial from a development as well as a data ownership point of view as it minimizes redundant data copies within a service-based application landscape. However, from a runtime governance point of view, these hidden dependencies are disadvantageous as they affect the availability of the whole application landscape.

We have investigated dependency trees in SOAs in [169] based on an analysis of a large-scale productive service-based application landscape. Future work that aims at integrating pull- and push-based approaches has to consider the implications of dependency trees on the process of negotiating and guaranteeing QoI requirements.

9.4 Economic Perspective: Incentives and Negotiation on Two-Sided Markets

The widespread adaptation of public Cloud offerings for large-scale deployments of applications emphasizes the impact of monetary aspects such as dynamic pricing strategies or incentives on the design and runtime management of distributed applications. Current research in this area directly fuses advances in economics with those in computer science as most public Cloud offerings reflect the dynamics of classical single-sided markets [173, 289, 373, 420].

Two-sided markets differ from single-sided markets in several aspects that affect strategies for negotiation and revenue maximization [335, 358, 359]. In a two-sided market, participants can only interact with each other using an intermediary party, the platform provider. The platform itself cannot satisfy the demand of consuming participants without the presence of providing participants. As the sole party knowing demand and supply, the platform provider is not only responsible for negotiating demand and supply for each participant but it can also apply different strategies to attract or discourage participants from joining the system and trading goods. Examples for those two-sided markets are app stores for mobile operating system such as Android¹ or iOS, video game platforms such as Steam² and marketplaces such as ebay³.

In an EBS, the MOM already acts as a platform to negotiate the demand of subscribers with the supply provided by publishers. However, we currently assume that subscribers demand only their factual requirements while publishers adhere to adaptation advices from the MOM wherever and whenever technically possible.

Autonomous and interoperable participants interacting with each other across organizational boundaries to provide and request data from and for the Internet of Things (IoT) are a vision to be realized in the near future [63, 170]. This vision, however, turns an EBS into a two-sided market with autonomously acting and maneuvering participants. This requires future work in the area of QoI to address behavioral aspects of participants currently investigated in economics. Incentives, pricing strategies, and negotiation approaches in the face of network effects, multi-homing, gambling behavior, bandwagon effects, or fraudulent behavior on two-sided markets are just a few examples for aspects to be considered in the design and runtime management of future EBSs [26, 71, 149, 256, 276, 279, 335, 358, 359, 360, 361].

¹ <https://play.google.com/>

² <http://store.steampowered.com/>

³ <http://www.ebay.com>

A Appendix

A.1 Literature Survey Details

Table A.1.: List of QoD properties discussed in literature addressing runtime quality aspects.

Quality of Device (QoD) properties							
Reference	Cost	Distance	Drift	Location	Resolution	Sampling Rate	Sensitivity
[1]	■	□	□	□	□	□	□
[5]	□	□	□	□	□	□	□
[6]	■	□	□	■	□	□	□
[21]	□	□	□	□	□	□	□
[22]	□	□	□	□	□	■	□
[31]	□	□	□	□	□	□	□
[32]	□	□	□	□	□	□	□
[35]	■	■	□	■	■	■	□
[37]	■	□	□	□	□	□	□
[42]	□	□	□	□	□	□	□
[47]	□	□	□	□	□	■	□
[46]	■	□	□	■	□	□	□
[51]	□	□	□	□	□	■	□
[54]	■	□	□	□	□	■	□
[55]	□	□	□	■	■	□	□
[56]	■	□	□	■	□	□	□
[57]	□	□	■	■	■	■	□
[62]	□	□	□	■	■	■	□
[75]	□	■	□	■	■	■	□
[85]	□	□	□	□	□	□	□
[95]	□	□	□	□	□	□	□
[96]	■	□	■	■	□	■	□
[99]	■	□	□	□	□	■	□
[109]	■	■	■	■	■	■	■
[110]	■	□	□	■	■	■	□
[112]	□	□	□	□	□	□	□
[122]	□	□	□	□	□	□	□
[125]	■	□	□	□	□	□	□
[153]	■	□	□	□	□	■	□
[159]	□	□	□	□	□	□	□
[182]	□	□	□	□	□	□	□
[183]	□	□	□	■	□	□	□
[197]	■	□	□	□	□	■	□
[208]	■	□	□	□	□	□	□
[209]	■	□	□	□	□	■	□
[210]	□	□	□	□	□	■	□
[223]	□	□	□	■	□	□	□
[224]	□	□	□	□	□	□	□
[225]	□	□	□	■	□	□	□
[226]	□	□	□	□	□	□	□
[244]	□	□	□	□	□	□	□

Table A.1.: List of QoD properties discussed in literature addressing runtime quality aspects.

Quality of Device (QoD) properties							
Reference	Cost	Distance	Drift	Location	Resolution	Sampling Rate	Sensitivity
[247]	□	□	□	□	□	□	□
[248]	■	□	□	□	□	□	□
[250]	■	□	□	□	□	■	□
[254]	□	□	□	□	□	□	□
[261]	■	□	□	□	■	■	□
[263]	■	□	□	□	□	□	□
[265]	□	□	□	□	□	□	□
[271]	□	□	□	□	□	□	□
[288]	□	□	□	□	□	□	□
[291]	□	□	□	□	■	■	□
[293]	□	□	□	■	■	■	■
[295]	□	□	□	□	■	■	■
[318]	□	□	□	□	□	□	□
[321]	■	□	□	□	■	□	□
[324]	□	□	□	□	□	■	□
[325]	■	□	□	□	□	■	■
[334]	□	□	□	□	□	■	□
[339]	□	□	□	□	□	□	□
[340]	■	□	■	■	■	■	■
[345]	□	■	□	□	□	■	□
[346]	■	□	□	□	□	□	□
[354]	■	□	□	■	□	■	□
[357]	□	□	□	□	□	□	□
[366]	■	□	□	□	□	□	□
[384]	■	□	□	□	□	□	□
[386]	■	□	□	■	■	■	■
[403]	□	□	□	□	□	□	□
[415]	■	□	□	□	■	■	■
[425]	□	□	□	□	■	□	□
[428]	□	□	□	□	□	□	□
[430]	□	□	□	□	□	□	□

Table A.2.: List of QoS properties discussed in literature addressing runtime quality aspects.

Reference	Quality of Service (QoS) properties											
	Alternatives	Availability	Bandwidth	Completeness	Compression	Delivery guarantees	Jitter	Latency	Loss	Message size	Order	Priority
[1]	□	□										
[5]	□	□										
[6]	□	□	■									
[21]	□	□	■	□	□	□	□	□	□	□		
[22]	□	□	■	□	□	□	□	□	□	□		
[31]	□	□	□	□	□	□	□	□	□	□		
[32]	□	□	□	□	□	□	□	□	□	□		
[35]	■											
[37]	□	□	□	□	□	□	□	□	□	□		
[42]	■											
[47]	□	□	□	□	□	□	□	□	□	□		
[46]	□	□	□	□	□	□	□	□	□	□		
[51]	□	□	□	□	□	□	□	□	□	□		
[54]	□	□	□	□	□	□	□	□	□	□		
[55]	□	□	□	□	□	□	□	□	□	□		
[56]	□	□	■	□	□	□	□	□	□	□		
[57]	□	□	□	□	■	□	□	□	□	□		
[62]	□	□	□	□	□	□	□	□	□	□		
[75]	□	□	■	□	□	□	□	□	□	□		
[85]	□	□	□	□	□	□	□	□	□	□		
[95]	□	□	□	■	□	□	□	□	□	□		
[96]	□	□	□	□	□	□	□	□	□	□		
[99]	□	□	□	□	□	■	□	□	□	□		
[109]	□	□	□	□	□	□	□	■	□	□		
[110]	□	□	□	□	□	□	□	□	□	□		
[112]	□	■	□	□	□	□	□	□	□	□		
[122]	□	□	□	□	■	□	□	□	□	□		
[125]	□	□	□	□	□	■	□	□	□	□		
[153]	□	□	□	□	□	■	□	□	□	□		
[159]	□	■	■	□	□	□	■	□	□	□		
[182]	□	□	□	□	■	□	□	□	□	□		
[183]	□	□	□	□	□	□	□	□	□	□		
[197]	□	□	■	□	□	□	■	□	□	□		
[208]	□	□	□	□	□	□	□	■	□	□		
[209]	□	■	■	■	■	■	□	□	□	□		
[210]	□	■	□	□	□	□	■	■	□	□		
[223]	□	□	□	□	□	□	□	■	□	□		

Table A.2.: List of QoS properties discussed in literature addressing runtime quality aspects.

Quality of Service (QoS) properties																		
Reference	Alternatives			Availability			Bandwidth			Completeness			Compression			Delivery guarantees		
[224]	□	□	□	□	□	□	□	□	□	□	□	□	□	□	■	■	■	■
[225]	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
[226]	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
[244]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[247]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[248]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[250]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[254]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[261]	□	□	■	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[263]	□	□	■	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[265]	□	□	■	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[271]	□	□	■	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[288]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[291]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[293]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[295]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[318]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[321]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[324]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[325]	■	□	■	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[334]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[339]	■	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[340]	■	■	■	■	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[345]	□	■	■	■	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[346]	□	■	■	■	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[354]	□	□	□	■	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[357]	□	□	■	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[366]	□	□	■	□	□	□	■	□	□	■	□	□	□	□	□	□	□	□
[384]	□	■	■	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[386]	■	■	■	■	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[403]	□	□	□	■	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[415]	■	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[425]	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[428]	□	■	□	□	□	□	□	□	□	■	□	□	□	□	□	□	□	□
[430]	□	□	□	■	□	□	□	□	□	■	□	□	□	□	□	□	□	□

Table A.3.: List of QoI properties discussed in literature addressing runtime quality aspects.

Reference	Quality of Information (QoI) properties					
	Accuracy	Confidence	Freshness	Precision	Provenance	Trustworthiness
[1]	□	□	□	□	□	□
[5]	■	■	■	□	□	■
[6]	■	■	■	■	□	■
[21]	■	■	□	□	□	□
[22]	□	□	□	□	□	□
[31]	■	■	■	■	■	□
[32]	□	□	□	□	□	□
[35]	■	■	□	■	■	■
[37]	■	■	■	□	□	■
[42]	□	□	■	□	□	■
[47]	□	□	■	□	□	■
[46]	■	□	■	■	□	■
[51]	■	■	■	■	□	■
[54]	■	□	■	■	□	□
[55]	■	□	■	□	■	■
[56]	■	□	■	□	■	■
[57]	■	■	■	□	■	■
[62]	□	□	□	■	□	□
[75]	■	■	■	■	□	■
[85]	□	□	□	□	□	□
[95]	■	□	■	□	□	□
[96]	■	■	□	□	□	□
[99]	■	□	■	□	□	□
[109]	■	□	■	■	■	■
[110]	■	□	□	□	■	□
[112]	□	□	■	□	□	■
[122]	□	□	■	■	□	□
[125]	■	■	□	■	□	□
[153]	□	□	■	□	□	□
[159]	□	□	■	□	□	□
[182]	■	□	■	□	□	■
[183]	■	■	□	□	□	□
[197]	□	□	□	□	□	□
[208]	□	□	■	□	□	□
[209]	■	■	■	■	□	■
[210]	■	■	□	□	□	□
[223]	■	■	■	■	□	□
[224]	■	■	■	□	□	□
[225]	■	■	■	□	□	□
[226]	■	■	■	□	□	■
[244]	■	□	■	□	□	■
[247]	■	□	■	□	□	■
[248]	□	□	■	□	□	■

Table A.3.: List of QoI properties discussed in literature addressing runtime quality aspects.

Reference	Quality of Information (QoI) properties					
	Accuracy	Confidence	Freshness	Precision	Provenance	Trustworthiness
[250]	■	■	■	■	■	■
[254]	■	□	■	□	□	□
[261]	■	■	■	■	□	■
[263]	■	□	■	■	□	■
[265]	□	□	□	□	□	□
[271]	■	□	□	□	□	□
[288]	□	□	■	□	□	□
[291]	□	■	■	■	□	■
[293]	■	□	□	■	□	□
[295]	■	□	■	■	□	■
[318]	□	□	□	□	□	□
[321]	■	■	□	□	□	■
[324]	■	■	■	■	□	■
[325]	■	■	□	■	□	□
[334]	□	□	□	□	□	□
[339]	■	■	□	□	□	□
[340]	■	□	□	■	□	■
[345]	□	□	□	□	□	□
[346]	□	□	□	□	□	□
[354]	□	□	■	□	□	□
[357]	□	□	■	□	□	□
[366]	■	■	■	■	□	□
[384]	□	□	□	□	□	■
[386]	■	■	□	■	□	■
[403]	□	□	■	□	□	□
[415]	■	■	□	■	□	□
[425]	■	□	■	■	□	□
[428]	□	□	■	□	□	■
[430]	■	■	■	□	■	□

Table A.4.: List of VoI properties discussed in literature addressing runtime quality aspects.

Value of Information (VoI) properties												
	Reference	Accessibility	Appropriate Amount	Believability	Conciseness	Currency	Consistency	Ease of Manipulation	Effort	Integrity	Interpretability	Objectivity
[1]												
[5]												
[6]												
[21]												
[22]												
[31]												
[32]												
[35]	■											
[37]	■											
[42]												
[47]												
[46]												
[51]												
[54]												
[55]												
[56]												
[57]												
[62]												
[75]												
[85]												
[95]												
[96]												
[99]												
[109]												
[110]												
[112]												
[122]												
[125]												
[153]												
[159]												
[182]	■											
[183]												
[197]									■			
[208]									■			
[209]										■		
[210]										■		
[223]										■		

Table A.4.: List of VoI properties discussed in literature addressing runtime quality aspects.

Value of Information (VoI) properties															
	Reference	Accessibility		Appropriate Amount		Believability		Conciseness		Currency		Consistency		Ease of Manipulation	
[224]		□	□	□	□	□	□	□	□	□	□	□	■	□	
[225]		□	■	□	□	□	□	□	□	□	□	□	□	□	
[226]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[244]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[247]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[248]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[250]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[254]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[261]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[263]		■	□	□	□	□	□	□	□	□	□	□	□	□	
[265]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[271]		■	□	□	□	□	□	□	□	□	□	□	□	□	
[288]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[291]		■	□	□	□	□	□	□	□	□	□	□	□	□	
[293]		■	■	□	□	□	□	□	□	□	□	□	□	□	
[295]		■	□	□	□	□	□	□	□	□	□	□	□	□	
[318]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[321]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[324]		□	□	■	□	□	□	□	□	□	□	□	□	□	
[325]		□	■	□	□	□	□	□	□	□	□	□	□	□	
[334]		■	□	□	□	□	□	□	□	□	□	□	□	□	
[339]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[340]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[345]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[346]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[354]		□	□	□	□	□	□	□	□	□	□	■	□	□	
[357]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[366]		□	□	□	□	□	□	□	□	□	□	■	□	□	
[384]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[386]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[403]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[415]		□	□	□	□	□	□	□	□	□	□	□	□	□	
[425]		□	□	□	□	□	□	□	□	□	□	■	□	□	
[428]		□	□	□	□	□	□	□	□	□	□	■	□	□	
[430]		□	□	□	□	□	□	□	□	□	□	■	■	□	

A.2 Runtime Negotiation: Pseudocode

Algorithm 2: Set-matching algorithm described on page 81 in pseudocode used to store capability profiles in $\text{Nom}_{X_i^e}$ that have capabilities matching the requirements defined in X_i^e .

```

Function FindNOM( $X_i^e$ ,  $\{\text{CP}_1^e, \dots, \text{CP}_n^e\}$ ) is
     $\text{Nom}_{X_i^e} \leftarrow \emptyset$                                 // No suitable nominees so far
    foreach  $\text{CP}_l^e \in \{\text{CP}_1^e, \dots, \text{CP}_n^e\}$  do          // For every capability profile
         $match \leftarrow \text{true}$ 
        foreach  $p_k \in X_i^e \wedge \neg \text{isInterdependent}(p_k)$  do
            if  $\Theta(p_k) \notin \text{CP}_l^e$  then                      // Check if property is supported
                 $match \leftarrow \text{false}$                             // No matching capability
            if  $match = \text{true}$  then                          // All properties are supported
                 $\text{Nom}_{X_i^e}.add(\text{CP}_l^e)$                     // Add capability profile to nominees
    return  $\text{Nom}_{X_i^e}$ 

```

Algorithm 3: Pseudocode for relationship detection used during range-matching requirements to capabilities in Algorithms 4 and 5 that generalizes Figures 4.3 and 4.4.

```

Function isSatisfied( $p_k, C_k^e.CV$ ) is
    if isOpen( $p_k$ ) then
        if toMaximize( $p_k$ ) then
            if  $p_k.LB \leq C_k^e.CV$  then
                 $\text{return true}$ 
            else                                            // Minimize
                if  $p_k.UB \geq C_k^e.CV$  then
                     $\text{return true}$ 
        else                                            // Closed property
            if  $p_k.LB \leq C_k^e.CV \wedge C_k^e.CV \leq p_k.UB$  then
                 $\text{return true}$ 
    return ( $\text{false}$ )
Function isOverfulfilled( $p_k, C_k^e.CV$ ) is
    if toMaximize( $p_k$ ) then
        if  $C_k^e.CV > p_k.UB$  then
             $\text{return true}$ 
        else                                            // Minimize
            if  $C_k^e.CV < p_k.LB$  then
                 $\text{return true}$ 
    return ( $\text{false}$ )
Function isCovered( $p_k, C_k^e.LB, C_k^e.UB$ ) is
    if  $C_k^e.LB \leq p_k.UB \wedge C_k^e.UB \geq p_k.LB$  then
         $\text{return true}$ 
    else
         $\text{return (false)}$ 

```

Algorithm 4: Range matching capabilities to requirements about atomic, broker-side, or complex generic properties as described on page 84. Algorithm uses Algorithm 3; interdependent properties are checked in Algorithm 5.

```

Function RangeMatchesNonInterdependent( $X_i^e$ ,  $\text{NOM}_{X_i^e}$ ) is
    Result:  $CAND_{X_i^e}$ ,  $\overline{CAND}_{X_i^e}$ ,  $X_i^e.\text{state} \in \{\text{satisfied}, \text{satisfiable}, \text{unsatisfied}\}$ 
1       $X_i^e.\text{state} \leftarrow \text{unsatisfied}$                                 // Unsatisfied till proven otherwise
2       $\text{ADAPTATIONPLAN}_{X_i^e} \leftarrow \emptyset$                          // Start with empty adaptation plan
3      foreach  $CP_l^e \in \text{NOM}_{X_i^e}$  do
4           $\overline{\{P\}}_{i,l}, \{P\}_{i,l}, \underline{\{P\}}_{i,l} \leftarrow \emptyset$            // Which properties does  $CP_l^e$  satisfy?
5           $CAND_{X_i^e}, \overline{CAND}_{X_i^e} \leftarrow \emptyset$ 
6          foreach  $p_k \in X_i^e \wedge \neg \text{isInterdependent}(p_k)$  do
7               $C_k^e \leftarrow \Theta(p_k) \in CP_l^e$ 
8              if  $\text{isSatisfied}(p_k, C_k^e.CV)$  then                                // Already satisfied by  $C_k^e.CV$ 
9                   $p_k \rightarrow \{P\}_{i,l}$ 
10             else
11                 if  $\text{isOverfulfilled}(p_k, C_k^e.CV)$  then
12                     if  $\text{isClosed}(p_k)$  then
13                          $p_k \rightarrow \underline{\{P\}}_{i,l}$                                 // Overfulfilling illicit for closed intervals
14                     else                                                 // Could we satisfy by adaptation?
15                         if  $\text{isCovered}(p_k, C_k^e.LB, C_k^e.UB)$  then
16                              $p_k \rightarrow \overline{\{P\}}_{i,l}$ 
17                         else
18                              $p_k \rightarrow \underline{\{P\}}_{i,l}$ 
19
20             if  $\overline{\{P\}}_{i,l} \neq \emptyset$  then                                // At least one property is only satisfiable
21                  $CP_l^e \rightarrow \overline{CAND}_{X_i^e}$                                 // We would have to adapt.
22                 foreach  $p_k \in \overline{\{P\}}_{i,l}$  do
23                      $\text{ADAPTATIONPLAN}_{X_i^e} \leftarrow \text{addAllAvailableActions}(\Theta(p_k), CP_l^e)$ 
24                     if  $X_i^e.\text{state} \neq \text{satisfied}$  then
25                          $X_i^e.\text{state} \leftarrow \text{satisfiable}$ 
26
27             else
28                 if  $|\{P\}_{i,l}| = |X_i^e| - |\{\text{isInterdependent}(p_k)\}|$  then // All non-interdep. prop. satisfied
29                      $CP_l^e \rightarrow CAND_{X_i^e}$ 
30                      $\text{ADAPTATIONPLAN}_{X_i^e} \leftarrow \text{add}(\text{routingFrom}(P_l), CP_l^e)$ 
31                      $X_i^e.\text{state} \leftarrow \text{satisfied}$ 

```

Algorithm 5: Checking requirements about *alternatives* (interdependent) using Algorithm 3 as described in the example on page 84.

```

Function CheckAlternativesRequirement( $X_i^e$ ,  $CAND_{X_i^e}$ ,  $\overline{CAND}_{X_i^e}$ ) is
    Result:  $X_i^e.\text{state} \in \{\text{satisfied}, \text{satisfiable}, \text{unsatisfied}\}$ 
1      if  $\text{isSatisfied}(p_k, |CAND_{X_i^e}|)$  then                                // Already enough suitable publishers
2          return satisfied
3      else
4          if  $\text{isSatisfied}(p_k, |CAND_{X_i^e}| + |\overline{CAND}_{X_i^e}|)$  then
5              return satisfiable                                              // enough capable publishers to adapt
6          else                                                               // Not possible, even with adaptation, as
7              return unsatisfied                                         // there are not enough capable publishers

```

Algorithm 6: Determine the least expensive (minimal) adaptation plan for Algorithm 1 as discussed on page 89.

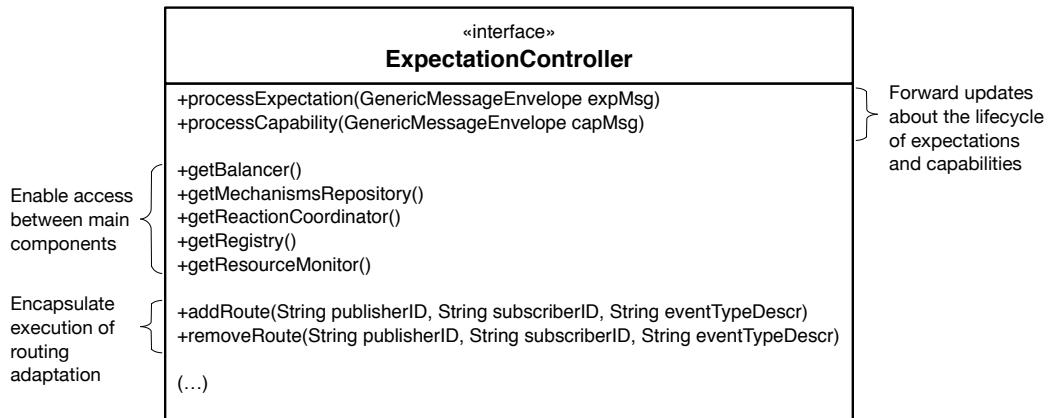
```

Function getAdaptationPlanMinimalCosts( $X_i^e$ , ADAPTATIONPLAN $X_i^e$ ) is
    Result:  $\overline{\text{ADAPTATIONPLAN}_{X_i^e}} = \{a_1, \dots, a_n\}$  with minimal costs
    foreach  $CP_l^e \in \text{ADAPTATIONPLAN}_{X_{i,new}^e}$  do
        foreach  $p_k \in \overline{\{P\}}_{i,l}$  do  $\text{// What has to be done for } CP_l^e?$ 
             $C_k^e \leftarrow \Theta(p_k) \in CP_l^e$ 
            if hasToBeIncreased( $p_k, C_k^e$ ) then  $\text{// Increase}$ 
                 $\vec{A}_k \leftarrow \text{getIncreasingActions}(p_k)$ 
            else  $\text{// Decrease}$ 
                 $\vec{A}_k \leftarrow \text{getDecreasingActions}(p_k)$ 
                        $\text{// Select least expensive action}$ 
             $\Omega \leftarrow \emptyset$ 
            foreach  $a_o \in \vec{A}_k$  do
                           if  $\Omega = \emptyset$  then
                     $\Omega = a_o$ 
                           else
                                 if getTotalCosts( $a_o, C_k^e, p_k.LB, p_k.UB$ ) < getTotalCosts( $\Omega, C_k^e, p_k.LB, p_k.UB$ ) then
                         $\Omega = a_o$ 
                      
             $\overline{\text{ADAPTATIONPLAN}_{X_i^e}} \leftarrow \text{add}(\Omega)$ 
                  
    return  $\overline{\text{ADAPTATIONPLAN}_{X_i^e}}$ 

Function hasToBeIncreased( $p_k, C_k^e$ ) is
    Result: true  $\vee$  false
    if toMaximize( $p_k$ ) then  $\text{// Maximize: test } p_k.LB$ 
        if  $p_k.LB - C_k^e.CV < 0$  then
            return false
        else
            return true
    else  $\text{// Minimize: test } p_k.UB$ 
        if  $p_k.UB - C_k^e.CV > 0$  then
            return true
        else
            return false

```

A.3 Reference Architecture: APIs and Code Examples



(a) Key methods of the **ExpectationController** interface implemented by the MOM as discussed on page 113.



(b) Interface to be implemented by each broker in a DEBS when participating in ASIA as described on page 119.

Figure A.1.: Broker interfaces for controllers.

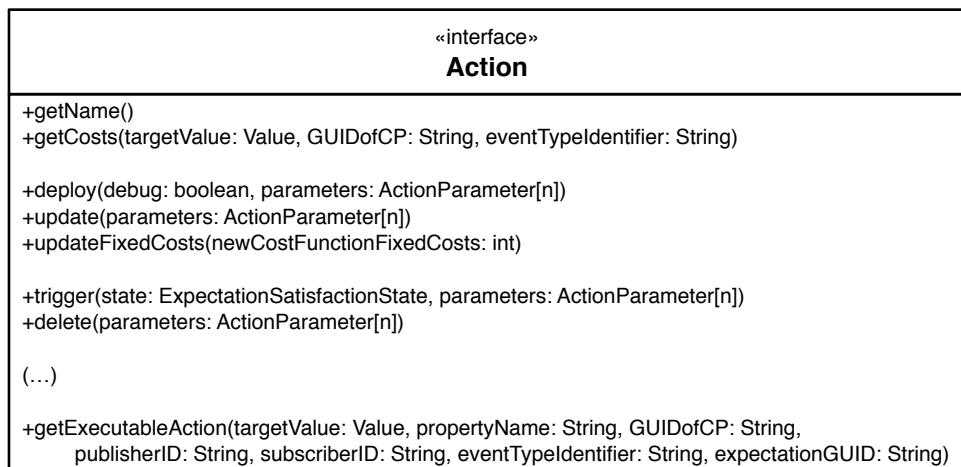


Figure A.2.: Interface any prototype of an action has to implement as described on page 116.

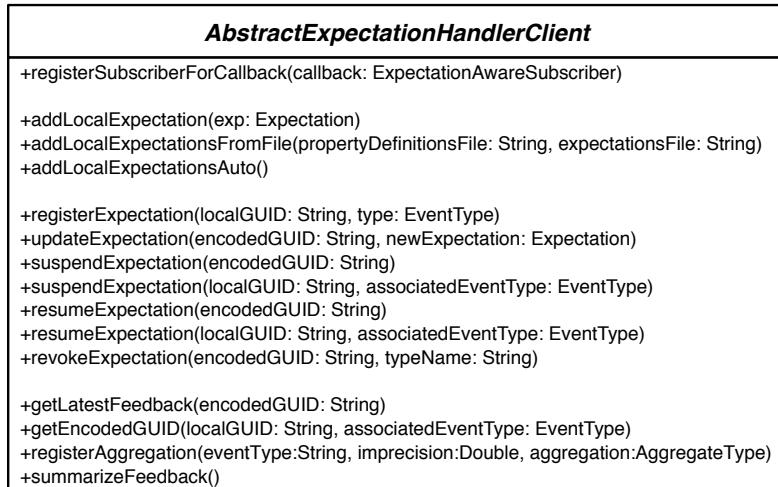


(a) Interface subscribers implement to be triggered as described on page 119.

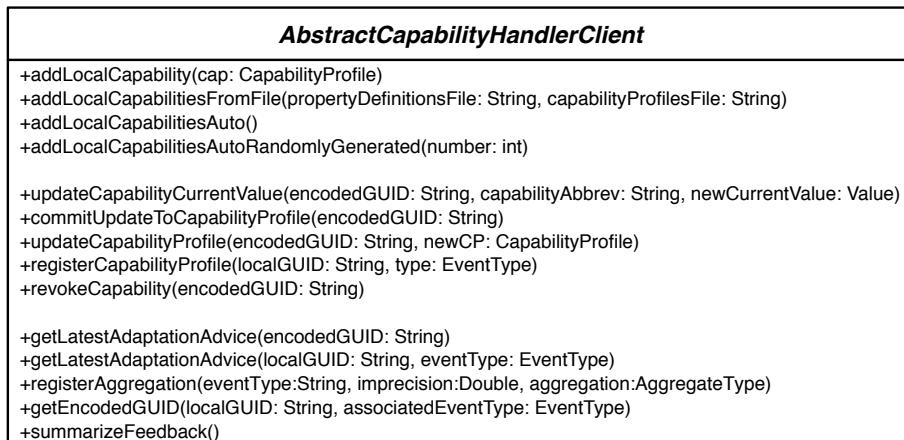


(b) Interface to be implemented by publishers as described on page 120.

Figure A.3.: Callback interfaces for subscribers and publishers.



(a) API for handling the lifecycle of expectations, provided as abstract class.



(b) API for managing the lifecycle of capabilities.

Figure A.4.: APIs of client handlers provided as abstract classes as discussed on page 119.

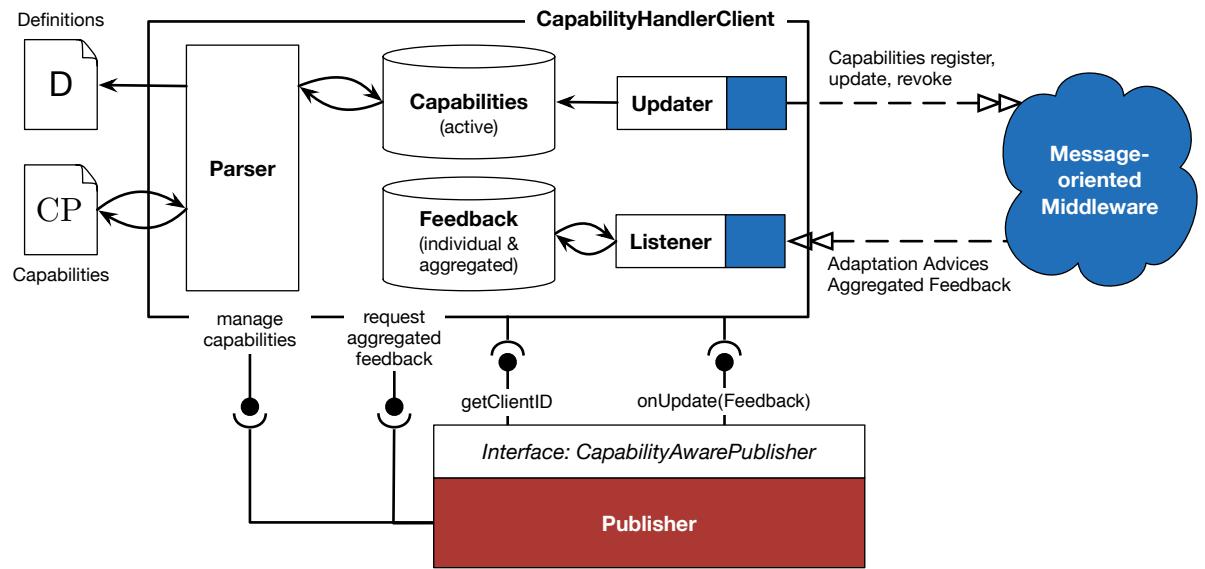


Figure A.5.: Handler for publishers to manage their capabilities and handle feedback as discussed in Section 5.1.3.

Access to	BrokerPluginSupport
Notification processing	<pre>+send(producerExchange: ProducerBrokerExchange, messageSend: Message) +preProcessDispatch(messageDispatch: MessageDispatch) +postProcessDispatch(messageDispatch: MessageDispatch)</pre>
Types & topics lifecycle	<pre>+addBroker(connection: Connection, info: BrokerInfo) +addConsumer(context: ConnectionContext, info: ConsumerInfo) +addProducer(context: ConnectionContext, info: ProducerInfo) +removeBroker(connection: Connection, info: BrokerInfo) +removeConsumer(context: ConnectionContext, info: ConsumerInfo) +removeProducer(context: ConnectionContext, info: ProducerInfo)</pre>
Population lifecycle	<pre>+addDestination(context: ConnectionContext, destination: ActiveMQDestination, createTemporary: boolean) +removeDestination(context: ConnectionContext, destination: ActiveMQDestination, timeout: long) +removeSubscription(context: ConnectionContext, info: RemoveSubscriptionInfo)</pre>

Figure A.6.: API offered to plugins by the BrokerPluginSupport class to access the broker state as discussed on page 126.

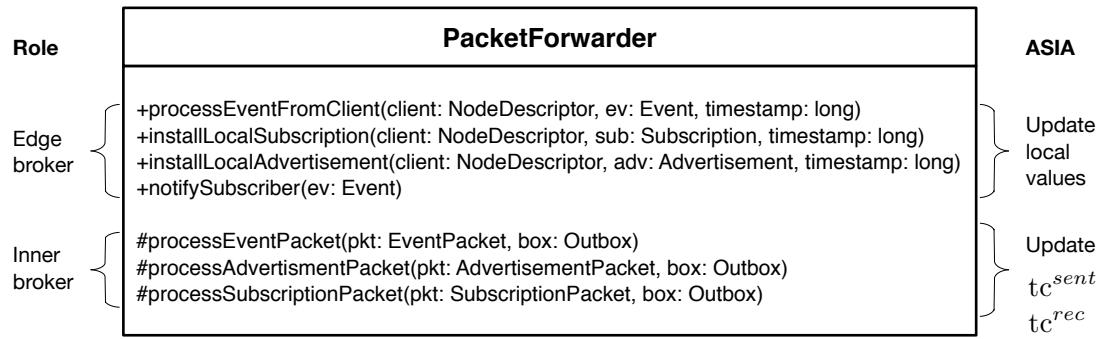


Figure A.7.: Joinpoints provided in REDS for ExpectationController and ASIAController as discussed on page 131.

Listing A.1: RateControllerDispatchPolicy for injecting rateController instances into ActiveMQ as discussed on page 127.

```

1 public class RateControllerDispatchPolicy implements DispatchPolicy {
2
3     final private ExpectationController callback;
4     final private Hashtable<String,Hashtable<String,RateController>> buffers;
5     final private ActiveMQDestination destination;
6
7     public RateControllerDispatchPolicy(ExpectationController controller,
8             ActiveMQDestination destination) {
9         this.callback = controller;
10        this.destination = destination;
11        this.buffers = new Hashtable<String,Hashtable<String,RateController>>();
12        System.out.println("[DispatchPolicy] Initialized for "+this.destination);
13    }
14
15    @Override
16    public boolean dispatch(MessageReference node, MessageEvaluationContext msgContext, List
17            <Subscription> consumers) throws Exception {
18
19        String clientID,eventTypeDescriptor,publisherID;
20        Message message = (Message) msgContext.getMessage();
21        publisherID = message.getStringProperty("publisherID");
22
23        int count = 0; //Required by DispatchPolicy
24        eventTypeDescriptor = EventType.trimActiveMQDestinationNames(message.getJMSDestination
25                () .toString());
26
27        synchronized(consumers) {
28
29            for (Subscription sub:consumers) {
30
31                clientID = sub.getContext().getClientId();
32
33                if (sub.getConsumerInfo().isBrowser()) continue;// Do not deliver to browsers
34
35                if (!sub.matches(node, msgContext)) {
36                    sub.unmatched(node); // Only dispatch to interested subscriptions
37                    continue;
38                }
39            }
40        }
41    }

```

```

36
37     if (this.callback.getMechanismRepository().is ActiveContentAggregation(
38         eventTypeDescriptor, clientID, publisherID) {
39
40         if (!this.buffers.containsKey(clientID))
41             this.buffers.put(clientID, new Hashtable<String, RateController>());
42
43         if (!this.buffers.get(clientID).containsKey(publisherID)) {
44             this.buffers.get(clientID).put(publisherID,
45                 new RateController(
46                     this.callback,
47                     eventTypeDescriptor,
48                     clientID,
49                     publisherID,
50                     true)
51             );
52         }
53
54         //Get aggregated notification from rateController
55         Message namsq = (org.apache.activemq.command.Message) this.buffers.get(clientID).
56             get(publisherID).bufferEventForAggregation(message);
57
58         //nmsg is NULL if there is nothing to dispatch for now...
59         if (nmsg == null) continue;
60
61         sub.add(namsq);
62     } else sub.add(node); //No rateController defined
63
64     count++;
65 }
66 }
67 return count > 0;
68 }
69 }
```

Listing A.2: Injecting RateConrollerDispatchPolicy in ActiveMQ as discussed on page 127.

```

1 @Override
2 public Destination addDestination(ConnectionString context, ActiveMQDestination
3     destination, boolean createIfTemporary) throws Exception {
4
5     (...)

6     if (this.getMechanismRepository().isRegistered("rate", "rateController") &&
7         !this.dispatchPoliciesForRateReduction.containsKey(destinationName)) {
8
9         this.dispatchPoliciesForRateReduction.put(destinationName, new
10            RateControllerDispatchPolicy(this, destination))
11
12         PolicyEntry policy = new PolicyEntry();
13         policy.setDispatchPolicy(this.dispatchPoliciesForRateReduction.get(destinationName));
14         policy.setDestination(destination);
15         this.getBrokerService().getDestinationPolicy().put(destination, policy);
16     }
17     return super.addDestination(context, destination, createIfTemporary);
18 }
```

```

<definition>  |= <abbrevdef> <titledef> <improvementdef> <unitdef> <valuesdef> | ε
<abbrevdef>  |= xproperty.<pname>.abbrev=<pname>
    <titledef> |= xproperty.<pname>.title=<ptitle>""
<improvementdef> |= xproperty.<pname>.improvement=<pimprovement>
    <unitdef> |= xproperty.<pname>.unit=<punit>
    <valuesdef> |= <valuestyledatatype> <valuestyledef>
    <valuestyledef> |= <rangestyle> <rangemin> <rangemax> | <liststyle> <list>
<valuestyledatatype> |= xproperty.<pname>.datatype=<datatype>
    <rangestyle> |= xproperty.<pname>.value.style=Range
    <rangemin> |= xproperty.<pname>.value.minimum=value
    <rangemax> |= xproperty.<pname>.value.maximum=value
    <liststyle> |= xproperty.<pname>.value.style=List
        <list> |= xproperty.<pname>.value.list=values
    <values> |= values,value | value
    <value> |= value<char> | <char> | ε
    <char> |= A...Z | a...z | 0...9 | - | . | ,
<pimprovement> |= MAXIMIZE | MINIMIZE
<datatype> |= Binary | Double | Float | Integer | Long | String
<pname> |= pname, name of the property
<ptitle> |= textual description of the property
<punit> |= unit, e.g., nanoseconds, events per second

```

(a) Backus-Naur Form (BNF) of a property definition.

```

1 xproperty.confidence.title="Confidence of detection"
2 xproperty.confidence.abbrev=confidence
3 xproperty.confidence.improvement=MAXIMIZE
4 xproperty.confidence.value.style=range
5 xproperty.confidence.value.datatype=Integer
6 xproperty.confidence.value.minimum=0
7 xproperty.confidence.value.maximum=100
8
9 xproperty.trust.title="Trustworthiness"
10 xproperty.trust.abbrev=trust
11 xproperty.trust.improvement=MAXIMIZE
12 xproperty.trust.value.style=list
13 xproperty.trust.value.datatype=String
14 xproperty.trust.value.list=none,low,medium,high
15 ...

```

(b) Examples of generic properties *confidence of detection* and *trustworthiness*.

Figure A.8.: Generic properties definition in BNF notation and line-based syntax.

A.4 FINCoS: Extensions and Experimental Setup

This section describes our extensions to the FINCoS benchmarking tool that we have introduced in Section 5.3 and used for our evaluation in Section 6.2.

We have customized and extended FINCoS to handle expectations, capabilities as well as the feedback provided by a middleware that supports expectations. Subscribers can now associate one or more expectations with a subscription and can change that behavior over the duration of an experiment with or without changing their subscriptions. Publishers can associate capabilities with their advertisements, change them at runtime, and – first and foremost – they now adapt their publication behavior at runtime based on individual feedback given by the MOM.

In particular, we have added the following elements to FINCoS as shown in Figure 5.27 (blue):

- *Adapters for ActiveMQ and REDS.* Adapters for drivers and sinks allow FINCoS to interact with a specific MOM. We have implemented adapters for drivers and sinks to interact with ActiveMQ and REDS based on the interfaces provided by FINCoS to be implemented by custom adapters. We implemented a separate adapter for ActiveMQ as the provided adapter for the Java Message Service (JMS) caused issues with dynamic topics on ActiveMQ.
- *Expectations and capabilities client libraries.* We have integrated the client libraries for handling expectations, capabilities and feedback we have developed for ActiveMQ and REDS and described in Section 5.1.3. Each instance of a subscriber or publisher maintains its own local repository of expectations (or capabilities, respectively). The client libraries handle the loading of property definitions and predefined expectations or capability profiles into FINCoS and manage the platform-specific communication of lifecycle messages. Furthermore, they process the individual and aggregated feedback given to subscribers and publishers alike. For publishers, they also trigger the self-adaptation logic of each publisher instance as drivers now implement the `CapabilityAwarePublisher` interface and can process adaptation advices sent as individual feedback. In addition, we have also implemented support for aggregated feedback about the dynamics and population of the EBS at runtime. Both subscribers and publishers can request updates about aggregated metrics such as `publisherCount`, `subscriberCount`, or `publicationRate`. For each request, an individual imprecision can be set. Aggregation requests are registered, updated or revoked at the start of each phase by both subscribers and publishers.
- *Phases for subscribers.* In FINCoS V2.4.2, publishers can change their behavior with every phase but subscribers can not. Thus, we have implemented the concept of phases for subscribers as this enables us to simulate requirements that change at runtime. Subscribers can now change their subscriptions, their aggregation requests for aggregated feedback and their expectations during the course of an experiment: as for publishers, multiple phases can be defined for each subscriber. Subscriptions and expectations are registered, revoked or updated at the start of each phase.
- *Self-adaptation based on feedback.* We have added self-adaptive behavior to publishers. At runtime, a publisher can now autonomously adapt the sampling rate, accuracy and artificial latency¹ of its publications based on the adaptation advises given by the MOM to FINCoS about the capabilities registered by that individual publisher. The current values of the

¹ We have enabled FINCoS to adjust the timestamp of a notification by an amount of milliseconds before publishing it. This artificial latency can be configured for a specific type of notification per publisher.

respective capabilities are also updated locally and reported back to the MOM as updates (cf., Section 3.4.3).

- *Detailed traces.* In addition to the traces recorded by FINCoS itself, we have added the collection of more detailed traces for both subscribers and publishers. This allows us to log additional attributes and metadata for every notification that has been sent by publishers or received by subscribers. While the traces logged by FINCoS contain only the information required by FINCoS to replay the workload, our detailed traces also include the separately received feedback from the MOM as well as timestamps of important events (pinpoints) that we use to synchronize information gathered from different participants.

A.4.1 Test Harness for Automated Testing

We have developed a test harness to automate experiments run with FINCoS and our prototypes. In this regard, we have implemented a console-only version of FINCoS that does not require a graphical user interface and is used by our test harness to run unsupervised experiments on remote hosts.

As illustrated in Figure A.9, the test harness is designed to automatically execute test plans. A test plan defines a sequence of single test runs. A single test run describes a single experiment and has two phases: an execution phase and an analysis phase.

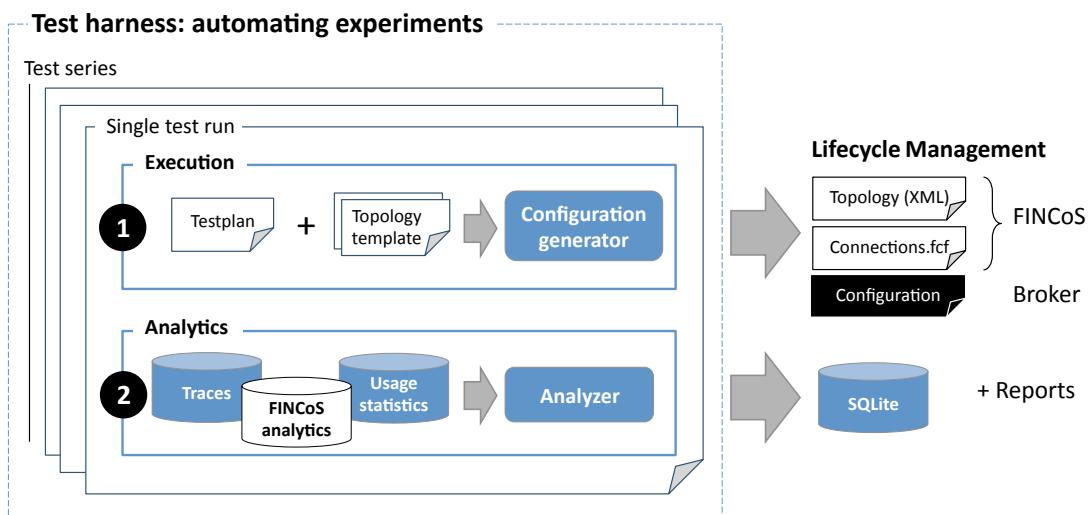


Figure A.9.: Test harness to automate executing and analyzing series of single test runs.

Automating the execution of a single run is done by updating the configuration files for FINCoS and the system under test based on a template and a testplan. A topology template describes the participants of a setup, their behavior and their relationships in a platform-independent syntax; a test plan describes a list of changes to that template for each single test run. For example, the publishers and subscribers populating the heterogeneity scenario described earlier are defined in a topology template while the testplan defines the different heterogeneity levels by configuring an increasing number of publishers to send unsuitable data. A configuration generator uses this input to generate platform-specific configuration files for FINCoS and the system under test for each single test run. The test harness then distributes these configurations, starts the resource monitoring on all hosts, the system under test, and FINCoS. After completing a single test run, the test harness collects all traces and monitoring information logged during the experiment,

performs a clean shutdown of all participants and finally analyzes the collected data using scripts written in Java and R². The results are stored in a database together with the raw log data. This process is automatically repeated for all test runs defined in a test series.

A.4.2 Test Setup

The test environment for experiments with FINCoS is shown in Figure A.10. All Virtual Machines (VMs) being part of an environment are controlled by *Vagrant*³ [199] V1.6.3 and provisioned using Oracle *VirtualBox*⁴ V4.3.10. They all run on a single host.

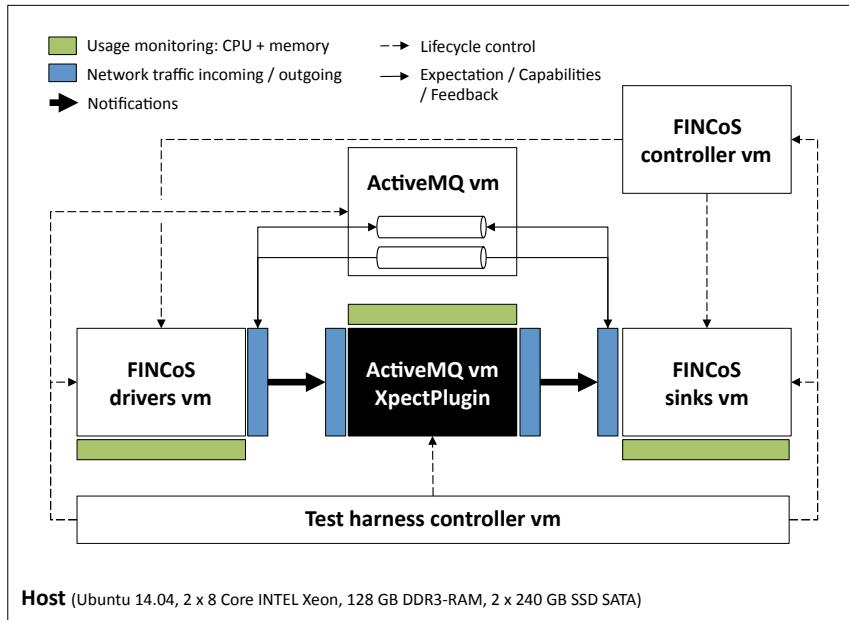


Figure A.10.: Test environment using six virtual machines running on the same host. Resource utilization is measured separately for publishers, subscribers and ActiveMQ.

Each host runs Ubuntu Linux 14.04 (Trusty Tahr) and is equipped with 2 x 8 Core INTEL Xeon E5-2650 2000MHz 15M Cache, 128 GB DDR3-RAM ECC PC1600 (8 x 16 GB DIMM), and 2 x 240 GB SSD SATA. Clocks on all VMs are automatically synchronized using the Network Time Protocol (NTP)⁵. The VMs on a single host form a virtual private network with static network addresses. This makes the whole setup portable and allows running multiple clones of a test environment on different hosts in parallel.

As shown in Figure A.10, a dedicated VM is used by our test harness to execute test plans and coordinate the lifecycle of the participants. For performance reasons and to measure the resource utilization of publishers and subscribers in isolation, we are running them on separate VMs while the FINCoS controller is hosted on a third VM. The system under test is hosted on one VM. In our setup for ActiveMQ we use two VMs: one hosting an ActiveMQ instance with an *ExpectationController* while a separate instance of ActiveMQ is hosted on another VM. The separate ActiveMQ instance runs without any plugin and just provides four topics used to exchange expectations, capabilities and feedback between clients and

² <https://www.r-project.org/>

³ <https://www.vagrantup.com/>

⁴ <https://www.virtualbox.org/>

⁵ <http://www.ntp.org/>

MOM (`control.expectations`, `control.capabilities`, `control.expectations.feedback`, and `control.capabilities.feedback`).

The VMs have different configurations depending on their role: VMs running FINCoS and the test harness have four virtual cores and 34 GB RAM each while the VMs running the system under test have one virtual CPU and 24 GB RAM each.

FINCoS does not measure the utilization of resources such as CPU utilization, network traffic, memory usage or disk space. Thus, we have added the necessary monitoring capabilities to our test harness. The test harness uses `dstat` to measure the resource utilization of the VMs hosting subscribers and publishers as well as each VM running a broker that is part of an EBS or DEBS.

A.4.3 Anatomy of a Single Test Run

As shown in Figure A.11, each single run has a setup stage followed by a measurement stage. The total duration of a single run is 60 seconds as additional experiments have shown that this duration is a good trade-off between the unambiguousness of the measured results and the effort necessary for analysis. Additional experiments with longer durations of the measurement stage have shown no impact on the measured results. Longer durations, however, aggravate the analysis by several orders of magnitude due to the high increase in measured data.

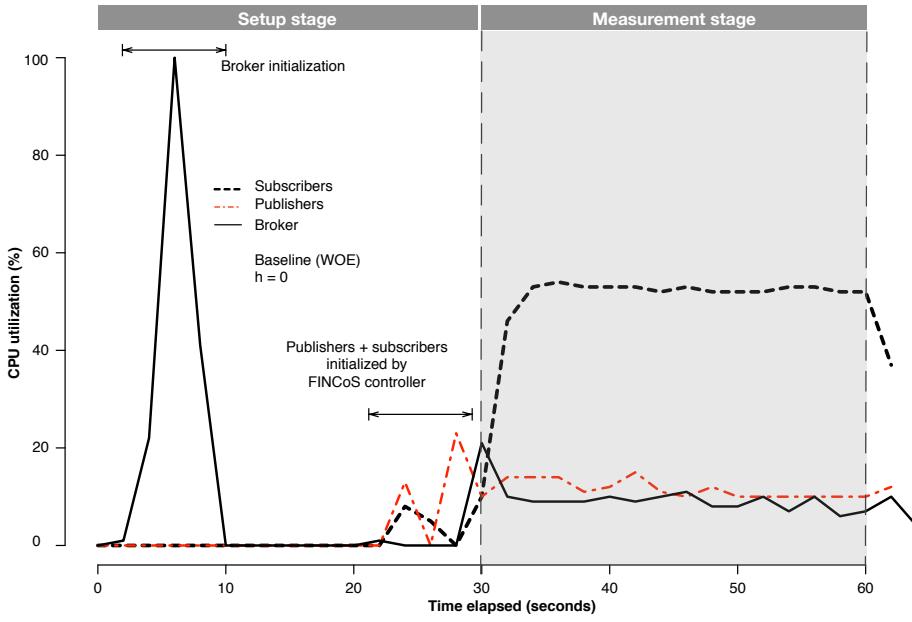
We use a test run from the baseline scenario without any plugin running on ActiveMQ and a heterogeneity of 0% to discuss the characteristics of a single run and show what is used as a Key Performance Indicator (KPI) when comparing different scenarios.

During the setup stage, all participants are started by the test harness while publishers and subscribers are initialized by the FINCoS controller. Starting ActiveMQ results in a high CPU utilization that is independent of the scenario or plugin being used by ActiveMQ as shown in Figure A.11a. Initializing subscribers and publishers on the other hand does not impact the CPU utilization but causes spikes in network traffic as the workload definition and phase configurations have to be deployed on all VMs hosting subscribers and publishers.

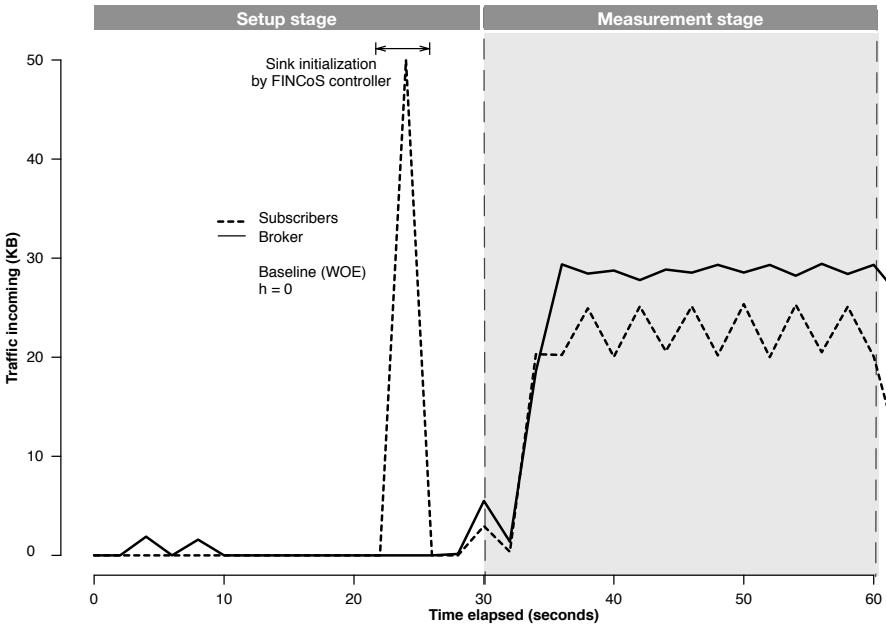
The measurement stage is started by the FINCoS controller triggering publishers and subscribers. Advertisements, subscriptions, expectations and capabilities are registered at the MOM first followed by the notifications sent by publishers, processed by the MOM and delivered to subscribers again. The values observed in this stage differ as they depend on the scenario and level of heterogeneity. Please note that several phases defined for publishers or subscribers in FINCoS can take place in the measurement stage.

For each single test run, we measure the average resource utilization in terms of CPU utilization and incoming traffic during the measurement stage. We use these measurements as KPIs to compare different scenarios with each other. We also log the timestamps for the beginning and ending of each phase in FINCoS as well as the start and end of the measurement stage.

Furthermore, we calculate the conformance of each notification received by the subscriber with the requirements expressed in X_5^{pos} . The cumulative fidelity as the sum of all these values reflects the overall satisfaction of the subscriber at the end of a single test run (cf., Section 3.3.3).



(a) CPU utilization during a single run.



(b) Traffic measured during a single run.

Figure A.11.: Anatomy of a single run using FINCoS: example plots showing the resource utilization in terms of incoming network traffic and CPU utilization for subscribers, publishers and MOM (ActiveMQ). Numbers are measured over the duration of a single FINCoS experiment in the baseline scenario (Without Expectations (WOE))

A.5 Drill Down Data for Heterogeneity Scenario with Surplus Publishers

Heatmaps Figures A.12 and A.13 complement the discussion of a scenario with surplus publishers in Section 6.2.3.

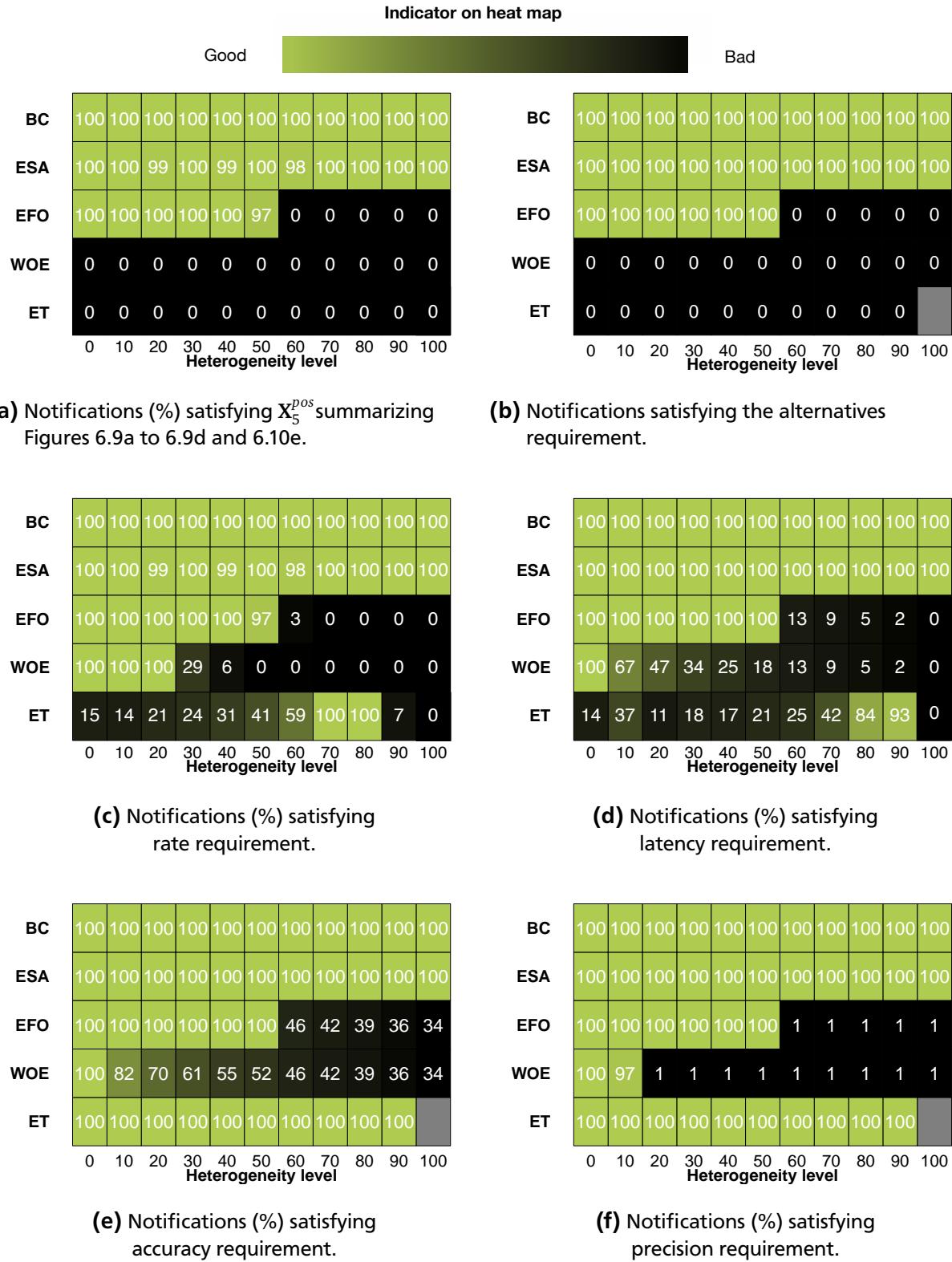
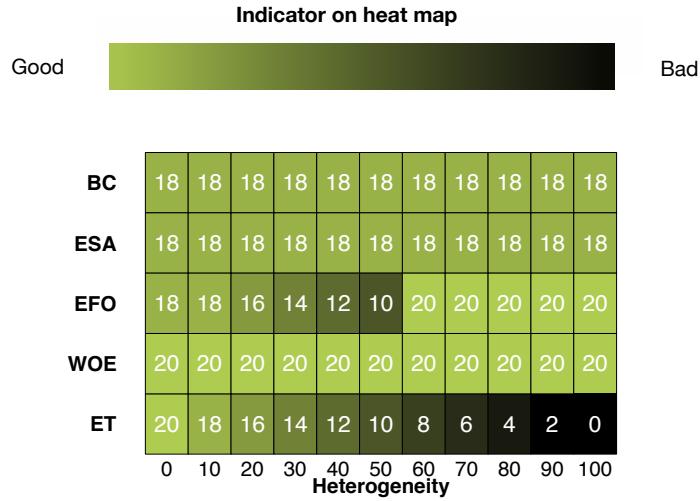
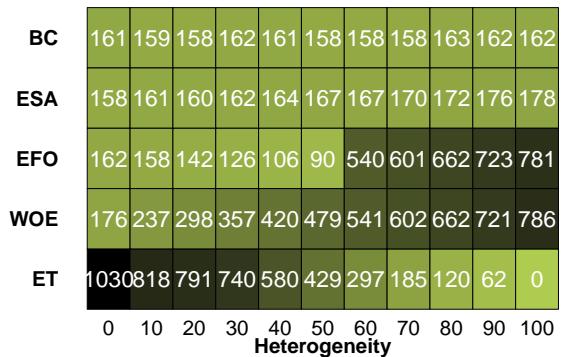


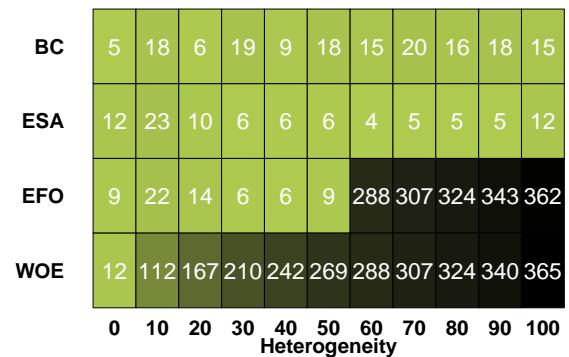
Figure A.12.: Drill down: conformance with requirements for surplus publishers.



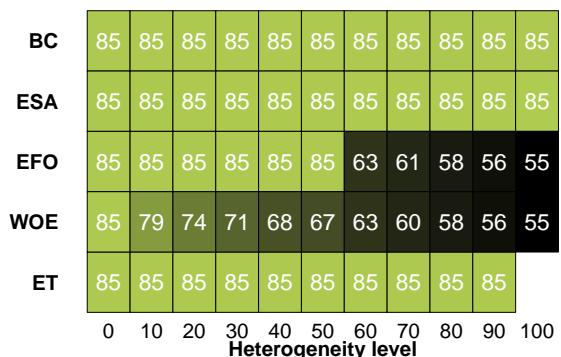
(a) Number of publishers data is received from.



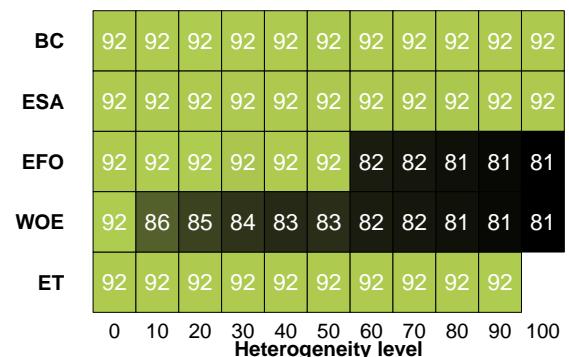
(b) Rates received (notifications/sec).



(c) Average latency of notifications (ms).



(d) Average accuracy of data received (%).



(e) Average precision of data received (%).

Figure A.13.: Drill down: data for scenario with surplus publishers.

A.6 jms2009-PS Extensions and Experimental Setup

We have extended jms2009-PS to include expectations and capabilities into its workload. The additional load introduced by expectations and capabilities is tied to the benchmark's workload and scales with it. We generate sets of random properties with random definitions. Based on these, our test harness generates expectations and capabilities with upper and lower bounds based on a uniform distribution. The seed can be configured. Participants express subscriptions on different product families by using the IN selector provided by JMS. This configuration is similar to Scenario II described in [368].

We introduce the *dynamics* parameter to simulate changing contexts for publishers and subscribers that result in changing requirements and capabilities at runtime. The dynamics parameter denotes how frequently each participant is changing.

We had to compensate for the major performance improvements achieved for ActiveMQ V5.10 in comparison to V5.4. This compensation is necessary to be able to recreate the results published in [368, 395] for ActiveMQ V5.4 with the more recent version V5.10 used in our setup. In particular, we had to adjust the load-balancing factor used by SPECjms2007 to share the load between all instances of publishers and subscribers that are generated for each JMS topic. Without adjusting the configuration parameters used in [368, 395], the benchmark had not been able to uphold the necessary sampling rates at the publisher side and handle all processed notifications at the subscriber side in time. Consequently, a single run of jms2009-PS had not been successful because the benchmark had become the bottleneck while ActiveMQ had not been exhausted.

A.6.1 Test Harness for Automated Testing

We use the same test harness we have described in Section A.4 to automate testing with jms2009-PS. As for FINCoS, the test harness generates the necessary configuration files for jms2009-PS and ActiveMQ. The configuration is automatically distributed to the different hosts running controller, satellites, and the ActiveMQ broker under test. The test harness also collects all data logged by both jms2009-PS and the system under test.

The test setup for experiments with jms2009-PS is shown in Figure A.14. All VMs being part of an environment are controlled by *Vagrant*⁶ [199] V1.6.3 and provisioned using Oracle *VirtualBox*⁷ V4.3.10. They all run on a single host with the configuration already described in Appendix A.4

As for FINCoS, a dedicated VM is used to execute test plans and coordinate the lifecycle of jms2009-PS and ActiveMQ. We have distributed the satellites across multiple VMs where each location resides on a separate VM. Each agent has its own Java Virtual Machine (JVM).

The system under test is hosted on one VM. In our setup for ActiveMQ, we use two VMs: one hosting an ActiveMQ instance with an `ExpectationController` while a separate instance of ActiveMQ is hosted on another VM. The separate ActiveMQ instance runs without any plugin and just provides four topics used to exchange expectations, capabilities and feedback between clients and MOM (`control.expectations`, `control.capabilities`, `control.expectations.feedback`, and `control.capabilities.feedback`).

The VMs have different configurations depending on their role: VMs running jms2009-PS and the test harness have four virtual cores and 34 GB RAM each while the VMs running the system

⁶ <https://www.vagrantup.com/>

⁷ <https://www.virtualbox.org/>

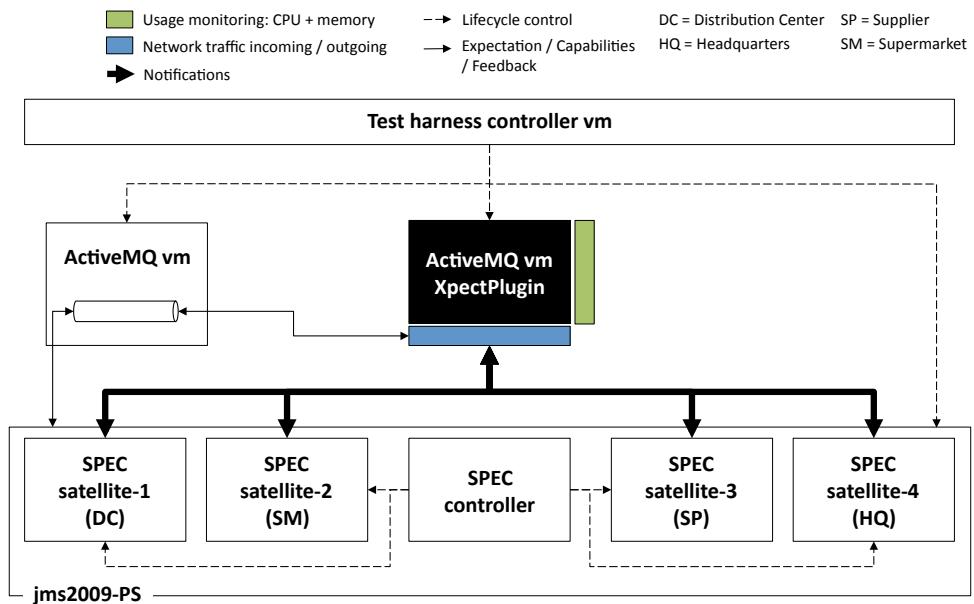


Figure A.14.: Test setup for evaluating the scalability of our approach using an extended version of jms2009-PS that is based on SPECjms2007.

under test have one virtual CPU and 24 GB RAM each. The VM running the ActiveMQ instance is dimensioned based on the configuration used in the published SPECjms 2007 evaluation results⁸ for ActiveMQ V5.4 by Sachs et al..

jms2009-PS does not measure the utilization of resources such as CPU utilization, network traffic, memory usage, or disk space. Thus, we have added the necessary monitoring capabilities.

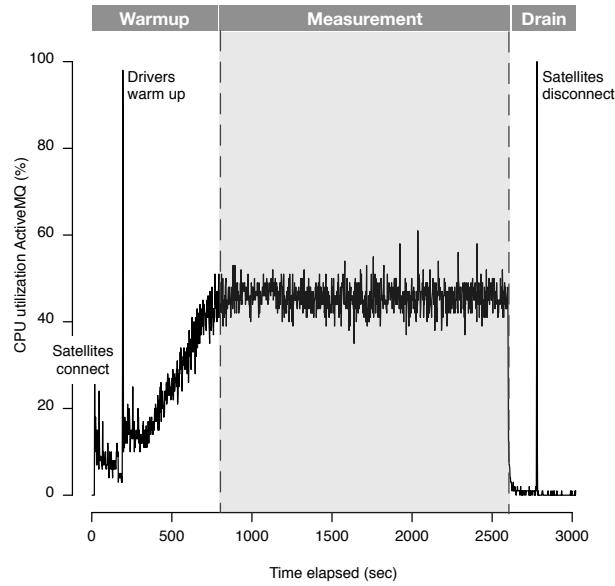
A.6.2 Anatomy of a Single Run

Each jms2009-PS run has 3 stages: warmup period, measurement period and drain period as shown in Figure A.15. The duration of the measurement phase is 30 minutes, which is the default duration recommended by SPEC. The total duration of a run including warmup and cool down phases is 45 minutes.

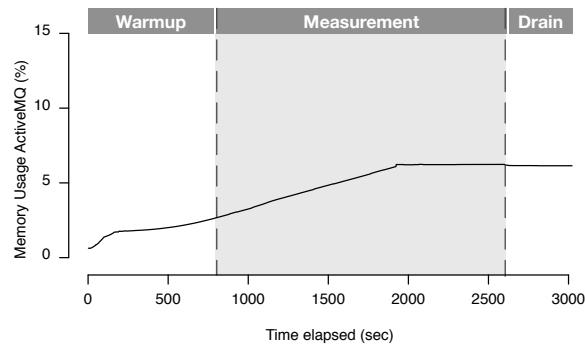
Please note that the first peak in the warmup phase is caused by the jms2009-PS satellites connecting to the ActiveMQ broker while the next peak is caused by the satellites all advertising and subscribing at the start of the warmup phase; the last peak is caused by the satellites all disconnecting as one from the broker after the measurement phase is completed. These peaks are not affected by the use of expectations or by any scaling factor introduced by us.

For the measurement phase of each run, we measure the average resource utilization in terms of CPU utilization and incoming/outgoing traffic in addition to the delay and throughput as monitored by jms2009-PS itself. We can resort to using averages here as the load on the system is meant to be stable during the measurement phase. Memory is constantly increasing over time as we configure ActiveMQ to use an in-memory database for persistence while notifications do not become obsolete.

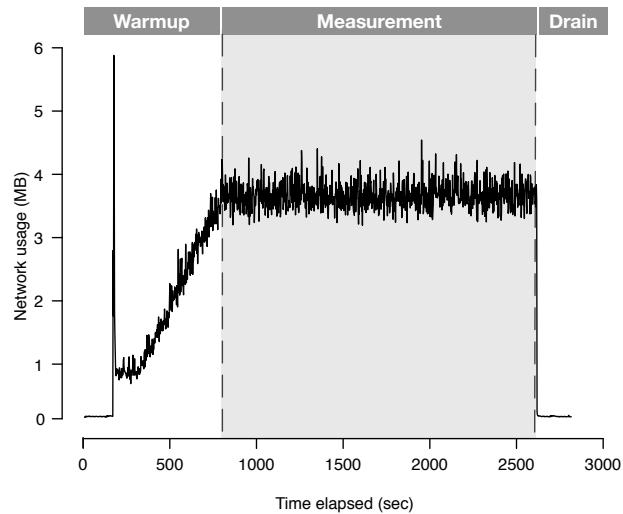
⁸ <https://www.spec.org/jms2007/results/res2010q3/jms2007-20100802-00022.html>



(a) CPU utilization.



(b) Memory consumption.



(c) Traffic incoming.

Figure A.15.: Anatomy of a single run of jms2009-PS on a bare ActiveMQ.

A.7 Regression Tables jms2009-PS Benchmark Results

The Tables A.5 to A.10 detail the results discussed in Section 6.3.4.

Table A.5.: Linear regression analysis results for our prototype when scaling the population or throughput for fixed d=0.01 and fixed p=5.

KPI	Horizontal BASE _h				Vertical BASE _v			
	α_h	β_h	R ²	adj. R ²	α_v	β_v	R ²	adj. R ²
CPU utilization	15.7	3.12	1.00	1.00	14.2	0.68	0.99	0.99
Memory usage	26.5	0.40	0.88	0.87	32.2	0.03	0.40	0.37
Network traffic	0.1	0.27	1.00	1.00	0.1	0.11	1.00	1.00

Table A.6.: Linear regression analysis results for our prototype when increasing the number of generic properties while keeping all other parameters fixed (d=0.01).

KPI	Horizontal BASE _h = 15				Vertical BASE _v = 50			
	α_h	β_h	R ²	adj. R ²	α_v	β_v	R ²	adj. R ²
CPU utilization	52.4	1.52	0.81	0.72	40.4	1.70	0.93	0.89
Memory usage	24.9	1.33	0.91	0.86	28.4	1.10	0.94	0.91
Network traffic	4.1	0.00	0.88	0.82	5.5	0.00	0.11	-0.33

Table A.7.: Linear regression analysis results for our prototype when simulating *moderate* and *aggressive* update frequencies; p=5, BASE_h =15, BASE_v =50.

Category	KPI	Horizontal BASE _h = 15				Vertical BASE _v = 50			
		α_h	β_h	R ²	adj. R ²	α_v	β_v	R ²	adj. R ²
Moderate	CPU utilization	64.2	67.82	0.87	0.87	50.4	103.13	0.96	0.96
	Memory usage	35.0	38.23	0.88	0.88	32.4	137.64	1.00	1.00
	Network traffic	4.2	1.75	0.48	0.48	5.4	5.40	0.81	0.81
Aggressive	CPU utilization	65.5	52.28	0.96	0.96	53.4	93.66	0.97	0.97
	Memory usage	38.7	17.24	0.73	0.73	40.8	16.27	0.73	0.73
	Network traffic	3.8	6.52	0.96	0.96	5.0	9.62	0.95	0.95

Table A.8.: Linear regression analysis results: scaling the number of generic properties for different update frequencies while keeping the population or throughput fixed.

KPI	f	Horizontal $\text{BASE}_h = 15$				Vertical $\text{BASE}_v = 50$			
		α_h	β_h	R^2	adj. R^2	α_v	β_v	R^2	adj. R^2
CPU utilization	0.001	53.8	0.71	0.83	0.81	50.1	0.08	0.32	0.21
CPU utilization	0.010	51.0	1.45	0.87	0.82	47.7	0.35	0.93	0.90
CPU utilization	0.100	55.0	2.39	0.93	0.86	65.5	-0.77	0.77	0.54
CPU utilization	0.200	68.5	1.70	0.94	0.87	78.9	-1.78	0.78	0.55
Memory usage	0.001	28.7	0.48	0.92	0.91	30.7	-0.07	0.34	0.23
Memory usage	0.010	27.3	1.09	0.89	0.85	34.1	0.53	0.43	0.24
Memory usage	0.100	33.3	1.19	0.83	0.66	45.0	-0.98	0.75	0.50
Memory usage	0.200	36.9	1.05	0.87	0.75	48.5	-1.23	0.75	0.50
Network traffic	0.001	4.1	0.00	0.13	-0.01	5.5	0.00	0.19	0.05
Network traffic	0.010	4.1	0.00	0.88	0.84	5.5	0.00	0.02	-0.30
Network traffic	0.100	4.6	-0.01	0.20	-0.61	6.3	-0.07	0.77	0.54
Network traffic	0.200	5.6	-0.07	0.65	0.30	7.5	-0.15	0.77	0.55

Table A.9.: Linear regression analysis results: scaling the population or throughput for different update frequencies while keeping the number of generic properties fixed ($p = 5$).

KPI	f	Horizontal BASE_h				Vertical BASE_v			
		α_h	β_h	R^2	adj. R^2	α_v	β_v	R^2	adj. R^2
CPU utilization	0.001	16.8	3.02	0.99	0.99	14.7	0.66	0.98	0.98
CPU utilization	0.010	15.7	3.12	1.00	1.00	14.2	0.68	0.99	0.99
CPU utilization	0.100	18.9	3.35	0.99	0.99	34.5	0.57	0.97	0.97
CPU utilization	0.200	26.0	3.27	0.98	0.97	51.7	0.44	0.92	0.92
Memory usage	0.001	24.0	0.50	0.89	0.88	32.6	0.00	0.42	0.38
Memory usage	0.010	26.5	0.40	0.88	0.87	32.2	0.03	0.40	0.37
Memory usage	0.100	32.1	0.49	0.83	0.81	40.8	0.03	0.30	0.25
Memory usage	0.200	36.3	0.40	0.89	0.88	43.4	0.01	0.44	0.41
Network traffic	0.001	0.1	0.27	1.00	1.00	0.1	0.11	1.00	1.00
Network traffic	0.010	0.1	0.27	1.00	1.00	0.1	0.11	1.00	1.00
Network traffic	0.100	-0.1	0.30	1.00	1.00	0.5	0.11	1.00	1.00
Network traffic	0.200	0.1	0.32	0.98	0.98	2.1	0.10	0.99	0.99

Table A.10.: Linear regression analysis results: scaling the population or throughput for expectations and capability profiles of different sizes.

KPI	p	Horizontal BASE_h				Vertical BASE_v			
		α_h	β_h	R^2	adj. R^2	α_v	β_v	R^2	adj. R^2
CPU utilization	5	15.9	3.12	1.00	0.99	18.3	0.65	0.99	0.99
CPU utilization	10	16.4	3.31	0.99	0.99	23.5	0.61	0.97	0.96
CPU utilization	20	21.6	3.81	0.87	0.83	74.4	0.23	0.92	0.91
CPU utilization	30	70.9	1.37	0.91	0.88	75.8	0.22	0.92	0.91
Memory usage	5	26.5	0.40	0.88	0.87	32.2	0.03	0.39	0.36
Memory usage	10	29.1	0.58	0.84	0.83	41.7	0.00	0.00	-0.06
Memory usage	20	35.4	0.97	0.71	0.63	62.0	-0.06	0.15	0.10
Memory usage	30	56.6	-0.29	0.91	0.88	53.9	-0.04	0.65	0.63
Network traffic	5	0.1	0.27	1.00	1.00	0.1	0.11	1.00	1.00
Network traffic	10	0.1	0.28	1.00	1.00	0.1	0.11	1.00	1.00
Network traffic	20	0.0	0.28	1.00	1.00	0.1	0.11	1.00	1.00
Network traffic	30	0.0	0.28	1.00	1.00	0.1	0.11	1.00	1.00

A.8 Experimental Setup Application-Specific Integrated Aggregation (ASIA)

The effectiveness of ASIA as a monitoring concept for a DEBS has been quantitatively evaluated using a testbed for distributed systems. It consists of 32 physical machines simulating a DEBS with up to 1600 clients connected to a total of 16 brokers (cf., Section 6.4).

16 Intel Core i5 nodes host the brokers of the DEBS. Each broker runs on a single node equipped with 8 cores at 3.1 GHz and 4 GiB of RAM. Another set of 16 nodes with 8 Intel Xeon 1.86 GHz cores and 8 GiB of RAM each host the subscribers and publishers that produce network traffic; all subscribers and publishers connected to a given broker are hosted on the same node. All machines run Linux version 3.4.13. Table A.11 shows further details.

Table A.11.: Parameters for the ASIA deployment used in the reference scenario.

Population	
Parameter	Value
Number of brokers	16
Publishers and subscribers per broker	100
Subscribers in delay tests	8
Different topics	100
Average link latency	0.2ms
Dynamics	
Parameter	Value
Average subscriptions per subscriber	50
Subscription/unsubscription ratio	50% / 50%
Average publication rate (per publisher)	1 notification/s
Average subscription rate (per subscriber)	0.1 subscription/s
Average aggregation requests per publisher/subscriber	3
Maximum imprecision	0

B Bibliography

- [1] A. M. Abbas and O. Kure. Quality of service in mobile ad hoc networks: a survey. *International Journal of Ad Hoc and Ubiquitous Computing*, 6(2):75–98, 2010. [Cited on pages 220, 222, 224, and 226.]
- [2] R. Abbott. Complex systems + systems engineering = complex systems engineering. *arXiv preprint cs/0603127*, 2006. [Cited on pages 22, 68, and 94.]
- [3] R. Abbott. Putting complex systems to work. *Complexity*, 13(2):30–49, 2007. [Cited on pages 22, 68, and 94.]
- [4] TF Abdelzater, E. M. Atkins, and K. G. Shin. QoS negotiation in real-time systems and its application to automated flight control. *IEEE Transactions on Computers*, 49(11):1170–1183, 2000. [Cited on page 203.]
- [5] Z. Abid and S. Chabridon. A fine-grain approach for evaluating the quality of context. In *PERCOM’11 Workshops*, 2011. [Cited on pages 28, 39, 220, 222, 224, and 226.]
- [6] Z. Abid, S. Chabridon, and D. Conan. A framework for quality of context management. In *QuaCon’09*, 2009. [Cited on pages 26, 28, 29, 34, 36, 37, 38, 39, 42, 45, 61, 220, 222, 224, and 226.]
- [7] S. Agarwala, Y. Chen, D. Milojicic, and K. Schwan. QMON: QoS-and utility-aware monitoring in enterprise systems. In *ICAC’06*, 2006. [Cited on pages 15, 37, 147, and 203.]
- [8] E. Aitenbichler and J. Kangasharju. Communication abstractions in MundoCore. In *CADS workshop at ECOOP’03*, 2003. [Cited on page 201.]
- [9] E. Aitenbichler, J. Kangasharju, and M. Mühlhäuser. MundoCore: A light-weight infrastructure for pervasive computing. *Pervasive and Mobile Computing*, 3(4):332–361, 2007. [Cited on page 201.]
- [10] T. Akida, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In *VLDB’15*, 2015. [Cited on page 19.]
- [11] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *Communications magazine*, 40(8):102–114, 2002. [Cited on pages 17 and 18.]
- [12] M. Alaya, S. Matoussi, T. Monteil, and K. Drira. Autonomic computing system for self-management of machine-to-machine networks. In *Self-IoT’12*, 2012. [Cited on pages 1 and 79.]
- [13] D. Ameller and X. Franch. Service level agreement monitor (SALMon). In *ICCBSS’08*, 2008. [Cited on page 20.]
- [14] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW’11*, 2011. [Cited on page 17.]

-
- [15] J. Antollini, M. Antollini, P. Guerrero, and M. Cilia. Extending REBECA to support concept-based addressing. In *ASIS'04*, 2004. [Cited on pages 15 and 16.]
- [16] Apache Software Foundation. Welcome to hadoop! <http://hadoop.apache.org/core/>, 2010. [Cited on page 3.]
- [17] S. Appel. *Integration of Event Processing with Service-oriented Architectures and Business Processes*. PhD thesis, Technische Universität Darmstadt, 2014. [Cited on pages 11, 15, 20, and 200.]
- [18] S. Appel, S. Frischbier, T. Freudenreich, and A. Buchmann. Eventlets: Components for the integration of event streams with SOA. In *SOCA'12*, 2012. [Cited on pages 8, 11, 17, 20, 98, and 114.]
- [19] S. Appel, S. Frischbier, T. Freudenreich, and A. Buchmann. Event stream processing units in business processes. In *BPM'13*, 2013. [Cited on pages 8, 11, and 98.]
- [20] S. Appel, P. Kleber, S. Frischbier, T. Freudenreich, and A. Buchmann. Modeling and execution of event stream processing in business processes. *Information Systems*, 2014. [Cited on pages 11 and 98.]
- [21] S. Appel, K. Sachs, and A. Buchmann. Quality of service in event-based systems. In *22nd GI-Workshop on Foundations of Databases (GvD'11)*, 2010. [Cited on pages 26, 27, 30, 202, 220, 222, 224, and 226.]
- [22] F. Araujo and L. Rodrigues. The IndiQoS message broker: an instantiation using RSVP. Technical report, University of Lisbon, 2002. [Cited on pages 27, 34, 46, 145, 201, 220, 222, 224, and 226.]
- [23] F. Araujo and L. Rodrigues. On QoS-aware publish-subscribe. In *ICDCSW'02*, 2002. [Cited on pages 5, 26, 145, 149, and 201.]
- [24] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010. [Cited on page 20.]
- [25] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, University of California, Berkeley, 2009. [Cited on page 20.]
- [26] M. Armstrong. Competition in two-sided markets. *The RAND Journal of Economics*, 37(3):668–691, 2006. [Cited on page 218.]
- [27] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010. [Cited on page 18.]
- [28] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems*, 6(3):138–151, May 1998. [Cited on page 26.]
- [29] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS'02*, 2002. [Cited on page 19.]
- [30] S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001. [Cited on pages 17 and 19.]

- [31] A. Bahjat, Y. Jiang, T. Cook, and T. La Porta. Quality of information functions for networked applications. In *PERCOM'12 Workshops*, 2012. [Cited on pages 28, 37, 38, 203, 220, 222, 224, and 226.]
- [32] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. Technical report, Universita di Roma La Sapienza, 2005. [Cited on pages 16, 220, 222, 224, and 226.]
- [33] D. Banks, J. Erickson, and M. Rhodes. Multi-tenancy in cloud-based collaboration services. *Information Systems*, 2009. [Cited on page 22.]
- [34] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL processes. In *ICSOC'05*, 2005. [Cited on page 20.]
- [35] P. Barnaghi, M. Bermudez-Edo, and R. Tönjes. Challenges for Quality of Data in Smart Cities. *Journal of Data and Information Quality*, 6(2):1–4, June 2015. [Cited on pages 2, 17, 29, 35, 39, 46, 60, 61, 94, 220, 222, 224, and 226.]
- [36] Y. Bartal, J. Byers, and D. Raz. Global optimization using local information with applications to flow control. In *FOCS'97*, 1997. [Cited on page 88.]
- [37] C. Batini, C. Cappiello, C. Francalanci, and A. Maurino. Methodologies for data quality assessment and improvement. *ACM Computing Surveys*, 41(3):1–52, July 2009. [Cited on pages 28, 31, 34, 61, 220, 222, 224, and 226.]
- [38] D. Battre, F. Brazier, K. Clark, M. Oey, A. Papaspouros, O. Wälrich, P. Wieder, and W. Ziegler. A proposal for WS-Agreement negotiation. In *GRID'10*, 2010. [Cited on page 20.]
- [39] H. Bauer, M. Patel, and J. Veira. The Internet of Things: Sizing up the opportunity. Technical report, McKinsey & Company, 2015. [Cited on page 1.]
- [40] M. Bechler, H. Ritter, G. Schäfer, and J. Schiller. Traffic shaping in end systems attached to QoS-supporting networks. In *ISCC'01*, 2001. [Cited on page 93.]
- [41] S. Behnel, A. Buchmann, P. Grace, B. Porter, and G. Coulson. A specification-to-deployment architecture for overlay networks. In *DOA'06*, 2006. [Cited on pages 28 and 35.]
- [42] S. Behnel, L. Fiege, and G. Mühl. On Quality-of-Service and publish/subscribe. In *DEBS'06*, 2006. [Cited on pages 2, 25, 28, 36, 63, 202, 220, 222, 224, and 226.]
- [43] M. Beigl, A. Krohn, T. Zimmer, and C. Decker. Typical sensors needed in ubiquitous and pervasive computing. In *INSS'04*, 2004. [Cited on page 18.]
- [44] C. Belady. In the data center, power and cooling costs more than the it equipment it supports. <http://goo.gl/9zvNG>, 2007. [Cited on page 3.]
- [45] M. Belghachi and M. Feham. QoS routing scheme and route repair in WSN. *International Journal of Advanced Computer Science and Applications*, 3(12):81–86, 2012. [Cited on page 18.]
- [46] P. Bellavista, A. Corradi, M. Fanelli, and L. Foschini. A survey of context data distribution for mobile ubiquitous systems. *ACM Computing Surveys*, 44(4):1–45, August 2012. [Cited on pages 34, 39, 220, 222, 224, and 226.]

- [47] P. Bellavista, A. Corradi, and A. Reale. Quality of Service in wide scale publish/subscribe systems. *IEEE Communications Surveys & Tutorials*, pages 1–26, 2014. [Cited on pages 28, 36, 37, 38, 39, 40, 220, 222, 224, and 226.]
- [48] A. Beloglazov and R. Buyya. Energy efficient resource management in virtualized cloud data centers. In *ICCCGC'10*, 2010. [Cited on page 3.]
- [49] A. Berl, E. Gelenbe, M. Di Girolamo, G. Giuliani, H. De Meer, M. Dang, and K. Pentikousis. Energy-efficient cloud computing. *The Computer Journal*, 53(7):1045–1051, 2010. [Cited on page 3.]
- [50] E. Bertino. Data trustworthiness – approaches and research challenges. In *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance*, pages 17–25. Springer, 2015. [Cited on page 2.]
- [51] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Rionbi. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161 – 180, 2010. [Cited on pages 38, 220, 222, 224, and 226.]
- [52] F. Bian, D. Kempe, and R. Govindan. Utility based sensor selection. In *IPSN'06*, 2006. [Cited on page 18.]
- [53] IEC BiPM, ILAC IFCC, IUPAC ISO, and OIML IUPAP. International vocabulary of metrology—basic and general concepts and associated terms, 2008. *JCGM*, 200:99–12, 2008. [Cited on pages 38 and 39.]
- [54] C. Bisdikian. On sensor sampling and quality of information: A starting point. In *PERCOM'07 Workshops*, pages 279–284, 2007. [Cited on pages 18, 27, 34, 38, 39, 61, 220, 222, 224, and 226.]
- [55] C. Bisdikian, J. Branch, K. Leung, and R. Young. A letter soup for the quality of information in sensor networks. In *PERCOM'09*, 2009. [Cited on pages 2, 26, 28, 34, 220, 222, 224, and 226.]
- [56] C. Bisdikian, L. Kaplan, and M. Srivastava. On the quality and value of information in sensor networks. *ACM Transactions on Sensor Networks*, 9(4):39:26, 2010. [Cited on pages 2, 18, 27, 46, 220, 222, 224, and 226.]
- [57] C. Bisdikian, L. Kaplan, M. Srivastava, D. Thornley, D. Verma, and R. Young. Building principles for a quality of information specification for sensor information. In *FUSION'09*, 2009. [Cited on pages 28, 34, 38, 39, 63, 220, 222, 224, and 226.]
- [58] T. Bishop and R. Karne. A survey of middleware. In *CATA'03*, 2003. [Cited on page 16.]
- [59] G. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci. The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems. In *Middleware'11*, 2011. [Cited on page 23.]
- [60] G. Blair, G. Coulson, M. Clarke, and N. Parlantzas. Performance and Integrity in the OpenORB Reflective Middleware. In *Metalevel Architectures and Separation of Crosscutting Concerns*, pages 268–269. Springer, 2001. [Cited on pages 22 and 68.]
- [61] G. Blair, G. Coulson, and P. Grace. Research directions in reflective middleware: the Lancaster experience. In *ARM'04*, 2004. [Cited on pages 22 and 68.]

-
- [62] U. Blanke, R. Rehner, and B. Schiele. South by South-east or Sitting at the Desk: Can Orientation be a Place? In *ISWC'11*, 2011. [Cited on pages 5, 34, 43, 60, 61, 62, 94, 220, 222, 224, and 226.]
- [63] J. Boardman and B. Sauser. System of systems-the meaning of of. In *SYSOSE'06*, 2006. [Cited on pages 22, 23, 68, and 218.]
- [64] C. Bolchini, C. Curino, E. Quintarelli, F. Schreiber, and L. Tanca. A data-oriented survey of context models. *ACM Sigmod Record*, 36(4):19–26, 2007. [Cited on page 28.]
- [65] P. Boonma and J. Suzuki. TinyDDS: An interoperable and configurable. *Principles and Applications of Distributed Event-Based Systems*, 2010. [Cited on pages 18, 200, and 201.]
- [66] J. Borges Neto, T. Silva, R. Assunção, R. Mini, and A. Loureiro. Sensing in the collaborative internet of things. *Sensors*, 15(3):6607–6632, 2015. [Cited on pages 18 and 35.]
- [67] S. Bosse, M. Splieth, and K. Turowski. Multi-objective optimization of IT service availability and costs. *Reliability Engineering & System Safety*, 2015. [Cited on page 20.]
- [68] A. Bouius. Characterising the ripple effects of introducing energy-awareness functionality in cyber-physical system software. Master’s thesis, University of Twente, 2015. [Cited on pages 22 and 68.]
- [69] D. Box, L. Cabrera, C. Critchley, F. Curbera, D. Ferguson, A. Geller, S. Graham, D. Hull, G. Kakivaya, A. Lewis, et al. Web services eventing (WS-Eventing). *W3C member submission*, 15, 2006. [Cited on page 20.]
- [70] J. Boyd. A discourse on winning and losing. maxwell air force base, al: Air university. *Library Document No. MU*, 43947, 1987. [Cited on page 79.]
- [71] D. Braha, N. Suh, S. Eppinger, M. Caramanis, and D. Frey. *Complex engineered systems*. Springer, 2006. [Cited on pages 22, 23, and 218.]
- [72] P. Brebner. Performance modeling for service oriented architectures. In *ICSE'08*, 2008. [Cited on page 90.]
- [73] A. Brook. Low-latency distributed applications in finance. *Communications of the ACM*, 58(7):42–50, June 2015. [Cited on pages 3 and 19.]
- [74] P. Brooks and B. Hestnes. User measures of quality of experience: why being objective and quantitative is important. *IEEE Network*, 24(2):8–13, 2010. [Cited on pages 26 and 28.]
- [75] T. Buchholz, A. Küpper, and M. Schiffers. Quality of context: What it is and why we need it. In *HP-OVUA'03 Workshop*, 2003. [Cited on pages 26, 27, 28, 34, 38, 39, 220, 222, 224, and 226.]
- [76] A. Buchmann, S. Appel, T. Freudenreich, S. Frischbier, and P. Guerrero. From calls to events: Architecting future BPM systems. In *BPM'12*, 2012. [Cited on pages 1, 4, 8, and 11.]
- [77] A. Buchmann, C. Bornhövd, M. Cilia, L. Fiege, F. Gärtner, C. Liebig, M. Meixner, and G. Mühl. DREAM: Distributed reliable event-based application management. In *Web Dynamics: Adapting to Change in Content, Size, Topology and Use*, pages 319–350. Springer, May 2004. [Cited on pages 14, 16, 201, 203, and 204.]

- [78] A. Buchmann, H. Pfohl, S. Appel, T. Freudenreich, S. Frischbier, I. Petrov, and C. Zuber. Event-Driven services: Integrating production, logistics and transportation. In *SOCLOG'10*, 2010. [Cited on pages 1, 4, 8, 11, 17, 20, and 72.]
- [79] J. Byers and G. Nasser. Utility-based decision-making in wireless sensor networks. In *MobiHOC'00*, 2000. [Cited on pages 18, 60, 61, 62, 87, 88, 94, and 203.]
- [80] J. Byers and G. Nasser. Utility-based decision-making in wireless sensor networks. Technical report, Boston University, 2000. [Cited on pages 18, 87, 88, and 203.]
- [81] C. Cai, L. Wang, S. Khan, and J. Tao. Energy-aware high performance computing: A taxonomy study. In *ICPADS'11*, 2011. [Cited on pages 1, 3, and 4.]
- [82] A. Campbell, C. Aurrecoechea, and L. Hauw. A review of QoS architectures. In *IWQS'96*, 1996. [Cited on page 26.]
- [83] M. Caramia and P. Dell'Olmo. *Multi-objective management in freight logistics: Increasing capacity, service level and safety with optimization algorithms*. Springer Science & Business Media, 2008. [Cited on pages 87 and 88.]
- [84] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes. Long-term SLOs for reclaimed cloud computing resources. In *SOCC'14*, 2014. [Cited on page 210.]
- [85] N. Carvalho, F. Araujo, and L. Rodrigues. Scalable QoS-based event routing in publish-subscribe systems. In *NCA'05*, 2005. [Cited on pages 2, 5, 25, 29, 37, 145, 147, 148, 201, 220, 222, 224, and 226.]
- [86] A. Carzaniga. *Architectures for an event notification service scalable to wide-area networks*. PhD thesis, Politecnico di Milano, 1998. [Cited on pages 201 and 203.]
- [87] A. Carzaniga, E. Di Nitto, D. Rosenblum, and A. Wolf. Issues in supporting event-based architectural styles. In *ISAW'98*, 1998. [Cited on page 15.]
- [88] A. Carzaniga, D. S Rosenblum, and A. L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, 2001. [Cited on page 16.]
- [89] M. Castellanos, U. Dayal, and M. Hsu. *Live Business Intelligence for the Real-Time Enterprise*, volume 6462 of *LNCS*, pages 325–336. Springer, 2010. [Cited on page 17.]
- [90] P. Chahuara, F. Portet, and M. Vacher. Making context aware decision from uncertain information in a smart home: A markov logic network approach. In *Ambient Intelligence*, pages 78–93. Springer, 2013. [Cited on page 17.]
- [91] S. Chakravarthy and R. Adaikkalavan. Events and streams: harnessing and unleashing their synergy! In *DEBS'08*, 2008. [Cited on page 19.]
- [92] J. Chambers. *Graphical methods for data analysis*. Springer, 1983. [Cited on page 51.]
- [93] M. Chandy. Sense and respond systems. In *CMG'05*, 2005. [Cited on page 1.]
- [94] M. Chandy, M. Charpentier, and A. Capponi. Towards a theory of events. In *DEBS'07*, 2007. [Cited on page 13.]
- [95] M. Chandy and W. Schulte. *Event Processing: Designing IT Systems for Agile Companies*. McGraw-Hill, Inc., 2010. [Cited on pages 28, 36, 38, 46, 220, 222, 224, and 226.]

-
- [96] Z. Charbiwala, S. Zahedi, Y. Kim, Y. Cho, and M. Srivastava. Toward quality of information aware rate control for sensor networks. In *FeBID'09*, 2009. [Cited on pages 18, 38, 60, 94, 220, 222, 224, and 226.]
- [97] A. Chazalet. Service level agreements compliance checking in the cloud computing: Architectural pattern, prototype, and validation. In *ICSEA'10*, 2010. [Cited on page 20.]
- [98] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD'00*, 2000. [Cited on page 19.]
- [99] M. Chen and M. Fowler. Data compression trade-offs in sensor networks. In *SPIE'04*, 2004. [Cited on pages 5, 18, 38, 40, 60, 61, 94, 220, 222, 224, and 226.]
- [100] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *CIDR'03*, 2003. [Cited on page 19.]
- [101] A. Cheung and H.A. Jacobsen. Publisher placement algorithms in content-based publish-/subscribe. In *ICDCS'10*, 2010. [Cited on page 210.]
- [102] S. Chong, C. Skalka, and J. Vaughan. Self-identifying sensor data. In *IPSN'10*, 2010. [Cited on page 18.]
- [103] K. Church, A. Greenberg, and J. Hamilton. On Delivering Embarrassingly Distributed Cloud Services. *Hotnets'08*, 2008. [Cited on page 22.]
- [104] M. Cilia, M. Antollini, C. Bornhövd, and A. Buchmann. Dealing with Heterogeneous Data in Pub/Sub Systems: The Concept-Based Approach. In *DEBS'04*, 2004. [Cited on pages 15 and 16.]
- [105] M. Cilia, C. Bornhövd, and A. Buchmann. CREAM: An infrastructure for distributed, heterogeneous event-based applications. In *OTM Confederated International Conferences, CoopIS, DOA, and ODBASE*. Springer, 2003. [Cited on pages 16, 201, 203, and 204.]
- [106] T. Cioara, I. Salomie, I. Anghel, I. Chira, A. Cocian, E. Henis, and R. Kat. A dynamic power management controller for optimizing servers' energy consumption in service centers. In *ICSOC'11*, 2011. [Cited on pages 1, 3, and 4.]
- [107] R. Cocchi, S. Shenker, D. Estrin, and L. Zhang. Pricing in computer networks: Motivation, formulation, and example. *IEEE/ACM Transactions on Networking*, 1(6):614–627, 1993. [Cited on page 78.]
- [108] Federal Telecommunications Standards Committee. Federal Standard 1037C: Glossary of Telecommunications Terms (FED-STD-1037C). *National Communications System Technology Program Office*, 1996. [Cited on page 34.]
- [109] M. Compton, P. Barnaghi, L. Bermudez, R. Garcia-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. Kelsey, D. Le Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, and K. Taylor. The SSN Ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17(0), 2012. [Cited on pages 18, 34, 35, 38, 39, 45, 60, 61, 94, 220, 222, 224, and 226.]
- [110] M. Compton, C. Henson, H. Neuhaus, L. Lefort, and A. Sheth. A survey of the semantic specification of sensors. In *ISSN'09 Workshop*, 2009. [Cited on pages 18, 38, 61, 220, 222, 224, and 226.]

- [111] D. Cook and S. Das. *Smart Environments: Technology, Protocols and Applications*. Wiley & Sons, 2005. [Cited on page 17.]
- [112] A. Corsaro, L. Querzoni, S. Scipioni, S. Piergiovanni, and A. Virgillito. Quality of service in publish/subscribe middleware. *Global Data Management*, 8:1–19, 2006. [Cited on pages 37, 220, 222, 224, and 226.]
- [113] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. Pearson Education, 2005. [Cited on page 37.]
- [114] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A Generic Component Model for Building Systems Software. *ACM TOCS*, 26(1):1–42, March 2008. [Cited on pages 22 and 68.]
- [115] G. Cugola and E. Di Nitto. On adopting content-based routing in service-oriented architectures. *Information and Software Technology*, 50(1-2):22–35, 2008. [Cited on page 20.]
- [116] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *ICSE'98*, 1998. [Cited on page 201.]
- [117] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *Software Engineering*, 27(9):827–850, 2001. [Cited on page 201.]
- [118] G. Cugola, D. Frey, A. Murphy, and G. Picco. Content-based routing for publish-subscribe on a dynamic topology: Concepts, protocols, and evaluation. Technical report, Politecnico di Milano, 2005. [Cited on page 16.]
- [119] G. Cugola and A. Margara. RACED: an adaptive middleware for complex event detection. In *ARM'09*, 2009. [Cited on page 17.]
- [120] G. Cugola and A. Margara. TESLA: a formally defined event specification language. In *DEBS'10*, 2010. [Cited on page 17.]
- [121] G. Cugola and A. Margara. Complex event processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012. [Cited on page 17.]
- [122] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012. [Cited on pages 17, 19, 28, 35, 220, 222, 224, and 226.]
- [123] G. Cugola and A. Margara. Deployment strategies for distributed complex event processing. *Computing*, 95(2):129–156, 2013. [Cited on page 17.]
- [124] G. Cugola and A. Margara. The complex event processing paradigm. *Data Management in Pervasive Systems*, 2015. [Cited on pages 17 and 19.]
- [125] G. Cugola, A. Margara, M. Matteucci, and G. Tamburrelli. Introducing uncertainty in complex event processing: Model, implementation, and validation. *Computing*, 2015. [Cited on pages 2, 17, 28, 220, 222, 224, and 226.]
- [126] G. Cugola and G. Picco. REDS: a reconfigurable dispatching system. In *SEM'06*, 2006. [Cited on pages 16 and 131.]

-
- [127] C. Dai, D. Lin, E. Bertino, and M. Kantarcioglu. An approach to evaluate data trustworthiness based on data provenance. In *Secure Data Management*, pages 82–98. Springer, 2008. [Cited on pages 23 and 39.]
- [128] R. Das, J. Kephart, J. Lenchner, and H. Hamann. Utility-function-driven energy-efficient cooling in data centers. In *ICAC’10*, 2010. [Cited on page 3.]
- [129] O. de Carvalho, E. Roloff, and P. Navaux. A Survey of the state-of-the-art in event processing. In *WSPPD’13*, 2013. [Cited on page 16.]
- [130] T. De Wolf and T. Holvoet. *Emergence versus self-organisation: Different concepts but promising when combined*, volume 3464 of *LNCS*, pages 77–91. Springer, 2005. [Cited on page 23.]
- [131] N. Deakin. Java Message Service (JMS) API. <http://www.jcp.org/en/jsr/detail?id=914>, 2002. [Cited on pages 16 and 200.]
- [132] J. Dean. Modeling and checking service level agreements for service-oriented architectures. Master’s thesis, Technische Universität Darmstadt, 2012. In cooperation with Software AG, Darmstadt, Germany. [Cited on page 11.]
- [133] K. Deb. *Multi-objective optimization using evolutionary algorithms*. Wiley & Sons, 2001. [Cited on page 88.]
- [134] C. Demichelis and P. Chimento. IP packet delay variation metric for ip performance metrics (IPPM). Technical report, The Internet Society, 2002. [Cited on page 36.]
- [135] P. Dube, N. Halim, K. Karenos, M. Kim, Z. Liu, S. Parthasarathy, D. Pendarakis, and H. Yang. Harmony: Holistic messaging middleware for event-driven systems. *IBM Systems Journal*, 47(2):281–287, 2008. [Cited on pages 146 and 201.]
- [136] R. Dumke, S. Mencke, and C. Wille. *Quality Assurance of Agent-Based and Self-Managed Systems*. CRC Press, 2009. [Cited on pages 22 and 23.]
- [137] M. Eckert, F. Bry, S. Brodt, O. Poppe, and S. Hausmann. A CEP babelfish: Languages for complex event processing and querying surveyed. In *Reasoning in Event-Based Distributed Systems*, pages 47–70. Springer, 2011. [Cited on page 17.]
- [138] M. Eichholz. Supporting latency requirements in activemq broker networks by self-adaptation. Master’s thesis, Technische Universität Darmstadt, 2015. [Cited on pages 11, 43, 90, 94, 126, 146, 200, and 201.]
- [139] G. Eisenhauer, F. Bustamante, and K. Schwan. Event services for high performance computing. In *HPDC’00*, 2000. [Cited on page 16.]
- [140] B. Elahi, K. Romer, B. Ostermaier, M. Fahrnair, and W. Kellerer. Sensor ranking: A primitive for efficient content-based sensor search. In *IPSN’09*, 2009. [Cited on page 18.]
- [141] A. Elwalid and D. Mitra. Traffic shaping at a network node: theory, optimum design, admission control. In *INFOCOM’97*, 1997. [Cited on page 93.]
- [142] Y. Engel and O. Etzion. Towards proactive event-driven computing. In *DEBS’11*, 2011. [Cited on page 210.]
- [143] M. Eppler and M. Helfert. A classification and analysis of data quality costs. In *ICIQ’04*, 2004. [Cited on pages 34, 40, 46, 47, 50, 61, and 62.]

- [144] D. Estrin and L. Zhang. Design considerations for usage accounting and feedback in internetworks. *SIGCOMM Computer Communication Review*, 20(5):56–66, 1990. [Cited on pages 68 and 78.]
- [145] O. Etzion and P. Niblett. *Event processing in action*. Manning Publications Co., 2010. [Cited on page 17.]
- [146] P. Eugster. *Type-based publish/subscribe*. PhD thesis, EPFL Lausanne, 2001. [Cited on page 15.]
- [147] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003. [Cited on pages 1, 15, and 101.]
- [148] P. Eugster, R. Guerraoui, and C. Damm. On objects and events. In *ACM SIGPLAN Notices*, volume 36, pages 254–269. ACM, 2001. [Cited on page 15.]
- [149] D. Evans, A. Hagiwara, and R. Schmalensee. *Invisible Engines: How Software Platforms Drive Innovation and Transform Industries*. The MIT Press, 2006. [Cited on page 218.]
- [150] D. Eyers, T. Freudenreich, A. Margara, S. Frischbier, P. Pietzuch, and P. Eugster. Living in the present: on-the-fly information processing in scalable web architectures. In *CloudCP’12*, 2012. [Cited on page 11.]
- [151] M. Farina and P. Amato. On the optimal solution definition for many-criteria optimization problems. In *NAFIPS’02*, 2002. [Cited on page 88.]
- [152] E. Fasolo, M. Rossi, J. Widmer, and M. Zorzi. In-network aggregation techniques for wireless sensor networks: A survey. *IEEE Wireless Commun.*, 14(2):70–87, 2007. [Cited on pages 18, 93, and 204.]
- [153] T. Ferrari. End-to-end performance analysis with traffic aggregation. *Computer Networks*, 34(6):905–914, 2000. [Cited on pages 93, 220, 222, 224, and 226.]
- [154] G. Feuerlicht. Enterprise SOA: what are the benefits and challenges? *Systems Integration*, 2006. [Cited on page 19.]
- [155] G. Feuerlicht. Next generation SOA: can SOA survive cloud computing? *AWIC’09*, 2010. [Cited on page 19.]
- [156] G. Feuerlicht and S. Govardhan. SOA: trends and directions. *Systems Integration*, 2009. [Cited on page 19.]
- [157] E. Fidler, H.A. Jacobsen, G. Li, and S. Mankovski. The PADRES distributed publish/subscribe system. *Feature Interactions in Telecommunications and Software Systems*, 2005. [Cited on pages 16, 201, and 203.]
- [158] M. Fidler, V. Sander, and W. Klimala. Traffic shaping in aggregate-based networks: implementation and analysis. *Computer Communications*, 28(3):274–286, 2005. [Cited on page 93.]
- [159] M. Fiedler, T. Hossfeld, and P. Tran-Gia. A generic quantitative relationship between quality of experience and quality of service. *Network*, 24(2):36–41, 2010. [Cited on pages 26, 28, 29, 35, 220, 222, 224, and 226.]
- [160] L. Fiege, M. Mezini, G. Mühl, and A. Buchmann. Engineering event-based systems with scopes. *ECOOP’02*, 2006. [Cited on page 211.]

-
- [161] K. Fleszar, C. Glaßer, F. Lipp, C. Reitwießner, and M. Witek. The complexity of solving multiobjective optimization problems and its relation to multivalued functions. In *ECC'11*, 2011. [Cited on page 87.]
- [162] Will Forrest and C. Barthold. Clearing the air on cloud computing. Discussion document, McKinsey & Company, March 2008. [Cited on page 20.]
- [163] I. Foster. What is the Grid? A Three Point Checklist. *GRIDtoday*, 2002. [Cited on page 22.]
- [164] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *GCE'08*, 2008. [Cited on pages 20, 21, and 22.]
- [165] T. Freudenreich. Automatic context transformation in complex event processing systems. Master's thesis, Technische Universität Darmstadt, 2011. [Cited on pages 11 and 44.]
- [166] T. Freudenreich, S. Appel, S. Frischbier, and A. Buchmann. ACTrESS - automatic context transformation in event-based software systems. In *DEBS'12*, 2012. [Cited on pages 11, 44, and 45.]
- [167] T. Freudenreich, S. Appel, S. Frischbier, and A. Buchmann. Using policies for handling complexity of event-driven architectures. In *ECSA'14*, 2014. [Cited on page 11.]
- [168] T. Freudenreich, P. Eugster, S. Frischbier, S. Appel, and A. Buchmann. Implementing federated object systems. In *ECOOP'13*, 2013. [Cited on pages 11 and 44.]
- [169] S. Frischbier, A. Buchmann, and D. Pütz. FIT for SOA? Introducing the F.I.T. – metric to optimize the availability of service oriented architectures. In *CSDM*, 2011. [Cited on pages 11, 19, 20, 22, 30, 72, 88, 91, and 217.]
- [170] S. Frischbier, M. Gesmann, D. Mayer, A. Roth, and C. Webel. Emergence as competitive advantage - engineering tomorrow's enterprise software systems. In *ICEIS'12*, 2012. [Cited on pages 11, 20, 23, 25, 72, 73, 190, and 218.]
- [171] S. Frischbier, A. Margara, T. Freudenreich, P. Eugster, D. Eyers, and P. Pietzuch. ASIA: application-specific integrated aggregation for Publish/Subscribe middleware. In *Middleware 2012 Posters and Demos Track*, 2012. [Cited on pages 5, 11, 68, 73, 101, and 148.]
- [172] S. Frischbier, A. Margara, T. Freudenreich, P. Eugster, D. Eyers, and P. Pietzuch. Aggregation for implicit invocations. In *AOSD'13*, 2013. [Cited on pages 11, 15, 16, 34, 73, 101, 113, 137, 148, and 188.]
- [173] S. Frischbier, A. Margara, T. Freudenreich, P. Eugster, D. Eyers, and P. Pietzuch. McCAT: Multi-cloud Cost-aware Transport. In *EuroSys Poster Track*, 2014. [Cited on pages 5, 11, 22, 34, 43, 51, 61, 77, 90, 94, 101, 120, 135, and 218.]
- [174] S. Frischbier and I. Petrov. *Aspects of Data-Intensive Cloud Computing*, volume 6462 of *LNCS*, pages 57–77. Springer, 2010. [Cited on pages xv, 11, 20, 21, and 22.]
- [175] S. Frischbier, P. Pietzuch, and A. Buchmann. Managing expectations: Runtime negotiation of information quality requirements in event-based systems. *ICSOC'14*, 2014. [Cited on pages 11, 25, and 51.]
- [176] S. Frischbier, K. Sachs, and A. Buchmann. Evaluating RFID Infrastructures. In *ITG-Fachbericht - 2. Workshop RFID*, 2006. [Cited on pages 4, 11, and 190.]

- [177] S. Frischbier, E. Turan, M. Gesmann, A. Margara, D. Eyers, P. Eugster, P. Pietzuch, and A. Buchmann. Effective runtime monitoring of distributed event-based enterprise systems with ASIA. *SOCA'14*, 2014. [Cited on pages 4, 11, 25, 72, 77, 101, 148, and 188.]
- [178] J. Fromm. *The Emergence of Complexity*. Kassel University Press, 2004. [Cited on page 23.]
- [179] J. Fromm. Ten questions about emergence. *arXiv:nlin/0509049 [nlin.AO]*, 2005. [Cited on page 23.]
- [180] A. Frömmgen, R. Rehner, M. Lehn, and A. Buchmann. Fossa: Learning ECA rules for adaptive distributed systems. In *ICAC'15*, 2015. [Cited on page 210.]
- [181] M. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: you only get one look a tutorial. In *SIGMOD'02*, 2002. [Cited on page 19.]
- [182] M. Gharib and P. Giorgini. Modeling and reasoning about information quality requirements. In *REFSQ'15*, 2015. [Cited on pages 2, 28, 35, 38, 39, 46, 220, 222, 224, and 226.]
- [183] D. Gillies, D. Thornley, and C. Bisdikian. Probabilistic approaches to estimating the quality of information in military sensor networks. *The Computer Journal*, 53(5):493–502, 2010. [Cited on pages 220, 222, 224, and 226.]
- [184] R. Giordanelli and C. Mastroianni. The cloud computing paradigm: Characteristics, opportunities and research issues. Technical report, ICAR-CNR, 2010. [Cited on pages 20, 21, and 22.]
- [185] L. Golab and M. Özsü. Data stream management issues—a survey. Technical report, University of Waterloo, 2003. [Cited on page 19.]
- [186] L. Golab and M. Özsü. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003. [Cited on pages 17 and 19.]
- [187] S. Graham, P. Niblett, D. Chappell, A. Lewis, N. Nagaratnam, J. Parikh, S. Patil, S. Samdarshi, I. Sedukhin, D. Snelling, et al. Publish-subscribe notification for web services. *IBM DeveloperWorks Whitepaper*, 2004. [Cited on page 20.]
- [188] T. Grandison and M. Sloman. A survey of trust in internet applications. *IEEE Communications Surveys & Tutorials*, 3(4):2–16, 2000. [Cited on page 39.]
- [189] A. Greenberg, J. Hamilton, D. Maltz, and P. Patel. The cost of a cloud : Research problems in data center networks. *Computer Communication Review*, 39(1):68–73, 2009. [Cited on pages 20 and 22.]
- [190] R. Grossman. The case for cloud computing. *IT Professional*, 11(2):23–27, March/April 2009. [Cited on page 21.]
- [191] P. Groth, S. Miles, W. Fang, S. Wong, K. Zauner, and L. Moreau. Recording and using prove-nance in a protein compressibility experiment. In *HPDC'05*, 2005. [Cited on page 39.]
- [192] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013. [Cited on page 1.]
- [193] P. Guerrero. *Workflow Support for Low-Power Wireless Sensor and Actuator Networks*. PhD thesis, Technische Universität Darmstadt, 2014. [Cited on page 18.]

-
- [194] P. Guerrero, K. Sachs, M. Cilia, C. Bornhövd, and A. Buchmann. Pushing business data processing towards the periphery. In *ICDE'07*, 2007. [Cited on page 18.]
- [195] S. Guinea, G. Kecskemeti, A. Marconi, and B. Wetzstein. Multi-layered monitoring and adaptation. In *ICSOC*, 2011. [Cited on pages 203 and 204.]
- [196] C. Hagen and G. Alonso. Beyond the black box: Event-based inter-process communication in process support systems. In *ICDCS'99*, 1999. [Cited on page 210.]
- [197] A. Hakiri, P. Berthou, A. Gokhale, D. Schmidt, and T. Gayraud. Supporting end-to-end quality of service properties in OMG data distribution service publish/subscribe middleware over wide area networks. *Journal of Systems and Software*, 86(10):2574–2593, October 2013. [Cited on pages 5, 22, 29, 200, 220, 222, 224, and 226.]
- [198] A. Hani, I. Paputungan, and M. Hassan. Renegotiation in service level agreement management for a cloud-based system. *ACM Comput. Surv.*, 47(3):51:1–51:21, April 2015. [Cited on page 203.]
- [199] M. Hashimoto. *Vagrant: Up and Running*. O'Reilly, 2013. [Cited on pages 239 and 244.]
- [200] J. Hauer, V. Handziski, A. Köpke, A. Willig, and A. Wolisz. A component framework for content-based publish/subscribe in sensor networks. In *Wireless Sensor Networks*, pages 369–385. Springer, 2008. [Cited on pages 18 and 201.]
- [201] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *HICSS'00*, 2000. [Cited on page 18.]
- [202] J. Her, S. Choi, S. Oh, and S. Kim. A framework for measuring performance in Service-Oriented architecture. In *NWeSP'07*, 2007. [Cited on page 90.]
- [203] M. Hermann, T. Pentek, and B. Otto. Design principles for industrie 4.0 scenarios: A literature review. Technical report, Technische Universität Dortmund, 2015. [Cited on page 4.]
- [204] E.N. Harness, R.J. High, Jr., and J.R. McGee. Websphere application server: A foundation for on demand computing. *IBM Systems Journal*, 43(2):213–237, 2004. [Cited on page 16.]
- [205] M. Herr, U. Bath, and A. Koschel. Implementation of a service oriented architecture at deutsche post MAIL. *Web Services*, pages 227–238, 2004. [Cited on page 20.]
- [206] A. Herzog and A. Buchmann. A3ME – Generic Middleware for Information Exchange in Heterogeneous Environments. In *INSS'12*, 2012. [Cited on page 18.]
- [207] A. Hinze and A. Buchmann. *Principles and applications of distributed event-based systems*. IGI Global, 2010. [Cited on page 27.]
- [208] A. Hinze, K. Sachs, and A. Buchmann. Event-Based Applications and Enabling Technologies. In *DEBS'09*, 2009. [Cited on pages 1, 4, 13, 14, 15, 28, 34, 36, 101, 220, 222, 224, and 226.]
- [209] M. Hirzel, R. Soule, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys*, 46(4):46, 2014. [Cited on pages 220, 222, 224, and 226.]

- [210] J. Hoffert. *Design and Run-Time Quality of Service Management Techniques for Publish-/Subscribe Distributed Real-time and Embedded Systems*. PhD thesis, Vanderbilt University, 2011. [Cited on pages 27, 28, 36, 146, 201, 220, 222, 224, and 226.]
- [211] J. Hoffert, D. Mack, and D. Schmidt. Using machine learning to maintain pub/sub system qos in dynamic environments. In *ARM'09*, 2009. [Cited on pages 146, 201, and 210.]
- [212] J. Hoffert, D. Mack, and D. Schmidt. Integrating machine learning techniques to adapt protocols for QoS-enabled distributed real-time and embedded Publish/Subscribe middleware. *Network Protocols & Algorithms*, 2(3), 2010. [Cited on pages 146, 201, and 210.]
- [213] J. Hoffert and D. Schmidt. Maintaining QoS for publish/subscribe middleware in dynamic environments. In *DEBS'09*, 2009. [Cited on pages 2, 5, 25, 29, 146, and 201.]
- [214] J. Hoffert, D. Schmidt, and A. Gokhale. A QoS policy configuration modeling language for publish/subscribe middleware platforms. In *DEBS'07*, 2007. [Cited on page 146.]
- [215] J. Hoffert, D. Schmidt, and A. Gokhale. DQML: A modeling language for configuring distributed publish/subscribe quality of service policies. In *OTM'08*, 2008. [Cited on page 146.]
- [216] J. Hoffert, D. Schmidt, and A. Gokhale. Productivity analysis of the distributed QoS modeling language. In *MDASD'10*, 2010. [Cited on page 146.]
- [217] J. Holland. *Emergence: From Chaos to Order*. Oxford University Press, 2000. [Cited on page 23.]
- [218] T. Holmes, E. Mulo, U. Zdun, and S. Dustdar. Model-aware monitoring of SOAs for compliance. In *Service Engineering*, 2010. [Cited on page 20.]
- [219] T. Holmes, U. Zdun, and S. Dustdar. Morse: A model-aware service environment. In *APSCC'09*, 2009. [Cited on pages 203 and 204.]
- [220] A. Holzer, L. Ziarek, K.R. Jayaram, and P. Eugster. Putting events in context: Aspects for event-based distributed programming. In *AOSD'11*, 2011. [Cited on pages 22 and 68.]
- [221] A. Holzer, L. Ziarek, K.R. Jayaram, and P. Eugster. Abstracting context in event-based software. In *Trans. on Aspect-Oriented Software Development*, volume 7271 of *LNCS*, pages 123–167, 2012. [Cited on pages 22 and 68.]
- [222] R. Holzer, H. de Meer, and C. Bettstetter. On autonomy and emergence in self-organizing systems. In *Self-Organizing Systems*, volume 5343 of *LNCS*, pages 157–169. Springer, 2008. [Cited on pages 22, 23, and 68.]
- [223] M. Hossain, D. Ahmed, and J. Parra. A framework for computing quality of information in multi-sensor systems. In *I2MTC'12*, 2012. [Cited on pages 220, 222, 224, and 226.]
- [224] M. Hossain, P. Atrey, and A. El Saddik. Modeling quality of information in multi-sensor surveillance systems. In *ICDEW'07*, 2007. [Cited on pages 18, 27, 28, 38, 39, 40, 220, 223, 224, and 227.]
- [225] M. Hossain, P. Atrey, and A. Saddik. Context-aware QoI computation in multi-sensor systems. In *MASS'08*, 2008. [Cited on pages 2, 39, 40, 43, 66, 220, 223, 224, and 227.]

-
- [226] M. Hossain and A. El Saddik. Quality-driven human-centered approach for service provisioning in ambient environment. In *HCC'08*, 2008. [Cited on pages 26, 27, 28, 31, 38, 220, 223, 224, and 227.]
- [227] S. Hudert, H. Ludwig, and G. Wirtz. Negotiating SLAs—an approach for a generic negotiation framework for WS-agreement. *Journal of Grid Computing*, 7(2):225–246, 2009. [Cited on pages 20 and 203.]
- [228] M. Huebscher and J. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40:7:1–7:28, 2008. [Cited on page 23.]
- [229] M. Humphrey, G. Wasson, K. Jackson, J. Boverhof, M. Rodriguez, J. Gawor, J. Bester, S. Lang, I. Foster, S. Meder, et al. State and events for web services: a comparison of five WS-resource framework and WS-notification implementations. In *HPDC'05*, 2005. [Cited on page 20.]
- [230] U. Hunkeler, H. Truong, and A. Stanford-Clark. MQTT-S—a publish/subscribe protocol for wireless sensor networks. In *COMSWA'08*, 2008. [Cited on page 200.]
- [231] J. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE'05*, 2005. [Cited on page 19.]
- [232] IBM. Seeding the Clouds: Key Infrastructure Elements for Cloud Computing. Technical report, IBM, February 2009. [Cited on page 20.]
- [233] IBM. The smarter supply chain of the future: Insights from the global chief supply chain office study. <http://public.dhe.ibm.com/common/ssi/ecm/en/gbe03163usen/GBE03163USEN.PDF>, 2010. [Cited on page 1.]
- [234] S. Ickin, K. Wac, M. Fiedler, L. Janowski, J. Hong, and A. Dey. Factors influencing quality of experience of commonly used mobile applications. *IEEE Communications Magazine*, 50(4):48–56, 2012. [Cited on pages 26, 28, and 29.]
- [235] D. Jacobi, P. Guerrero, I. Petrov, and A. Buchmann. Distributed network structuring with scopes. Technical report, Technische Universität Darmstadt, 2009. [Cited on pages 18 and 211.]
- [236] S. Jacobi, P. Guerrero, K. Nawaz, C. Seeger, A. Herzog, K. Van Laerhoven, and I. Petrov. *Towards Declarative Query Scoping in Sensor Networks*, volume 6462 of *LNCS*, pages 281–292. Springer, 2010. [Cited on page 18.]
- [237] M. Jaeger. Self-organizing publish/subscribe. In *DSM'05*, 2005. [Cited on page 211.]
- [238] M. Jaeger, G. Mühl, M. Werner, and H. Parzy jegla. Reconfiguring self-stabilizing publish-/subscribe systems. In *DSOM'06*, 2006. [Cited on page 211.]
- [239] M. Jaeger, G. Mühl, M. Werner, H. Parzy jegla, and H. Heiss. Algorithms for reconfiguring self-stabilizing publish/subscribe systems. In *Autonomous Systems – Self-Organization, Management, and Control*, pages 135–147. Springer, 2008. [Cited on pages 210 and 211.]
- [240] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. STAR: Self-tuning aggregation for scalable monitoring. In *VLDB*, 2007. [Cited on pages 93, 103, and 204.]

- [241] N. Jain, P. Mahajan, D. Kit, P. Yalagandula, M. Dahlin, and Y. Zhang. Network imprecision: a new consistency metric for scalable monitoring. In *OSDI'08*, 2008. [Cited on pages 192 and 204.]
- [242] Z. Jerzak. *XSiena: The Content-Based Publish/Subscribe System*. PhD thesis, Dresden University of Technology, 2009. [Cited on page 16.]
- [243] L. Juszczysz, H. Psaier, A. Manzoor, and S. Dustdar. Adaptive query routing on distributed context-the cosine framework. In *MDM'09*, 2009. [Cited on page 30.]
- [244] B. Kahn, D. Strong, and R. Wang. Information quality benchmarks: Product and service performance. *Commun. ACM*, 45(4):184–192, April 2002. [Cited on pages 28, 40, 220, 223, 224, and 227.]
- [245] E. Kalyvianaki, T. Charalambous, M. Fiscato, and P. Pietzuch. Overload management in data stream processing systems with latency guarantees. In *Feedback Computing'12*, 2012. [Cited on page 19.]
- [246] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE'03*, 2003. [Cited on page 19.]
- [247] K. Karimi and G. Atkinson. What the Internet of Things (IoT) needs to become a reality. *White Paper, FreeScale and ARM*, 2013. [Cited on pages 1, 40, 221, 223, 224, and 227.]
- [248] A. Kattepur, N. Georgantas, and V. Issarny. QoS analysis in heterogeneous choreography interactions. In *ICSOC'13*, 2013. [Cited on pages 38, 40, 147, 202, 203, 221, 223, 224, and 227.]
- [249] R. Keeney, H. Raiffa, et al. Decisions with multiple objectives. *Cambridge Books*, 1993. [Cited on pages 46, 47, 48, and 87.]
- [250] K. Keeton, P. Mehra, and J. Wilkes. Do you know your IQ? a research agenda for information quality in systems. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):26–31, 2010. [Cited on pages 1, 2, 3, 28, 38, 40, 46, 61, 202, 203, 221, 223, 225, and 227.]
- [251] I. Kellner and L. Fiege. Viewpoints in complex event processing: industrial experience report. In *DEBS'09*, 2009. [Cited on page 17.]
- [252] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, 1997. [Cited on page 101.]
- [253] M. Kim, K. Karenos, F. Ye, J. Reason, H. Lei, and K. Shagin. Efficacy of techniques for responsiveness in a wide-area publish/subscribe system. In *Middleware'10*, 2010. [Cited on pages 5, 146, and 201.]
- [254] Y. Kim and K. Lee. A quality measurement method of context information in ubiquitous environments. In *ICHIT'06*, 2006. [Cited on pages 18, 35, 38, 221, 223, 225, and 227.]
- [255] A. Kirilenko, A. Kyle, M. Samadi, and T. Tuzun. The flash crash: The impact of high frequency trading on an electronic market. Available at *SSRN 1686004*, 2014. [Cited on page 3.]
- [256] M. Klein, P. Faratin, H. Sayama, and Y. Bar-Yam. Negotiation algorithms for collaborative design settings. In *Complex Engineered Systems*, pages 246–261. Springer, 2006. [Cited on page 218.]

-
- [257] R. Koller, A. Verma, and A. Neogi. WattApp: an application aware power meter for shared data centers. In *ICAC'10*, 2010. [Cited on page 3.]
- [258] S. Kounev. *Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction*. PhD thesis, Technische Universität Darmstadt, 2005. [Cited on page 90.]
- [259] S. Kounev and K. Sachs. Benchmarking and performance modeling of event-based systems. *it - Information Technology*, 51(5):262–269, October 2009. [Cited on pages 169, 171, and 210.]
- [260] J. Kramer. Advanced message queuing protocol (AMQP). *Linux Journal*, (187), 2009. [Cited on page 200.]
- [261] M. Krause and I. Hochstatter. Challenges in modelling and using quality of context (QoC). In *MATA'05*, 2005. [Cited on pages 5, 31, 38, 60, 61, 94, 221, 223, 225, and 227.]
- [262] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *NetDB'11*, 2011. [Cited on page 19.]
- [263] K. Kritikos, B. Pernici, P. Plebani, C. Cappiello, M. Comuzzi, S. Benrernou, I. Brandic, A. Kertész, M. Parkin, and M. Carro. A survey on service quality description. *ACM Computing Surveys*, 46(1):1, 2013. [Cited on pages 28, 35, 36, 37, 221, 223, 225, and 227.]
- [264] R. Kübert, G. Katsaros, and T. Wang. A RESTful implementation of the WS-agreement specification. In *WS-REST'11*, 2011. [Cited on page 20.]
- [265] M. Ladan. Web services metrics: A survey and a classification. *Journal of Communication and Computer*, 9(7):824–829, 2012. [Cited on pages 221, 223, 225, and 227.]
- [266] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: MapReduce-style processing of fast data. In *VLDB'12*, 2012. [Cited on page 19.]
- [267] C. Langguth and H. Schuldt. Extended WS-Agreement protocol to support multi-round negotiations and renegotiations. In *ICSOC'10*, 2010. [Cited on pages 20 and 203.]
- [268] N. Leavitt. Complex-event processing poised for growth. *Computer*, (4):17–20, 2009. [Cited on page 17.]
- [269] K. Lee, C. Kim, S. Lee, and W. Kim. Rateless code based reliable multicast for data distribution service. In *BigComp'15*, 2015. [Cited on pages 5, 29, and 200.]
- [270] B. Li, J. Li, K. Tang, and X. Yao. Many-objective evolutionary algorithms: A survey. *ACM Computing Surveys*, 48(1), 2015. [Cited on page 88.]
- [271] F. Li, S. Nastic, and S. Dustdar. Data quality observation in pervasive environments. In *ICCSE'12*, 2012. [Cited on pages 18, 28, 29, 35, 38, 40, 221, 223, 225, and 227.]
- [272] M. Li, S. Yang, and X. Liu. A performance comparison indicator for pareto front approximations in many-objective optimization. In *GECCO'15*, 2015. [Cited on page 88.]
- [273] S. Li, L. Xu, and S. Zhao. The internet of things: a survey. *Information Systems Frontiers*, 17(2):243–259, April 2014. [Cited on pages 1 and 18.]
- [274] H. Lim, Y. Moon, and E. Bertino. Research issues in data provenance for streaming environments. In *SPRINGL'09*, 2009. [Cited on page 39.]

- [275] H. Lim, Y. Moon, and E. Bertino. Provenance-based trustworthiness assessment in sensor networks. In *DMSN'10*, 2010. [Cited on page 39.]
- [276] M. Lin, S. Li, and A. Whinston. Innovation and price competition in a two-sided market. *Journal of Management Information Systems*, 28(2):171–202, 2011. [Cited on page 218.]
- [277] L. Liu, B. Bamba, M. Doo, P. Pesti, and M. Weber. mTrigger: An event-based framework for location-based mobile triggers. In *Principles and Applications of Distributed Event-Based Systems.*, 2009. [Cited on pages 37 and 46.]
- [278] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *Transactions on Knowledge and Data Engineering*, 11(4):610–628, 1999. [Cited on pages 17 and 19.]
- [279] Q. Liu and K. Serfes. Price discrimination in two-sided markets. *Journal of Economics & Management Strategy*, 22(4):768–786, 2013. [Cited on page 218.]
- [280] S. Liu, S. Homsi, M. Fan, S. Ren, G. Quan, and S. Ren. Power minimization for data center with guaranteed QoS. In *DATE'15*, 2015. [Cited on page 3.]
- [281] Y. Liu and B. Plale. Survey of publish subscribe event systems. Technical report, Indiana University, 2003. [Cited on page 16.]
- [282] M. Löffler and A. Tschiesner. The internet of things and the future of manufacturing. *McKinsey & Company*, 2013. [Cited on page 4.]
- [283] E. Løken. Use of multicriteria decision analysis methods for energy planning problems. *Renewable and Sustainable Energy Reviews*, 11(7):1584–1595, 2007. [Cited on page 88.]
- [284] D. Luckham. *The power of events*. Addison-Wesley, 2002. [Cited on page 17.]
- [285] H. Ludwig, T. Nakata, O. Wälrich, P. Wieder, and W. Ziegler. Reliable orchestration of resources using WS-Agreement. In *HPCC'06*, 2006. [Cited on page 20.]
- [286] J. MacKie-Mason and H. Varian. Pricing congestible network resources. *Selected Areas in Communications*, 13(7):1141–1149, 1995. [Cited on page 78.]
- [287] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: a tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002. [Cited on page 18.]
- [288] S. Mahambre, M. Kumar, and U. Bellur. A taxonomy of QoS-aware, adaptive event-dissemination middleware. *IEEE Internet Computing*, 11(4):35–44, 2007. [Cited on pages 27, 28, 37, 221, 223, 225, and 227.]
- [289] L. Mai, E. Kalyvianaki, and P. Costa. Exploiting time-malleability in cloud-based batch processing systems. In *LADIS'13*, 2013. [Cited on pages 203 and 218.]
- [290] T. Mäkilä, A. Järvi, M. Rönkkö, and J. Nissilä. How to define Software-as-a-Service - an empirical study of finnish SaaS providers. In *ICSOB'10*, 2010. [Cited on page 21.]
- [291] A. Manzoor, H. Truong, and S. Dustdar. On the evaluation of Quality of Context. In *EuroSSC'08*, 2008. [Cited on pages 28, 35, 38, 221, 223, 225, and 227.]
- [292] A. Manzoor, H. Truong, and S. Dustdar. Using quality of context to resolve conflicts in context-aware systems. In *QuaCon'09*, 2009. [Cited on pages 28, 36, 38, 39, and 40.]

- [293] A. Manzoor, H. Truong, and S. Dustdar. Quality of Context: models and applications for context-aware systems in pervasive environments. *The Knowledge Engineering Review*, 29:154–170, 3 2014. [Cited on pages 27, 28, 30, 221, 223, 225, and 227.]
- [294] A. Margara, G. Cugola, and G. Tamburrelli. Learning from the past: Automated rule generation for complex event processing. In *DEBS'14*, 2014. [Cited on page 17.]
- [295] P. Marie, T. Desprats, S. Chabridon, and M. Sibilla. QoCIM: a meta-model for quality of context. In *CONTEXT'13*, 2013. [Cited on pages 146, 202, 221, 223, 225, and 227.]
- [296] P. Marie, L. Lim, A. Manzoor, S. Chabridon, D. Conan, and T. Desprats. QoC-aware context data distribution in the internet of things. In *M4IOT'14*, 2014. [Cited on pages 146, 147, and 202.]
- [297] R. Marler and J. Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004. [Cited on pages 87 and 88.]
- [298] M. Massie. *Monitoring with Ganglia*. O'Reilly, 2012. [Cited on pages 95 and 137.]
- [299] M. Massie, B. Chun, and D. Culler. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, July 2004. [Cited on page 95.]
- [300] M. Mathew, N. Weng, and L. Vespa. Quality-of-information modeling and adapting for delay-sensitive sensor network applications. In *PCCC'12*, 2012. [Cited on page 18.]
- [301] F. Mattern and C. Floerkemeier. *From the Internet of Computers to the Internet of Things*, volume 6462 of *LNCS*, pages 242–259. Springer, 2010. [Cited on page 22.]
- [302] J. Matuschek. Evaluating intelligent routing mechanisms in distributed message-oriented middleware. Master's thesis, Technische Universität Darmstadt, 2012. [Cited on pages 11 and 126.]
- [303] P. Mell and T. Grance. The NIST definition of cloud computing. Technical report, National Institute of Standards and Technology, Information Technology Laboratory, July 2009. [Cited on pages 20, 21, and 22.]
- [304] M. Mendes. FINCoS - benchmarking tools for complex event processing (CEP) systems - google project hosting. <http://code.google.com/p/fincos/>. [Cited on page 95.]
- [305] M. Mendes, P. Bizarro, and P. Marques. A framework for performance evaluation of complex event processing systems. In *DEBS'08*, 2008. [Cited on pages 90, 140, and 143.]
- [306] A. Menditto, M. Patriarca, and B. Magnusson. Understanding the meaning of accuracy, trueness and precision. *Accreditation and quality assurance*, 12(1):45–47, 2007. [Cited on page 38.]
- [307] A. Michlmayr. *Event Processing in QoS-Aware Service Runtime Environments*. PhD thesis, Vienna University of Technology, 2010. [Cited on pages 20 and 203.]
- [308] A. Michlmayr, P. Leitner, F. Rosenberg, and S. Dustdar. Publish/subscribe in the VRESCo SOA runtime. In *DEBS'08*, 2008. [Cited on page 203.]

-
- [309] A. Michlmayr, P. Leitner, F. Rosenberg, and S. Dustdar. Event processing in web service runtime environments. In *Principles and Applications of Distributed Event-Based Systems*, pages 284–306. Information Science Reference, 2010. [Cited on page 20.]
- [310] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. Comprehensive QoS monitoring of web services and event-based SLA violation detection. In *MWSOC'09*, 2009. [Cited on page 203.]
- [311] M. Migliavacca and G. Cugola. Adapting publish-subscribe routing to traffic demands. In *DEBS'07*, 2007. [Cited on pages 90 and 211.]
- [312] P. Missier, K. Belhajjame, and J. Cheney. The W3C PROV family of specifications for modelling provenance metadata. In *EDBT'13*, 2013. [Cited on page 39.]
- [313] G. Mone. Redesigning the data center. *Communications of the ACM*, 55(10):14–16, 2012. [Cited on pages 3 and 4.]
- [314] L. Moreau, P. Groth, S. Miles, J. Vazquez-Salceda, J. Ibbotson, S. Jiang, S. Munroe, O. Rana, A. Schreiber, V. Tan, et al. The provenance of electronic data. *Communications of the ACM*, 51(4):52–58, 2008. [Cited on page 39.]
- [315] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for WS-BPEL. In *WWW'08*, 2008. [Cited on pages 3 and 203.]
- [316] O. Moser, F. Rosenberg, and S. Dustdar. VieDAME - flexible and robust BPEL processes through monitoring and adaptation. In *ICSE'08*, 2008. [Cited on pages 203 and 204.]
- [317] G. Mühl. *Large-scale content-based publish-subscribe systems*. PhD thesis, Technische Universität Darmstadt, 2002. [Cited on pages 15 and 210.]
- [318] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-based Systems*. Springer, 2006. [Cited on pages 14, 15, 16, 28, 201, 211, 221, 223, 225, and 227.]
- [319] G. Mühl, M. Jaeger, K. Herrmann, T. Weis, A. Ulbrich, and L. Fiege. Self-stabilizing publish/subscribe systems: Algorithms and evaluation. In *Euro-Par'05*, 2005. [Cited on pages 210 and 211.]
- [320] G. Mühl, M. Werner, M. A. Jaeger, K. Herrmann, and H. Parzy jegla. On the definitions of self-managing and self-organizing systems. In *KiVS'07*, 2007. [Cited on pages 210 and 211.]
- [321] E. Munera, J. Poza-Lujan, J. Posadas-Yagüe, M. Muñoz, and J. Blanes Noguera. Poster: Context-aware adaptation mechanism for smart resources. In *SensSys'15*, 2015. [Cited on pages 221, 223, 225, and 227.]
- [322] L. Narens. On the scales of measurement. *Journal of Mathematical Psychology*, 24(3):249–275, 1981. [Cited on page 41.]
- [323] M. Nasir, G. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. *arXiv preprint arXiv:1510.05714*, 2015. [Cited on page 19.]
- [324] R. Neisse, M. Wegdam, and M. Sinderen. Trustworthiness and quality of context information. In *ICYCS'08*, 2008. [Cited on pages 26, 27, 28, 38, 39, 45, 60, 221, 223, 225, and 227.]

- [325] H. Nguyen, E. Munthe-Kaas, and T. Plagemann. Energy saving for activity recognition through sensor selection, scheduling and sampling rate tuning. In *WMNC'14*, 2014. [Cited on pages 5, 18, 61, 221, 223, 225, and 227.]
- [326] OASIS. OASIS Advanced Message Queuing Protocol (AMQP) version 1.0. <http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf>, 2011. [Cited on page 200.]
- [327] OASIS. MQTT version 3.1.1 oasis standard. <http://docs.oasis-open.org/mqtt/mqtt-v3.1.1/os/mqtt-v3.1.1-os.pdf>, 2014. [Cited on page 200.]
- [328] J. O'Hara. Toward a commodity enterprise middleware. *Queue*, 5(4):48–55, 2007. [Cited on page 200.]
- [329] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *SOSP'93*, 1993. [Cited on page 15.]
- [330] F. Osinga. Getting 'a discourse on winning and losing': A primer on boyd's theory of intellectual evolution. *Contemporary Security Policy*, 34(3):603–624, 2013. [Cited on page 79.]
- [331] M. Palacios, J. García-Fanjul, J. Tuya, and G. Spanoudakis. Automatic test case generation for WS-Agreements using combinatorial testing. *Computer Standards & Interfaces*, 38:84–100, 2015. [Cited on page 20.]
- [332] M. Palacios, J. Garcia-Fanjul, J. Tuya, and G. Spanoudakis. Coverage-based testing for service level agreements. *IEEE Services Computing*, 8(2):299–313, March 2015. [Cited on page 20.]
- [333] M. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE'03*, 2003. [Cited on page 19.]
- [334] G. Pardo-Castellote. OMG Data-Distribution Service: architectural overview. In *ICD-CSW'03*, 2003. [Cited on pages 5, 29, 146, 200, 221, 223, 225, and 227.]
- [335] G. Parker and M. Van Alstyne. Two-sided network effects: A theory of information product design. *Management Science*, 51(10):1494–1504, 2005. [Cited on page 218.]
- [336] H. Parzy jegla, G. Mühl, and M. Jaeger. Reconfiguring publish/subscribe overlay topologies. In *ICDCSW'06*, 2006. [Cited on page 210.]
- [337] B. Patel and C. Bisdikian. End-station performance under leaky bucket traffic shaping. *Network*, 10(5):40–47, 1996. [Cited on page 93.]
- [338] C. Patel and A. Shah. Cost model for planning, development and operation of a data center. Technical report, HP Laboratories Palo Alto, 2005. [Cited on page 22.]
- [339] V. Pejovic and M. Musolesi. Anticipatory mobile computing: A survey of the state of the art and research challenges. *ACM Comput. Surv.*, 47(3):47:1–47:29, April 2015. [Cited on pages 28, 35, 72, 221, 223, 225, and 227.]
- [340] C. Perera, A. Zaslavsky, P. Christen, M. Compton, and D. Georgakopoulos. Context-aware sensor search, selection and ranking model for internet of things middleware. In *MDM'13*, 2013. [Cited on pages 18, 27, 28, 34, 35, 46, 47, 201, 203, 221, 223, 225, and 227.]
- [341] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. CA4IOT: Context awareness for internet of things. In *GreenCom'12*, 2012. [Cited on pages 1 and 202.]

- [342] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. Context Aware Computing for The Internet of Things: A Survey. *IEEE Communications Surveys & Tutorials*, 16(1):414–454, 2014. [Cited on pages 18 and 202.]
- [343] C. Perera, A. Zaslavsky, P. Christen, A. Salehi, and D. Georgakopoulos. Capturing sensor data from mobile phones using global sensor network middleware. In *PIMRC’12*, 2012. [Cited on pages 18 and 61.]
- [344] C. Perera, A. Zaslavsky, P. Christen, A. Salehi, and D. Georgakopoulos. Connecting mobile things to global sensor network middleware using system-generated wrappers. In *MobiDE’12*, 2012. [Cited on page 18.]
- [345] H. Pérez and J. Gutiérrez. Modeling the QoS parameters of DDS for event-driven real-time applications. *Journal of Systems and Software*, 104:126–140, 2015. [Cited on pages 5, 29, 221, 223, 225, and 227.]
- [346] B. Pernici and S. Siadat. Adaptation of web services based on QoS satisfaction. In *IC-SOC’10*, 2010. [Cited on pages 47, 88, 147, 203, 221, 223, 225, and 227.]
- [347] B. Pernici, S. Siadat, S. Benbernou, and M. Ouziri. A penalty-based approach for QoS dissatisfaction using fuzzy rules. In *ICSOC’11*, 2011. [Cited on page 203.]
- [348] P. Pietzuch. *Hermes: A scalable event-based middleware*. PhD thesis, University of Cambridge Cambridge, UK, 2004. [Cited on page 201.]
- [349] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW’02*, 2002. [Cited on page 16.]
- [350] P. Pietzuch, D. Evers, S. Kounev, and B. Shand. Towards a common API for publish/subscribe. In *DEBS’07*, 2007. [Cited on page 15.]
- [351] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE’06*, 2006. [Cited on page 19.]
- [352] J. Pike. National image interpretability rating scales. <http://fas.org/irp/imint/niirs.htm>, 1998. [Cited on page 34.]
- [353] L. Pinto, G. Cugola, and C. Ghezzi. Dealing with changes in service orchestrations. *SAC’12*, 2012. [Cited on pages 22 and 68.]
- [354] I. Podnar Zarko, A. Antonic, and K. Pripužić. Publish/subscribe middleware for energy-efficient mobile crowdsensing. In *UbiComp’13*, 2013. [Cited on pages 18, 27, 34, 61, 221, 223, 225, and 227.]
- [355] G. Pottie. Wireless sensor networks. In *ITW’98*, 1998. [Cited on pages 17 and 18.]
- [356] M. Razzaque, C. Bleakley, and S. Dobson. Compression in wireless sensor networks: A survey and comparative evaluation. *ACM Transactions on Sensor Networks*, 10(1):5, 2013. [Cited on pages 18, 34, 36, 38, 43, 61, and 63.]
- [357] P. Reinecke, K. Wolter, and M. Malek. A survey on fault-models for QoS studies of service-oriented systems. Technical report, Freie Universität Berlin & Humboldt-Universität zu Berlin, 2010. [Cited on pages 35, 221, 223, 225, and 227.]

-
- [358] J. Rochet and J. Tirole. Platform competition in two-sided markets. *Journal of the European Economic Association*, pages 990–1029, 2003. [Cited on page 218.]
- [359] J. Rochet and J. Tirole. Defining two-sided markets. Technical report, IDEI and GREMAQ, 2004. [Cited on page 218.]
- [360] J. Rochet and J. Tirole. Two-sided markets: an overview. Technical report, IDEI and GREMAQ, 2004. [Cited on page 218.]
- [361] J. Rochet and J. Tirole. Two-sided markets: a progress report. *The RAND Journal of Economics*, 37(3):645–667, 2006. [Cited on page 218.]
- [362] O. Rorato, S. Bertoldo, C. Lucianaz, M. Allegretti, and G. Perona. A multipurpose node for low cost wireless sensor network. In *APWC’12*, 2012. [Cited on pages 60 and 94.]
- [363] D. Rosenblum and A. Wolf. A design framework for internet-scale event observation and notification. In *ESEC’97*, 1997. [Cited on page 15.]
- [364] J. Ross and G. Westerman. Preparing for utility computing: The role of IT architecture and relationship management. *IBM Systems Journal*, 43(1):5–19, 2004. [Cited on page 21.]
- [365] J. Sacha, J. Napper, C. Stratan, and G. Pierre. Adam2: Reliable distribution estimation in decentralised environments. In *ICDCS’10*, 2010. [Cited on page 73.]
- [366] V. Sachidananda, A. Khelil, and N. Suri. Quality of information in wireless sensor networks: A survey. In *ICIQ’10*, 2010. [Cited on pages 28, 31, 40, 221, 223, 225, and 227.]
- [367] K. Sachs. *Performance Modeling and Benchmarking of Event-Based Systems*. PhD thesis, Technische Universität Darmstadt, 2010. [Cited on pages 90 and 210.]
- [368] K. Sachs, S. Appel, S. Kounev, and A. Buchmann. Benchmarking publish/subscribe-based messaging systems. In *DASFAA’10*, 2010. [Cited on pages 16, 140, 143, 169, 171, 174, and 244.]
- [369] K. Sachs, S. Kounev, S. Appel, and A. Buchmann. A performance test harness for publish/subscribe middleware. In *SIGMETRICS/Performance’09*, 2009. [Cited on pages 140, 169, and 171.]
- [370] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann. Workload characterization of the SPECjms2007 benchmark. In *EPEW’07*, 2007. [Cited on pages 169 and 170.]
- [371] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Performance Evaluation*, 66(8):410–434, 2009. [Cited on pages 169 and 171.]
- [372] K. Sachs, S. Kounev, M. Carter, and A. Buchmann. Designing a workload scenario for benchmarking message-oriented middleware. In *SPEC Benchmark Workshop’07*, 2007. [Cited on pages 169 and 170.]
- [373] T. Sandholm and K. Lai. Dynamic proportional share scheduling in Hadoop. In *JSSPP’10*, 2010. [Cited on page 218.]
- [374] J. Schiller and A. Voisard. *Location-based services*. Elsevier, 2004. [Cited on page 28.]
- [375] A. Schröter, D. Graff, G. Mühl, J. Richling, and H. Parzy jegla. Self-optimizing hybrid routing in publish/subscribe systems. In *DSOM’09*. Springer, 2009. [Cited on pages 210 and 211.]

- [376] C. Seeger. *Event-driven Middleware for Body and Ambient Sensor Applications*. PhD thesis, Technische Universität Darmstadt, 2013. [Cited on page 18.]
- [377] C. Seeger, A. Buchmann, and K. Van Laerhoven. Wireless sensor networks in the wild: Three practical issues after a middleware deployment. In *MidSens'11*, 2011. [Cited on page 18.]
- [378] C. Seeger and K. Van Laerhoven. Sensor network middleware that mediates between sensors and healthcare applications. *Life Sciences Newsletter*, November 2013. [Cited on page 18.]
- [379] C. Seeger, K. Van Laerhoven, and A. Buchmann. MyHealthAssistant: An event-driven middleware for multiple medical applications on a smartphone-mediated body sensor network. *Biomedical and Health Informatics*, 19(2):752 – 760, 2014. [Cited on page 18.]
- [380] S. Sharaf and K. Djemame. Enabling service-level agreement renegotiation through extending WS-agreement specification. *Service Oriented Computing and Applications*, 9(2):177–191, 2015. [Cited on pages 20 and 203.]
- [381] S. Shenker. Fundamental design issues for the future internet. *Selected Areas in Communications*, 13(7):1176–1188, 1995. [Cited on pages 29, 46, 47, 53, 78, 87, and 88.]
- [382] S. Shenker. Service models and pricing policies for an integrated services internet. *Public access to the Internet*, pages 315–337, 1995. [Cited on page 78.]
- [383] S. Shenker, D. Clark, D. Estrin, and S. Herzog. Pricing in computer networks: Reshaping the research agenda. *SIGCOMM Comput. Commun. Rev.*, 26(2):19–43, 1996. [Cited on page 78.]
- [384] Y. Shi and X. Chen. A survey on QoS-aware web service composition. In *MINES'11*, 2011. [Cited on pages 34, 39, 61, 221, 223, 225, and 227.]
- [385] Y. Simmhan, B. Plale, and D. Gannon. A survey of data provenance techniques. Technical report, Indiana University, 2005. [Cited on page 39.]
- [386] L. Skorin-Kapov, K. Pripuzic, M. Marjanovic, A. Antonic, and I. Zarko. Energy efficient and quality-driven continuous sensor management for mobile IoT applications. In *Collab-orateCom'14*, 2014. [Cited on pages 18, 34, 35, 61, 72, 221, 223, 225, and 227.]
- [387] A. Slominski, Y. Simmhan, A. Rossi, M. Farrellee, and D. Gannon. XEVENTS/XMESSAGES: Application events and messaging framework for Grid. Technical report, Indiana University, 2001. [Cited on page 16.]
- [388] M. Smit, A. Nisbet, E. Stroulia, A. Edgar, G. Iszlai, and M. Litoiu. Capacity planning for service-oriented architectures. In *CASCON'08*, 2008. [Cited on page 90.]
- [389] M. Smit, B. Simmons, and M. Litoiu. Distributed, application-level monitoring for heterogeneous clouds using stream processing. In *Future Generation Computer Systems*, volume 29, pages 2103–2114. Elsevier, 2013. [Cited on page 203.]
- [390] B. Snyder, D. Bosanac, and R. Davies. *ActiveMQ in Action*. Manning Publications Co., 2011. [Cited on pages 16 and 125.]
- [391] J. Søberg. *CommonSens: A Multimodal Complex Event Processing System for Automated Home Care*. PhD thesis, University of Oslo, 2011. [Cited on page 201.]

-
- [392] J. Søberg, V. Goebel, and T. Plagemann. CommonSens: Personalisation of complex event processing in automated homecare. In *ISSNIP'10*, 2010. [Cited on pages 17, 18, and 201.]
- [393] J. Søberg, V. Goebel, and T. Plagemann. Detection of spatial events in CommonSens. In *EiMM'10*, 2010. [Cited on pages 17 and 201.]
- [394] J. Søberg, V. Goebel, and T. Plagemann. Deviation detection in automated home care using CommonSens. In *PERCOM'11 Workshops*, 2011. [Cited on page 201.]
- [395] SPEC. SPECjms2007 result ActiveMQ 5.4 on IBM x3850. <https://www.spec.org/jms2007/results/res2010q3/jms2007-20100802-00022.html>. [Cited on pages 171 and 244.]
- [396] J. Steffan, L. Fiege, M. Cilia, and A. Buchmann. Scoping in wireless sensor networks. In *MPAC'04*, 2004. [Cited on page 18.]
- [397] J. Steffan, L. Fiege, M. Cilia, and A. Buchmann. Towards multi-purpose wireless sensor networks. In *SENET'05*, 2005. [Cited on pages 17 and 18.]
- [398] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997. [Cited on page 19.]
- [399] S. Stepney, F. Polack, and H. Turner. Engineering emergence. In *ICECCS'06*, 2006. [Cited on page 23.]
- [400] S. Stevens. On the theory of scales of measurement. *Science*, 103(2684):677–680, 1946. [Cited on pages 32 and 41.]
- [401] A. Stühlmeyer. Erweiterbares Framework zur Ausführung von Aktionen für Event-basierte Enterprise Software Systeme unter Verwendung von Apama. Bachelor's thesis, Technische Universität Darmstadt and Software AG Darmstadt, 2014. [Cited on pages 11 and 120.]
- [402] R. Sumathi and M. Srinivas. A survey of QoS based routing protocols for wireless sensor networks. *Journal of information processing systems*, 8(4):589–602, 2012. [Cited on pages 18 and 26.]
- [403] H. Sun, T. Zhao, Y. Tang, and X. Liu. A QoS-aware load balancing policy in multi-tenancy environment. In *SOSE'14*, 2014. [Cited on pages 221, 223, 225, and 227.]
- [404] Sun Microsystems, Inc. Java Message Service (JMS) Specification - Ver. 1.1, 2002. [Cited on pages 16 and 200.]
- [405] W. Tam, K. Lui, S. Uludag, and K. Nahrstedt. Quality-of-service routing with path information aggregation. *Computer Networks*, 51(12):3574–3594, August 2007. [Cited on page 93.]
- [406] W. Terpstra, C. Leng, M. Lehn, and A. Buchmann. Channel-based unidirectional stream protocol (CUSP). In *INFOCOM'10*, 2010. [Cited on page 210.]
- [407] R. Tews. Beyond IT: exploring the business value of SOA. *AIIM E-DOC*, 2007. [Cited on page 19.]
- [408] W. Thomson et al. *Popular lectures and addresses*, volume 1. Macmillan London, 1891. [Cited on page 1.]

- [409] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@twitter. In *SIGMOD'14*, 2014. [Cited on pages 19 and 203.]
- [410] E. Twellmeyer. Modeling and performance evaluation of an RFID scenario applying EPC-global's EPCIS architecture. Master's thesis, Technische Universität Darmstadt and SAP Research Dresden, 2008. [Cited on pages 90 and 210.]
- [411] G. Tychogiorgos and C. Bisdikian. Selecting relevant sensor providers for meeting your quality information needs. In *MDM'11*, 2011. [Cited on page 18.]
- [412] R. Van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003. [Cited on page 204.]
- [413] Verizon. State of the market: The internet of things 2015. Technical report, 2015. [Cited on page 1.]
- [414] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *EuroSys'15*, 2015. [Cited on page 3.]
- [415] N. Viet, E. Munthe-Kaas, and T. Plagemann. Towards quality and energy aware complex event processing. In *WiMOB'13*, 2013. [Cited on pages 5, 34, 35, 60, 61, 94, 221, 223, 225, and 227.]
- [416] S. Vinoski. Web services notifications. *IEEE Internet Computing*, 8(2):86–90, 2004. [Cited on page 20.]
- [417] S. Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, (6):87–89, 2006. [Cited on page 200.]
- [418] W3C Semantic Sensor Network Incubator Group. Semantic Sensor Network Ontology. <http://www.w3.org/2005/Incubator/ssn/ssnx/ssn>, 2011. [Cited on pages 5, 18, 34, 35, 37, 38, 45, 61, 64, and 201.]
- [419] G. Wang, C. Wang, A. Chen, H. Wang, C. Fung, S. Uczekaj, Y. Chen, W. Guthmiller, and J. Lee. Service level management using QoS monitoring, diagnostics, and adaptation for networked enterprise systems. In *EDOC'05*, 2005. [Cited on page 20.]
- [420] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou. Distributed systems meet economics: pricing in the cloud. In *HotCloud'10*, 2010. [Cited on page 218.]
- [421] R. Wang and D. Strong. Beyond accuracy: What data quality means to data consumers. *Journal of management information systems*, pages 5–33, 1996. [Cited on pages 2, 26, and 40.]
- [422] R. Wearn and N. Larson. Measurements of the sensitivities and drift of digiquartz pressure sensors. *Deep Sea Research Part A. Oceanographic Research Papers*, 29(1):111–134, 1982. [Cited on pages 18 and 34.]
- [423] R. Welke, R. Hirschheim, and A. Schwarz. Service oriented architecture maturity. *Computer*, 44(2):61–67, February 2011. [Cited on page 19.]
- [424] J. Wilkes. Utility functions, prices, and negotiation. In *Market Oriented Grid and Utility Computing*. Wiley & Sons, 2008. [Cited on pages 2, 46, 47, 51, 54, 61, 62, 78, 87, and 202.]

-
- [425] Q. Wu, G. Ding, Y. Xu, S. Feng, Z. Du, J. Wang, and K. Long. Cognitive Internet of Things: A new paradigm beyond connection. *IEEE Internet of Things Journal*, 1(2):129–143, April 2014. [Cited on pages 26, 28, 221, 223, 225, and 227.]
- [426] R. Würtz. *Organic computing*. Springer, 2008. [Cited on page 23.]
- [427] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM'04*, 2004. [Cited on pages 73 and 204.]
- [428] H. Yang, M. Kim, K. Karenos, F. Ye, and H. Lei. Message-oriented middleware with QoS awareness. In *ICSOC'09*, 2009. [Cited on pages 5, 29, 146, 201, 221, 223, 225, and 227.]
- [429] Y. Yoon. *Adaptation Techniques for Publish/Subscribe Overlays*. PhD thesis, University of Toronto, 2013. [Cited on page 210.]
- [430] S. Zahedi, M. Srivastava, and C. Bisdikian. A computational framework for quality of information analysis for detection-oriented sensor networks. In *MILCOM'08*, 2008. [Cited on pages 38, 221, 223, 225, and 227.]
- [431] P. Zappi, C. Lombriser, T. Stiefmeier, E. Farella, D. Roggen, L. Benini, and G. Tröster. Activity recognition from on-body sensors: accuracy-power trade-off by dynamic sensor selection. In *EWSN'08*, 2008. [Cited on page 18.]
- [432] A. Zeidler and L. Fiege. Mobility support with rebeca. In *ICDCS MCM'03*, 2003. [Cited on page 16.]
- [433] L. Zeng, H. Lei, and H. Chang. Monitoring the QoS for web services. In *ICSOC'07*, 2010. [Cited on page 20.]
- [434] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010. [Cited on page 20.]
- [435] F. Zhao and L. Guibas. *Wireless sensor networks: an information processing approach*. Morgan Kaufmann, 2004. [Cited on page 18.]
- [436] W. Ziegler, P. Wieder, and D. Battro. Extending WS-Agreement for dynamic negotiation of service level agreements. Technical report, CoreGRID-Network of Excellence, 2008. [Cited on pages 20 and 203.]
- [437] E. Zitzler. *Evolutionary algorithms for multiobjective optimization: Methods and applications*. PhD thesis, ETH Zurich, 1999. [Cited on pages 87 and 88.]